Polaris:
A New-Generation Parallelizing Compiler for MPPs

R & D Status Report

June 15, 1993

David A. Padua

Rudolf Eigenmann

Jay Hoeflinger

Paul Petersen

Peng Tu

Stephen Weatherford

Keith Faigin

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign.

CSRD Report No. 1306

1 Introduction

The goal of the project is to develop an industrial-strength Fortran compiler for massively parallel processors (MPPs) with a global address space. The system, called Polaris, is depicted in Figure 1. Polaris will accept programs written in an HPF dialect and will generate code in explicitly parallel form. The system will incorporate the KAP, a commercial compiler extended with a collection of modules being developed at the University of Illinois at Urbana-Champaign. However, we also plan to integrate the Illinois modules into a free-standing compiler.

Even though HPF includes directives to specify parallelism and data distribution, the compiler will include the necessary transformation modules to perform automatically both data distribution and parallelization and, thus, make the necessity of directives as infrequent as possible. This is one of the central goals of our compiler for massively parallel machines because we believe that data distribution and parallelization are optimization problems, which are clearly amenable to automatization.

In this report we discuss the machines we are considering as targets of the compiler, the internal representation used by the Illinois modules, the mechanism we have designed to interface with KAP, the parallel program repository, and some of the compiler transformations we are implementing. Details of several of the topics discussed are presented in the appendices. The last two sections of this report discuss the meetings we attended that are related to this project and the papers published during the last quarter.

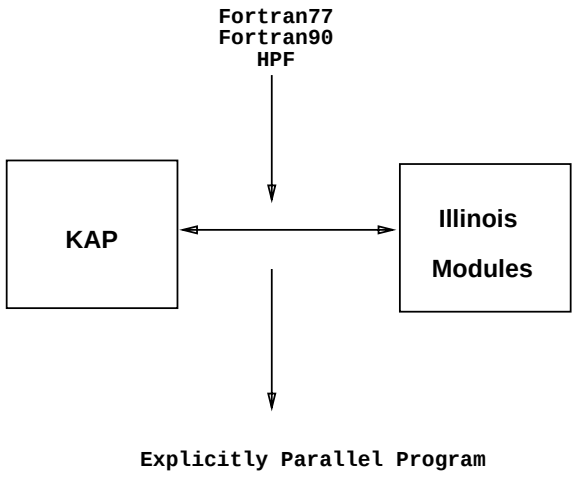

Figure 1: The Polaris System

2 Target Machine

In this project, we plan to generate explicitly parallel code that will be generic so that it can exploit parallelism on a variety of global address space MPPs. However, we would like to select one or two target machines to obtain realistic and believable measurements of the effectiveness of our techniques. We have considered two such

machines: the Kendall Square Research KSR-1 and the Cray T3D. We experimented for a few weeks with the KSR-1 multiprocessor at Cornell by translating by hand several parallel programs from our current repository and measuring the speedups. (We give more details on this repository below.) Even though we succeeded in obtaining good speedups in some cases, we found that the KSR-1 presents some difficulties that we hope will be overcome with new releases of the software in the near future. We have studied the Cray T3D and discussed its main characteristics with the people at Cray Research Inc. However, we have had no experience with the machine. We plan to start running experiments sometime in July 1993 on an emulator of the T3D available on the Cray C-90.

3 Internal Representation

The modules under development at the University of Illinois at Urbana-Champaign are being written in C++. We have designed and fully implemented a class hierarchy to represent and manipulate Fortran programs and related information such as control flow and data dependence. This class hierarchy, described in Appendix A, facilitates the implementation of compiler transformations in several ways: (1) By decreasing the number of programming errors. For example, in traditional representations updating pointers in the internal representation when the program is modified is usually the responsibility of the programmer, and this is frequently the source of errors. However, in our internal representation, all operations on the program are high-level operations where the details related to pointer updating are hidden from the programmer and performed by utility routines that have been extensively debugged. (2) The class hierarchy also makes the transformation and analysis routines compact because high-level operations are available to the programmer. (3) The class hierarchy enforces a discipline of development by requiring programmers to use the utilities it provides and in this way precludes the development of redundant code, as is usually the case in traditional implementations.

4 Exchange Format

To interface the routines developed at Illinois with KAP, we have designed, in conjunction with Kuck and Associates, Inc., an exchange format. This format is described in Appendix B. Before control is transferred from KAP to one of the Illinois modules, KAP's internal representation of the program will be translated into the exchange format and then translated into the Illinois internal representation discussed above. Similarly, two translations would take place where control is transferred from the Illinois modules to KAP.

Even though two translations would be necessary when control is translated between KAP and the C++ modules, we found this approach preferable to a direct translation for several reasons. First, development and debugging is simplified because by visualizing the program represented in the exchange format we would be able to rapidly identify the source of errors in the communication between the systems. Second, it is easier for KAI to insulate their proprietary software from the public-domain routines developed at Illinois. Finally, the exchange format provides an ideal mechanism for interfacing our project with modules developed by other

research groups using different internal representations.

The extra overhead associated with the two translations should not have an important effect on performance because we expect only a few changes of control between the Illinois modules and KAP and because the translation into and from the exchange format should take only a small fraction of time consumed by the analysis and transformation modules. We expect to initiate the coding of the exchange format translators sometime in July 1993.

5 The Parallel Program Repository

In work conducted by our group before the beginning of this project, we developed by hand parallel versions of 12 programs in the Perfect Benchmarks^{TM}. Under the new projectFortran, we have automatically transformed these programs into two other parallel forms. One is the Fortran dialect developed for the Cray T3D, and the other is KSR Fortran. We have used and plan to continue using these programs in our experiments with these machines and as a yardstick for our compiler transformation and analysis algorithms. Some of the uses we have made of the program repository are discussed in section 6 and in the appendices.

We plan to extend this repository with other programs gathered from different sources throughout the life of the project.

6 Compiler Transformations and Their Effectiveness

We have completed the implementation of prototypes of a few transformation and analysis modules that will form the basis of the final compiler. These are array privatization, procedure integration, constant propagation, and data dependence analysis. We are in the process of coding these and other modules (including induction variable recognition, reduction variable recognition, symbolic analysis) in its final C++ form using the hierarchy of classes discussed above. We expect to complete the implementation of all these modules within the next quarter.

We discuss next our work on array privatization, dependence analysis, and data locality.

6.1 Array Privatization

¿From our previous work in this area we know that array privatization is an important transformation needed not only to enhance parallelism but also to increase locality and decrease communication costs on compilers for machines with physically distributed memory.

We have implemented an algorithm for array privatization, which is described in Appendix C. Also, we have applied the prototype versions of the transformations to the original sequential programs from where we derived our repository. As can be seen in Appendix C, the algorithm is very effective in that it recognizes automatically most of the arrays manually recognized as privatizable. However, in a few cases the automatic techniques fail to reproduce the manual results and therefore further enhancements are needed. As discussed in Appendix C, these enhancements require

simple symbolic analysis of the relation between program variables whose value is unknown at compile-time. Routines for symbolic analysis are under implementation.

6.2 Data Dependence Analysis

Accurate dependence analysis is crucial to many compiler transformations, especially those related to automatic parallelization. We have conducted experiments using our parallel program repository to determine how accurate are traditional dependence analysis techniques are in detecting parallel loops in sequential programs. The parallel program repository is very useful because it allows us to separate the dependence analysis per se from transformations such as variable privatization and induction variable recognition.

Our plan is to use the information gathered in these experiments described in Appedix D, to design a combination of run-time and compile-time strategies that would make possible an accurate dependence analysis and, as a consequence, an effective parallelization.

6.3 Data Distribution and Program Locality

We have designed a preliminary strategy for automatic data distribution and continue work in this area. Also, a tool has been developed to instrument programs written in Cray MPP Fortran to count the number of memory references to the different memory modules from every processor. We plan to use this tool to make a first evaluation of our automatic data distribution techniques once they are implemented.

Appendix A: Implementation of Polaris' Internal Representation

Goals

Two main goals drew our attention in the implementation of Polaris' internal data representation, both of which came from our experience with the Delta Program Manipulation System (which was implemented in a very high level, but rather slow, language called SETL). The first of these is robustness and ease of correct programming. We have placed much attention on creating a system which preferably avoids but at least detects and reports as many possible system and user errors as possible, and as early in the production process as possible. The second goal is run-time efficiency, both for production reasons and also because of the need for practical debugging capability. If the system is too slow, debugging becomes a problem when large test cases are common.

In order to create a robust system which is also reasonably efficient led us to the decision to use C++ as the implementation language. The object-oriented aspect of C++ fits naturally with our desire to keep the operations inside the same module as the data structures, while the tendency of C++'s design towards efficiency allows us to implement a robust system without being overly worried about the usability problems that an less-efficient language would cause. Some of the features which we have implemented in order to realize our goal of robustness include

- requiring all data accesses to go through class methods, or functions (this helps keeps the interface consistent even if the implementation drastically changes)
- detection of aliased structures (structure sharing is not allowed) in the **Expression** trees and other structures, and reporting their existence with a run-time error. For example, it would be an error to create a new **Expression** object and insert it into two different Statements without first cloning the object.
- detection of the premature destruction of an object if its destructor is called before it is correctly removed from a structure (which could happen, for instance, if an object allocated on the stack is placed into an Expression tree, as well as for other more difficult-to-track reasons)
- error avoidance or checking extensively throughout the system, including
  - the avoidance of dangling pointer problems through reference counting (which can also make programming easier)
  - the quick detection of errors through the liberal use of assertions. I.e. within the data structures code, if any condition or system state is assumed by the programmer, that assumption is specified explicitly in a `p_assert()` (short for "**Polaris** assertion") statement which checks the assumed condition and reports an error if the assumption is incorrect. Whenever an error is discovered anywhere in the system, the C++ operation aborts and returns control to the calling code. An exact error message is reported as well as a dump of the calling sequence for debugging purposes.

- all methods declared in the major base classes of a class hierarchy to avoid the dangerous overuse of typecasting when referencing derived classes (see the discussion of the **Statement** class, below)
- use of template classes to strengthen type checking (see discussion at the end of this section)
- allowing only internally-consistent incremental changes outside the base system and keeping the external system state always consistent and correct. For example, statement blocks inserted into a **StatementList** object are required to be well-formed with respect to multi-statement constructs, meaning that, for instance, a DO statement cannot be inserted separately from its matching ENDDO statement, since the statement list would then be left in an incomplete and inconsistent state.
- automatically updating information derived from the program as far as practical, such as statement flow graph information and data dependence information
- hiding internal structure details which are not necessary for the user to see or alter
- supplying all commonly-needed methods so that users do not feel the need to reinvent the wheel or meddle with the system
- having all class constructor methods check that all the fields necessary to create a full, consistent object are provided

In addition, one of the features of our system is the conversion between various representations. Currently a conversion process has been implemented between the Delta SETL representation and the Polaris C++ representation. We are currently planning conversion between other internal representations as well, including a transformation to and from KAP's internal data structures.

Support Classes

The underlying support system for the Fortran internal representation is just as important as the representation itself. In order to provide fully complete support for the internal representation as well as user code, we have created an infrastructure of support classes that are heavily used both internally and externally. This infrastructure currently includes

- a **Collection** class hierarchy which includes sets, lists, dictionaries, and maps
- an **Iterator** class for iterating through any **Collection**
- an **Array** class
- a class for checking correctness of memory usage
- a "lazy" memory buffer class for dynamically growing memory buffers
- a dynamic **String** class

- a group of "BinStr" classes which define the interface intrinsics between SETL and C++
- "overflow" classes for special fields in most of the internal representation classes, used to hold unrecognized or temporary data (currently using the **BinStr** interface conventions)
- various other utility functions and classes

Although many of the internal representation classes are derived from the class **Listable** in order to be able to be placed in lists, we rely heavily on class templates in order to gain better compile-time type checking than a system using lists without templates. It is possible, for instance, to specify a list of statement objects by using the construct `List<Statement>` and an iterator over such a list with the construct `Iterator<Statement>`. Without templates, a **List** class could hold any object derived from class **Listable**. With templates, however, we are able to restrict lists to any specific class hierarchy that is desired, thereby increasing compile-time type error detection and control. All of the **Collection** classes, as well as the **Array** class, are class templates.

ProgramUnit Class

The **ProgramUnit** class is mostly a holder for the various data structure elements which make up a Fortran program unit. There is no class hierarchy specified for it, although it may be of any of the following types:

- **BLOCK_DATA_PU_TYPE** — a BLOCK DATA program unit
- **PROGRAM_PU_TYPE** — a main program
- **SUBROUTINE_PU_TYPE** — a subroutine
- **FUNCTION_PU_TYPE** — an external function

The **ProgramUnit** class contains and allows access to its component data structures, which are instances of the following classes:

- **StatementList** — a list of all program unit statements, if any
- **Symtab** — a symbol table of all symbols used in the program unit
- **DataList** — a list of the information contained in this program unit's DATA statements
- **CommonBlockDict** — a dictionary of all common blocks referenced by this program unit
- **EquivalenceDict** — a dictionary of this program unit's variable equivalence classes
- **FormatDict** — a dictionary of this program unit's FORMAT statement information

- **WorkSpaceStack** — a stack of temporary data structures which the user can define and use for a transformation pass (and which he or she is responsible for cleaning up)
- **OverflowMap** — a dictionary for either structures unrecognized by the conversion routines between SETL and C++ or for tuple and set-based experimental or temporary data structures which either are not or do not need to be supported officially

In addition to functions for accessing these data structures inside a **ProgramUnit** object, there are methods for all of the following operations:

- printing or displaying the program unit to any C++ stream (either in Fortran format or with moderate or extensive debugging information)
- copying entire **ProgramUnit** objects
- translating **ProgramUnit** objects to and from the exchange format and the internal representation, for conversion between different internal representations (currently a conversion between the Delta SETL representation and the Polaris C++ representation is supported)
- return the procedure or main program name, if any
- managing the **WorkSpaceStack**

The copy operations are implemented, as with most classes, as overloaded operators and as copy constructors. For instance, the following C++ fragment is legal and produces the expected results (copying and assignment of **ProgramUnit** objects):

```
ProgramUnit pgm1;
  ...
ProgramUnit *pgm2 = new ProgramUnit(pgm1);
ProgramUnit pgm3 = pgm1;
ProgramUnit pgm4;
  ...
pgm4 = pgm1;
```

Some of the more important class structures contained in the **ProgramUnit** class will be discussed below.

Statement Class

We have chosen to implement statements as simple, non-recursive structures. Thus, we have not implemented statement blocks directly. However, we have made the implementation flexible enough that methods which simulate the existence of statement blocks can easily be implemented on top of the current **statement** class.

Statements are implemented by an abstract base statement class which contains all of the structures common to all statements. For each specific type of Fortran statement, a distinct class is derived from the base class which contains structures

found only for that type of statement. This class hierarchy allows modifications and additions to specific statements to be kept local to the statement. In addition, however, if a new method is needed for every statement type, it can be implemented in the base class only.

The base statement class declares many types of information about the statement including statement-types, line numbers and flow and data information. These fields are all accessed through public methods. Among the fields declared in the base statement (and which therefore exist in all statements) are

- sets of successor and predecessor flow links which are implemented in the form of references to statements.
- lists of memory references. These include **in_refs**, **out_refs** and **act_refs** which are respectively memory read and write and access to parameters
- an **outer** link which points to the innermost enclosing DO loop.
- various predefined and user-defined loop information, such as data dependences and loop invariant variables

Whenever practical, we implement the methods such that any modification to a statement or to a statement list results in the updating of information, in order to retain consistency.

Each derived statement class may declare additional fields. Among the most common fields declared by derived statement classes is the **follow** field. Since the statement list is implemented as a singly-nested structure, compound statement types, such as block-IFs, are implemented with multiple statements much like they are expressed in Fortran syntax. Thus, a full block-IF without an ELSEIF clause is represented in Polaris by an **IfStmt** class object followed by some number of statements delimited by an **ElseStmt** which is itself delimited by an **EndIfStmt**. The **follow** field connects the statements of these compound structures. For instance, the **follow** field found in an **IfStmt** would point to the next unit of the block-IF which could be either an **ElseStmt**, an **ElseIfStmt** or an **EndIfStmt**.

The **DoStmt** declares a number of fields in addition to those declared by the base statement type. The **follow** field within a **DoStmt** points to its corresponding **EndDoStmt**, and likewise the **follow** field of an **EndDoStmt** points to the corresponding **DoStmt**. In addition, fields are declared which specify the index of the loop as well as the initial, limit and step expression. Each of these fields is an **Expression** tree.

In order to increase the robustness of the structure, all methods which access data fields are declared within the base statement class and redefined in the derived classes which use them. For example, the methods which access the 'step' field are only applicative to the **DoStmt** but are declared in the base class. The base class definition of the `step()` method, for instance, like all other base method definitions, calls an error-routine while the redefinition in **DoStmt** actually performs the correct operation. With this scheme, if a method is called for a statement to which it is not applicable, an error will be reported and the system can either try to continue or can perform a controlled abort rather than causing an uncontrolled crash.

Although this design has the disadvantage of moving the detection of some errors to run-time, it has two hopefully larger advantages. First, the user in general only

needs to include the header file of the base **Statement** class and not those of the derived classes, since all of the methods he would need are already defined in the base class; this reduces the compile time of user programs, which in turn makes the debugging process easier and friendlier. (The single exception to this rule is that if the user needs to create new statements rather than just modifying current ones, that user must include the appropriate derived class header files in order to access the constructors for that class. Generally only a few such header files, if any, need to be included by a user program.)

The second reason has to do with the fact that the **StatementList** class (and similarly for lists of **Symbol**s and **Expression**s) contains a list of references to the base **Statement** class. By C++'s rules of typing, it is legal for a reference or pointer to a base class to actually point to a derived class, and this capability is used extensively throughout our system. While iterating through a list of **Statement**s, for example, the programmer will receive a reference to the base **Statement** class. Once he has determined the type of **Statement** that reference refers to, he would normally have to then typecast the reference into the correct derived class in order to be able to access the methods appropriate to that statement type. However, we believe that the large number and variety of typecasts required by such a system creates an unnecessarily large possibility for errors made by programmers typecasting to the wrong class type. (These types of errors are especially easy to make as changes are made to the system or to the user program.) Such errors can neither be detected nor controlled by a C++ compiler or by the run-time system itself, and can be extremely difficult to trace. However, by placing all possible methods directly into the base **Statement** class, we gain exact run-time detection and control of errors of this type.

In order to ensure that no incomplete structures can exist within the program, the constructors for statements require all fields needed to completely define the statement. For example, the **DoStmt** constructor requires the statement label—a unique string which identifies the statement—as well as expressions for the index, initial value, limit and step of the loop. Exceptions are made to this rule for optional structures, such as the upper or lower bounds of an array dimension, in which case methods exist to determine if the optional structure does indeed exist in a particular object or not.

As a simple example, the DO statement header

```
DO I = 1, 10, 2
```

can be created (given a **ProgramUnit** `pgm` as reference) with the following constructor call, which gives the new **DoStmt** the label "S10":

```
ProgramUnit pgm;
  ...
Statement *stmt = new DoStmt(``S10'',
                      new IDExpr(pgm.symtab()[``I'']),
                      new IntConstExpr(1),
                      new IntConstExpr(10),
                      new IntConstExpr(2));
```

where the call `pgm.symtab()[``I'']` does a search for the symbol "I" in `pgm`'s

symbol table and returns a reference to the symbol which it finds, and the `new IntConstExpr(integer)` calls create a new integer constant expression with the value `integer`.

StatementList Class

The **StatementList** class is derived from the class template **List<Statement>**, which is simply a list of **Statement** objects. The **StatementList** class, however, overrides many of the basic list operations to include automatic updating of the flow graph whenever any statement or block of statement is deleted, inserted or moved. In addition to this basic functionality expansion, the following types of operations are all available

- returning an iterator over the entire or selected parts of a statement list
- getting the first and last statements or the list length
- finding a statement by its label
- copying, deleting, unlinking or moving any well-formed sublist of statements
- inserting any single statement or any well-formed list of statements
- inserting specific multi-block statement groups, such as a block-IF statement framework
- print to any C++ stream a Fortran or debugging display of all statements in the list

To maintain complete control of consistency inside the **StatementList** class, the insertion, deletion, unlinking, moving and copying of statements or statement lists are all given a few restrictions. The first of these is that the block to be processed must be entirely well-formed with regard to multi-block statements such as DO loops and block-IF statements. This restriction is checked at run-time. (At the same time, the **follow** links, flow graph and other internal structures are automatically updated.) In addition, some further restrictions are placed. For example, deleting a statement which is referenced by another statement outside of the statement block being deleted is flagged at run-time as an error.

Because of these restrictions, it is not possible, for instance, to sequentially insert a DO statement, followed by the statements inside the DO loop, finally followed separately by the ENDDO statement. To get around this there are two options which should provide plenty of flexibility to the programmer. The first is to call one of the several intrinsic methods of **StatementList** to create an empty DO loop (i.e. a header and an ENDDO), and then to singly insert the statements of the body separately in-between these two delimiter statements. The second method is to first create a **List<Statement>** statement list (which has no restrictions whatsoever on the order or type of insertions), and then to insert the entire **List<Statement>** into the **StatementList** at once.

All effort has been made to make the insertion, deletion, unlinking, copying and moving of statements within a **StatementList** robust against errors or dangling pointers.

As an simple example of the use of a **StatementList** object, consider the following short C++ code which iterates through all of the assignment statements in a **StatementList** and prints them (by default with debugging information) to the standard output:

```
StatementList stmt_list;
  ...
for (Iterator<Statement> stmt_iter
       = stmt_list.stmts_of_type(ASSIGNMENT_STMT);
     stmt_iter.valid();
     ++stmt_iter) {
  cout << stmt_iter.current();
}
```

Notice that the `stmt_iter.valid()` expression returns true if and only if the `stmt_iter` iterator is valid, that is, if there are still statements over which to iterate, and the `++stmt_iter` statement causes `stmt_iter` to update its current pointer to the next applicable statement.

Expression Class

Expressions are represented by a tree structure. They are implemented in much the same way as statements, in that an abstract base **Expression** class declares structures common to all expressions and specific expressions are derived from the base. However, most expressions are inherited from three intermediate derived classes: unary expressions (**UnaryExpr** class), binary expressions (**BinaryExpr** class) and non-binary expressions (**NonBinaryExpr** class). These are used to represent expressions with one, two and possibly more than two sub-expressions, respectively. For example, a `.NOT.` expression is represented with a unary node; and subtraction, division and `.EQ.` are represented with a binary node. The non-binary class represents expressions which can have an unlimited number of arguments. This is used mostly for commutative operators, such as addition, multiplication, and several logical operators, and also for lists of expressions, such as a list of actual or formal parameters to a procedure.

Other expression classes are derived which describe specific expression types such as identifier expressions (**IDExpr** class) and integer constant expressions (**IntConstExpr** class). Further, a string expression is derived from the base class which is used to derive classes for real constants and hollerith constants, both of which are represented by strings. Also, a number of expressions are derived from the three basic **UnaryExpr**, **BinaryExpr** and **NonBinaryExpr** classes for the sole purpose of defining methods with more readable names for accessing the sub-expressions. For instance, since the **FunctionCallExpr** class is derived from **BinaryExpr**, it inherits the functions `left()` and `right()` to access its two subexpressions, which are respectively the function symbol being called (represented by an **IDExpr**) and the parameter list. However, instead of requiring the user to abide by this somewhat ambiguous notation, two new methods named `function()` and `parameters()` and added to the **FunctionCallExpr** class to make the accesses to these fields more

clear and self-documenting.

Within the base **Expression** class, fields are declared which specify the expression as well as type information which includes the data type (ex. integer, real, etc..) and the size, making types such as "INTEGER*8" and "INTEGER*4" both possible and distinguishable. In addition, fields are declared which are used for expression simplification.

An example of an expression which derives from the **BinaryExpr** class is the expression to represent array references, **ArrayRefExpr**. The **BinaryExpr** class declares its two fields which are then accessed through **ArrayRefExpr**'s methods `array()` and `subscript()`. One of the benefits of having a binary expression class is that methods which are applicable to all expressions with two sub-expressions can be defined there and will be inherited by all such expressions. Thus, in addition to simply contributing two fields to an array reference, the binary expression also contributes methods which check whether the expression has any side-effects, as well as numerous methods which help in such operations as expression simplification.

All of the safeguards which were implemented within the **Statement** class are also implemented here. This includes the declaration of all methods at the base level which call error routines as well as constructors which fully specify each object.

Symbol Class

The Symbol class hierarchy is set up in a very similar manner to that of the Expression and Statement class hierarchies. The abstract class Symbol defines all possible functions for the derived classes, and the leaves of the Symbol class hierarchy correspond to the different types of symbols possible in a program unit. Six such symbol types are currently defined, represented by the **BlockDataSymbol**, **FunctionSymbol**, **ProgramSymbol**, **SubroutineSymbol**, **SymbolicConstantSymbol** and **VariableSymbol** classes. All such objects may be inserted into the **Symtab** class (see below). As with all classes, almost all of the required fields may be given directly to the constructor. For example, to create a **VariableSymbol** to represent the Fortran variable `XY_ARRAY` defined in the Fortran lines

```
DOUBLE PRECISION XY_ARRAY(0:100, -50:50)
SAVE XY_ARRAY
```

one could use the following C++ code:

```
Symbol *new_symbol = new VariableSymbol(
  ``XY_ARRAY'',
  make_type(DOUBLE_PRECISION_TYPE),
  NOT_FORMAL,
  IS_SAVED,
  new ArrayBounds(new IntConstExpr(0), new IntConstExpr(100)),
  new ArrayBounds(new IntConstExpr(-50),
    new IntConstExpr(50)));
```

Of course, facilities are also available for creating assumed-size arrays.

Symtab Class

The **Symtab** class is our implementation of a symbol table. Its major component is a Dictionary of **Symbol** class objects. It provides methods for, among other things, inserting new symbols (with or without automatic renaming in the case of name conflicts), deleting or unlinking symbols, renaming (and rehashing) symbols, finding symbols by name, printing all the Fortran lines necessary for specifying all the symbols, and creating an iterator to iterate over every symbol in the symbol table.

A Short Program Example

To conclude the discussion, we present the following short programming example of using the Polaris internal data structures.

```
//-----------------------------------------------------
// Insert instrumentation into a program unit:
//
//   At the beginning of the main program, insert:
//     CALL INIT_INTERVALS('program.intvl')
//
//   At the end of the main program, insert:
//     CALL EXIT_INTERVALS('program.sum')
//
//   Around each outer DO loop in the program unit, insert:
//     CALL START_INTERVAL(#)
//   and
//     CALL END_INTERVAL(#)
//   where # is a unique integer for each loop in the program unit
//
//   Assume for simplicity's sake that there are no jumps out of DO loops
//
//-----------------------------------------------------

#include <cvdl.h>

int main(int argc, char *argv[])
{
  // Capture any p_assert() errors here
  P_ASSERT_HANDLER(0);

  // Create a BinStr (representation interface class) and read in a
  //   Fortran program unit file
  BinStr bin;
  bin.read(argv[1]);

  // Create a ProgramUnit class object from the BinStr
  ProgramUnit pgm(``PGM1'', bin);
```

```
// Create and add to the symbol table (keeping a
//   reference around for later use) the necessary symbols:
//
//   EXTERNAL INIT_INTERVALS, EXIT_INTERVALS
//   EXTERNAL START_INTERVAL, END_INTERVAL

Symbol &init_intervals =
  *new SubroutineSymbol(``INIT_INTERVALS'', IS_EXTERNAL,
                        NOT_INTRINSIC, NOT_FORMAL);
pgm.symtab().ins(&init_intervals);

Symbol &exit_intervals =
  *new SubroutineSymbol(``EXIT_INTERVALS'', IS_EXTERNAL,
                        NOT_INTRINSIC, NOT_FORMAL);
pgm.symtab().ins(&exit_intervals);

Symbol &start_interval =
  *new SubroutineSymbol(``START_INTERVAL'', IS_EXTERNAL,
                        NOT_INTRINSIC, NOT_FORMAL);
pgm.symtab().ins(&start_interval);

Symbol &end_interval =
  *new SubroutineSymbol(``END_INTERVAL'', IS_EXTERNAL,
                        NOT_INTRINSIC, NOT_FORMAL);
pgm.symtab().ins(&end_interval);

// ---------------------------------------------
// Make: CALL INIT_INTERVALS( 'program.intvl' )
// ---------------------------------------------

// If this is a main program (not a procedure or BLOCK DATA),
//   add this CALL statement after the entry point.
{
  if (pgm.type() == PROGRAM_PU_TYPE) {
    ListIter<Statement> entries = pgm.stmts().iterate_entry_points();
    // Since this is a PROGRAM_PU_TYPE (main program), there
    //   will be exactly one entry point valid for the iterator.

    Statement *call_init =
      new CallStmt(pgm.new_label(),  // unique tag for the stmt
                   init_intervals,   // subroutine sym being called
                   new CommaExpr(    // actual parameter list
                     new StringConstExpr(``'program.intvl'')));

    // Insert the new statement after the entry point
    pgm.stmts().ins_stmt_after(call_init, entries.current());
  }
```

```
// ---------------------------------
// Make an iterator over all DO stmts
// ---------------------------------

int interval_number = 0;

for (ListIter<Statement> do_stmts = pgm.stmts().stmts_of_type(DO_STMT);
     do_stmts.valid();
     ++do_stmts) {

  if (do_stmts.current().outer() == NULL) { // Is this an outer loop?

    interval_number++;           // Get the next intvl #

    // -------------------------------------------
    // Make: CALL START_INTERVAL( interval_number )
    // -------------------------------------------

    Statement *call_start =
      new CallStmt(pgm.new_label(),  // unique tag for the stmt
                   start_interval,   // subroutine sym being called
                   new CommaExpr(    // actual parameter list
                     new IntConstExpr(interval_number)));

    // Insert the new statement before the current DO statement
    pgm.stmts().ins_stmt_before(call_start, do_stmts.current());

    // Find the matching ENDDO statement
    Statement &end_do = do_stmts.current().follow();

    // -------------------------------------------
    // Make: CALL END_INTERVAL( interval_number )
    // -------------------------------------------

    Statement *call_end =
      new CallStmt(pgm.new_label(),   // unique tag for the stmt
                   end_interval,      // subroutine sym being called
                   new CommaExpr(     // actual parameter list
                     new IntConstExpr(interval_number)));

    // Insert the new statement after the current ENDDO statement
    pgm.stmts().ins_stmt_after(call_end, end_do);
  }
}

// -----------------------------------------
```

```
    // Place the ''CALL EXIT_INTERVALS(...)'' before all exits:
    //
    // ---------------------------------
    // Make an iterator over all exit stmts
    // ---------------------------------

    {
      // Add this call only if this is a main program
      if (pgm.type() == PROGRAM_PU_TYPE) {
        // Iterator over all RETURN and STOP statements
        for (ListIter<Statement> exit_stmts =
             pgm.stmts().stmts_of_type(RETURN_STMT, STOP_STMT);
             exit_stmts.valid();
             ++exit_stmts) {

          Statement *call_exit =
            new CallStmt(pgm.new_label(),  // unique tag for the stmt
                         exit_intervals,   // subroutine sym being called
                         new CommaExpr(    // actual parameter list
                           new StringConstExpr("'program.sum'")));

          // Insert the new statement before the current exit statement
          pgm.stmts().ins_stmt_before(call_exit, exit_stmts.current());
        }
      }

      // Print the resulting program unit to standard output
      //   with debugging information.
      cout << pgm << endl << endl;

      // Print to standard output as Fortran code
      pgm.write(cout);
    }
  }
}
```

Appendix B: Polaris Exchange Format (v0.2)

Introduction

Research into compiler technology has driven the state-of-the-art in restructuring compilers for parallel machines. But, it seems that each research group needs to build the entire compiler framework to support the experimentation that the researcher need to perform. We propose an alternative scenario to this picture. If each compiler could produce an intermediate form that the other compilers could accept, then the scientist performing the experiment could mix-and-match the components from various projects to select the best or most appropriate techniques.

Properties of the Intermediate Form

The intermediate form should have several properties:

1. It should be extensible. It is unlikely that the initial intermediate form chosen will be able to provide all of the information that will ever be required. Thus, being able to extend the datastructure in a straight forward manner is essential.

2. The data structure should be able to support the common-denominator information of the features required by all of the major research projects.

3. It should be independent of the byte-order of the host machine. That is, the particular byte/bit order needs to be specified in the description of the intermediate form.

4. It should be possible to extract the common-denominator information from an intermediate form in the presence of extra information. That is, if extra fields are added during a revision of the specification, programs produced according to the prior revision should still be able to read in the information it understands.

5. It should be as compact as possible with simple rules for a program to read or write the intermediate form. This seems to indicate that a binary representation may be preferred.

Proposed Intermediate Form

The compiler group at CSRD is proposing an intermediate form for general use.

The intermediate form is composed of elements or atoms, ordered collections of elements and unordered collections of elements. A unordered collection of elements is called a set. The ordered collection of elements is called a tuple. We define a map to be a set of 2-tuples. This means that each element of the set is a tuple with two components. The first component is a *field name* (atom) describing the field and the second component is the value of that field.

We represent sets with the '{' and '}' delimiters, and tuples with the '[' and ']' delimiters. Thus a map could be represented as follows:

```
{["field_1", "data_1"], ["field_2", "data_2"] }
```

This form represents two pieces of information. The first field `"field_1"` had the value `"data_1"` and the second field `"field_2"` has the value `"data_2"`.

The data types for the atoms are:

- boolean
- integer
- string

These can be built into higher level structures using:

- set
- tuple

The immediate target language is an extended version of Fortran77. The extensions include block structured loops (DO-ENDDO, DO WHILE), statements for automatic memory allocation or arrays, and vector notation. The goal is to eventually include the HPF specifications and syntax into the intermediate language, but the exact format and not been determined.

We propose that an annotated syntax-tree representation of the program is the appropriate representation level for interchange in the domain of restructuring compilers.

Selecting this level as the common denominator has several advantages when considering restructuring compilers for Fortran. The translators no longer need to deal with the detailed syntactic structures that can be present in Fortran. Most experimental restructuring compilers operate on a source-to-source basis. Thus, they need to be able to recreate an abstract syntax tree in order to generate the Fortran program. Another possible format is the program dependence graph (PDG), however this form may not preserve enough of the original syntactic structure. For instrumentation purposes, you want to perturb the program as little as possible. Thus, the original lexical ordering information present in an abstract syntax tree (statement list) is important.

Details of the Representation

All fields names and most data elements will be strings. A string was chosen as the most flexible implementation mechanism for two reasons. First, semantic information can be encoded into a string (I.e. `"successor"` is much more meaningful than `37`). Second, collisions in the name space are less likely (i.e. local extensions could be prefixed with a unique string, for example `"CSRD_successor"`).

In the binary representation, Section 6.3, the data dictionary will be described. This mechanism allows the use of arbitrarily long self-descriptive extensible field names without suffering from excessive intermediate form size growth.

We will define a program unit to be a map describing a collection of first level fields. The field names are listed in Table 1.

- "symtab": The symbol table of the compilation unit.
- "routine_type": The type of the compilation unit.
- "common_blocks": A list of the COMMON blocks in the unit.
- "equivalences": A representation of the EQUIVALENCE groups.
- "data": An unparsed image of the DATA statements.
- "statements": A list of the statements in the unit.
- "expression": A flattened list of the expressions in the unit.
- "formats": A list of the FORMAT statements in the unit.
- "directives": A list of compiler directives that apply globally.

- *annotations*: Extra information such as "dependences".

Table 1: First level field names

1. "symtab", "routine_type", "common_blocks", "equivalences", "data", "statements", "expression", "formats", "directives" (see Table 1).
2. Statement fields: "outer", "follow" (see Table 5).
3. Statement fields: "successors", "predecessors" (see Table 5).
4. Data Dependences, Control Dependences (Section 6.3).

Figure 2: Required fields in the program unit

Required Information

The design of the intermediate language allows great flexibility in the amount of information that is included about a compilation unit. At a minimum, a program that reads and produces this language must be able to understand and generate the all of the syntactic/lexical information about a program. It is also recommended that some of the derived information, such as the lexically enclosing DO loop ("outer" field, see Table 5), the flow-graph ("successors" and "predecessors" fields, see Table 5), and the compound statement structure linking information ("follow" field, see Table 5) be maintained as well. This information can be regenerated from the syntactic representation of the program but it is of general enough use to be worth maintaining.

In general, we propose a layered requirement for information in the intermediate form. This layering implies that if any derived information at level *N* is present in the intermediate form, then recursively all information at level *N-1* must also be present. Initially all information at level 1 is required. The levels are described in Figure 2.

The "symtab"

The symbol table holds information about the named objects in the compilation unit. The value of the "symtab" field is a map from object names to attributes of that object. For example, if we had the following code fragment in Figure 3, the entries in the symbol table would appear as in Figure 4.

```
SUBROUTINE MATMUL(A,B,C,N)
REAL A, B, C
INTEGER I, J, N, K
DIMENSION A(N, N), B(N, N), C(N, N)
DO I = 1, N, 1
    DO J = 1, N, 1
        A(I,J) = 0.0
        DO K = 1, N, 1
            A(I,J) = A(I,J)+B(I,K)*C(K,J)
        ENDDO
    ENDDO
ENDDO
RETURN
END
```

Figure 3: Example compilation unit

The symbol table "symtab" is indexed by the names of the symbols in the compilation unit. The value of a particular entry is a map containing information about the symbol. The possible field names of the map for each symbol are shown in Table 2.

The "routine_type" of a compilation unit, and the "class" of a symbol table object can have the values shown in Table 3.

The intermediate form supports the same datatypes as the Fortran language. These datatypes are listed in Table 4.

The "dim" field of a "symtab" entry

Each entry in the symbol table with a "dim" field present has a list of upper and lower bounds. This list is a tuple of maps, each entry in the tuple represents the corresponding subscript of the array. The "ub" component of the map is the upper bound, the upper bound must be present if the array is not allocatable. The "lb" field is the lower bound, this field is optional and a lower bound of 1 is assumed if this field is absent.

The "data" table

Currently, DATA statements are not parsed by the system except to pick up variable references. These references are added to the symbol table. The format of the "data" section of the compilation unit is: ["data", [*d1*,...,*dn*]]. Each DATA statement

"class"	The class to which the symbol belongs (Table 3).
"type"	The type of the symbol (Table 4).
"size"	The number of bytes in the symbol (an integer).
"entry"	The statement describing this ENTRY point (a statement tag, see Figure 5).
"dim"	The dimensions of an array (Section 6.3).
"common"	The common block to which the variable belongs (a "common_block" entry).
"data"	The constant integer value of a PARAMETER.
"expr"	An expr. pointer for the value of a PARAMETER. (see Section 6.3). Either "data" or "expr" may be present but not both.
"formal"	Is this a formal parameter? (boolean)
"allocatable"	Is this an allocatable array? (boolean)
"intrinsic"	Is this an intrinsic function? (boolean)
"external"	Is this an external subroutine or function? (boolean)
"saved"	Is this variable listed in a SAVE statement? (boolean)

Table 2: Possible symbol table field names.

"UNKNOWN"	An error condition, normally not used.
"ROUTINE"	A function or a subroutine.
"VARIABLE"	A scalar or array variable.
"SYMBOLIC_CONSTANT"	A variable defined in a PARAMETER statement.
"PROGRAM"	The main program.
"BLOCKDATA"	A BLOCKDATA compilation unit.

Table 3: Possible values for the routine type or class.

"REAL"	The Fortran single precision (REAL*4) datatype.
"DOUBLE PRECISION"	An alias for the Fortran (REAL*8) datatype.
"INTEGER"	The Fortran integer datatype.
"LOGICAL"	The Fortran logical datatype.
"CHARACTER"	The Fortran character datatype.
"COMPLEX"	The Fortran complex datatype.

Table 4: Data types supported by the intermediate form.

```
["symtab",{
 ["MATMUL",{["class","ROUTINE"],["entry","S1"]}],
 ["A",{["class","VARIABLE"],["type","REAL"],["size",4],["formal","T"],
   ["dim",[{["ub",1]}, {["ub",2]}]]}],
 ["B",{["class","VARIABLE"],["type","REAL"],["size",4],["formal","T"],
   ["dim",[{["ub",3]}, {["ub",4]}]]}],
 ["C",{["class","VARIABLE"],["type","REAL"],["size",4],["formal","T"],
   ["dim",[{["ub",5]}, {["ub",6]}]]}],
 ["I",{["class","VARIABLE"],["type","INTEGER"],["size",4]}],
 ["J",{["class","VARIABLE"],["type","INTEGER"],["size",4]}],
 ["K",{["class","VARIABLE"],["type","INTEGER"],["size",4]}],
 ["N",{["class","VARIABLE"],["type","INTEGER"],["size",4],["formal","T"]}]}]
```

Figure 4: Example VDL output for the symbol table of the program in Fig. 3.

(*di*) is represented as an unparsed character string. The strings (*d1*,...,*dn*) are the data statements in the source program.

The "formats" table

Format statements are stored in the "formats" section of the compilation unit. The statement label of the FORMAT statement is converted to a character string, and a "F" is appended to the string. For example, a format statement with the label 999 is converted into the string "F999". The body of the format statement (i.e. all characters inside the outermost parentheses) are also converted into a character string. For example, `999 FORMAT(1X,I2)`, would have a body of "1X,I2". The value of the "formats" section is a map from converted labels like "F999" to the body of the corresponding FORMAT statement. In our example this would generate the entry: `["formats", {["F999", "1X,I2"]}]`.

The "directives" table

The declarative directives are in the "directives" section of the compilation unit. The statement oriented directives are in the "directives" field of each statement. Each directive is stored as a character string with all of the directives lexically listed in a tuple. For example, a statement directives entry could be, `["directives", ["CDIR$ IVDEP", "CDIR$ DOSHARED"]]`.

The "common_blocks" table

The COMMON statements in a compilation unit may define multiple areas of shared storage. Each area is given a unique name. For example, the statement, `COMMON /X/ A, B`, declares "/X/" to be the name of a shared area having `A` and `B` as components. The blank common is indicated by the name "//". The value of the "common_blocks" section is a map from shared area names to a tuple of symbol names present in the common area. In the previous example we would represent is as: `["common_blocks", {["/X/", ["A", "B"]]}]`.

The "equivalences" table

The equivalence table is one of the more complex sections in the compilation unit. Each equivalence class is assigned a unique name, for example `EQ1, EQ2, etc`. By convention an `INTEGER` array is created, with the name EQnBASE, which is the same size as the equivalence class. Each element of the equivalence class is assigned a byte offset from the beginning of the array which spans the equivalence class. The members are stored under the field name `"members"` and the size (in bytes) of the equivalence class is stored under the field name `"size"`. For example given the following statements

```
INTEGER A1(0:10)
EQUIVALENCE (A1(1),B1), (A1(2), B2), (A1(3), B3)
```

as an example we might see: `["equivalences", {["EQ1", {["size", 44], ["members", {["B2", 8], ["B3", 12], ["B1", 4], ["A1", 0]}]}]}]`.

The "statements"

The `"statements"` table is a map of statement tags to statement entries. The statement tag for each entry is a string that is unique for each statement. For example, possible statement tags are: `"S1"`, `"S2"`, `"S3"`, etc. Each statement entry is a map that can have fields indexed by the set of field names listed in Table 5. The `"st"` (statement type) field can contain values from the first column of the list in Figures 8 to 14.

An example of the `"statements"` table is shown in Figure 5. This example is generated from the program in Figure 3.

```
["statements",{
 ["S12",{["st","FLOW_ENTRY"], ["next","S1"]}],
 ["S1",{["st","ENTRY"],["routine","MATMUL"],["parameters",11],
  ["next","S2"]}],
 ["S2",{["st","DO"],["index",12],
  ["init_expr",13],["limit_expr",14],["step_expr",15],
  ["next","S3"],["follow","S9"]}],
 ["S3",{["st","DO"],["index",16],
  ["init_expr",17],["limit_expr",18],["step_expr",19],
  ["next","S4"],["follow","S8"]}],
 ["S4",{["st","ASSIGNMENT"],["lhs",24],["rhs",25], ["next","S5"]}],
 ["S5",{["st","DO"],["index",26],
  ["init_expr",27],["limit_expr",28],["step_expr",29],
  ["next","S6"],["follow","S7"]}],
 ["S6",{["st","ASSIGNMENT"],["lhs",34],["rhs",51], ["next","S7"]}],
 ["S7",{["st","ENDDO"], ["next","S8"],["follow","S5"]}],
 ["S8",{["st","ENDDO"], ["next","S9"],["follow","S3"]}],
 ["S9",{["st","ENDDO"], ["next","S10"],["follow","S2"]}],
 ["S10",{["st","RETURN"]}]}]
```

Figure 5: Example VDL code for the statements table.

"expr"	A pointer to the expression table.
"follow"	A statement tag connecting compound statements.
"index"	The index variable of a DO loop (expr. pointer).
"init_expr"	The lower bound of a DO loop (expr. pointer).
"io_list"	The list of expression/variables in an IO statement.
"label"	The label of a CONTINUE statement.
"label_list"	The list of targets for a COMPUTED GOTO statement.
"lhs"	The left-hand side of an assignment statement.
"limit_expr"	The upper bound of a DO loop (expr. pointer).
"line"	The line number in the source file (integer).
"next"	The lexically next statement (stmt. tag).
"outer"	The lexically enclosing DO loop (stmt. tag).
"parameters"	A parameter list for CALL/ENTRY statements.
"predecessors"	The flow predecessors.
"prev"	The lexically previous statement (stmt tag).
"rhs"	The right-hand side of an assignment statement.
"routine"	The routine name for CALL/ENTRY statements
"s_control"	The list of KEYWORD parameters in an IO statement.
"st"	The statement type. Possible values are in Tables 8 to 14.
"step_expr"	The stride of the DO loop (expr. pointer).
"successors"	The flow successors.
"target"	The label to which a GOTO statement branches.

Table 5: Possible field names for a statement.

"ACCESS"	"ASSOCIATEVARIABLE"	"BLANK"
"BLOCKSIZE"	"BUFFERCOUNT"	"CARRIAGECONTROL"
"DEFAULTFILE"	"DIRECT"	"DISPOSE"
"DISP"	"END"	"ERR"
"EXIST"	"EXTENDSIZE"	"FORM"
"FMT"	"FORMATTED"	"FILE"
"INITIALSIZE"	"IOSTAT"	"NML"
"KEY"	"KEYED"	"KEYEQ"
"KEYGE"	"KEYGT"	"KEYID"
"MAXREC"	"NAME"	"NAMED"
"NEXTREC"	"NUMBER"	"OPENED"
"ORGANIZATION"	"REC"	"RECL"
"RECORDSIZE"	"RECORDTYPE"	"SEQUENTIAL"
"STATUS"	"TYPE"	"UNFORMATTED"
"UNIT"	"USEROPEN"	

Table 6: Possible fields in an s_control list

Derived fields that apply to all statements.

- ["next", statement tag of lexically next statement]
- ["prev", statement tag of lexically previous statement]
- ["successors", a set of statement tags of control-flow successors]
- ["predecessors", a set of statement tags from which control flows]
- ["outer", the stmt. tag of the enclosing DO loop] (possibly omitted)
- ["in_refs", a set of expr. pointers of read references to "ID" or "ARRAY_REF"]
- ["out_refs", a set of expr. pointers of write references to "ID" or "ARRAY_REF"]
- ["act_refs", a set of expr. pointers of unknown references to "ID" or "ARRAY_REF"]

Table 7: Derived information about all statement types

The "expression" table

All expressions in the program are represented via nodes in the expression table. The expression table is a tuple which is indexed by an integer representing an expression pointer. Each node can have some of the fields from the list of field names in Table 15. The "op" field of the node can take on values listed in Tables 16 to 23. An example of a complete expression table, for the program in Figure 3, is shown in Figures 6 and 7.

The expression table representation is advantageous as it opens the possibility to express DAGs in the expression structure. Also, it allows the possibility of expression of sub-expression sharing at different points in the intermediate form. A alternative that may be more readable but does not have these two features is the expression tree form. The same information is present in this form. The difference is that instead of having expression pointers as the values of the "args" field the actual sub-expressions are explicitly nested as the values.

As an example of an expression we can consider a "," (comma) expression. The comma expression is used to create a list of expressions for use in places like array references, parameter lists. The operation code for this expression is ["op", ","]. The type of the expression is ["type", "VOID"]. The field which has the most significance is the arguments field, ["args", [*expr1*, *expr2*]]. In this example, the tokens *expr1* and *expr2* would be numbers which represent the index in the expression table of the expressions concatenated by the comma expression.

Other examples of complicated expressions would be the "LAMBDA_CALL" expression, and "ARRAY_CONSTANT" expression. The "LAMBDA_CALL" expression is used to indicate a statement function invocation. The expression takes two arguments. The first is the expression, from the statement function, to be substituted. This is an expression involving "ARG#" expressions which reference the formal parameters and "ID" expressions to reference variables in the enclosing compilation unit. The second expression in the args list is a comma expression. The arguments of the comma expressions are the actual parameters to this invocation of the statement function.

"ALLOCATE"	dynamic memory allocation • ["st", "ALLOCATE"] • ["parameters", a comma expr. pointer]
"ARITHMETIC_IF"	IF (X) 100,200,300 • ["st", "ARITHMETIC_IF"] • ["expr", boolean expr. pointer] • ["label_list", [list of statement tags]]
"ASSIGN"	ASSIGN 100 to G • ["st", "ASSIGN"] • ["lhs", id expr. pointer] • ["target", statement tag]
"ASSIGNED_GOTO"	GOTO G • ["st", "ASSIGNED_GOTO"] • ["expr", id expr. pointer] • ["label_list", [list of statement tags]]
"ASSIGNMENT"	A = B • ["st", "ASSIGNMENT"] • ["lhs", expr. pointer] • ["rhs", expr. pointer]

Table 8: Possible values for the type of a statement (Part 1).

"BACKSPACE"	I/O statement, rewind one record • ["st", "BACKSPACE"] • ["s_control", {map of ["keyword",expr] pairs}]
"CALL"	CALL SAXPY(A,X,Y) • ["st", "CALL"] • ["routine", name of routine to call] • ["parameters", a comma expr. pointer]
"CLOSE"	I/O statement, close a file • ["st", "CLOSE"] • ["s_control", {map of ["keyword",expr] pairs}]
"COMPUTED_GOTO"	GOTO (100,200,300) C • ["st", "COMPUTED_GOTO"] • ["expr", an expr. pointer] • ["label_list", [list of statement tags]]
"DEALLOCATE"	dynamic memory deallocation. • ["st", "DEALLOCATE"] • ["parameters", a comma expr. pointer]

Table 9: Possible values for the type of a statement (Part 2).

"DO"	DO I = 1, 100, 2 • ["st", "DO"] • ["index", an id expr. pointer] • ["init_expr", an expr. pointer] • ["limit_expr", an expr. pointer] • ["step_expr", an expr. pointer] • ["follow", the statement tag of the matching ENDDO]
"ELSE"	ELSE • ["st", "ELSE"] • ["follow", the statement tag of the matching ENDIF]
"ELSEIF"	ENDIF • ["st", "ELSEIF"] • ["expr", a boolean expr. pointer] • ["follow", the statement tag of the next ELSEIF/ELSE/ENDIF]
"ELSEWHERE"	ELSEWHERE • ["st", "ELSEIF"] • ["follow", the statement tag of the matching ENDWHERE]

Table 10: Possible values for the type of a statement (Part 3).

"ENDDO"	ENDDO • ["st", "ENDDO"] • ["follow", the statement tag of the matching DO]
"ENDFILE"	I/O statement (specify end of file) • ["st", "ENDFILE"] • ["s_control", {map of ["keyword",expr] pairs}]
"ENDIF"	ENDIF • ["st", "ENDIF"]
"ENDWHERE"	ENDWHERE • ["st", "ENDWHERE"]
"ENTRY"	ENTRY statement and compilation unit • ["st", "ENTRY"] • ["routine", name of the entry point] • ["parameters", a comma expr. pointer] (possibly omitted)
"EXIT"	EXIT, terminate a DO loop • ["st", "EXIT"]

Table 11: Possible values for the type of a statement (Part 4).

"FLOW_ENTRY"	The first node in the flow graph • ["st", "FLOW_ENTRY"]
"IF"	block IF • ["st", "IF"] • ["expr", a boolean expr. pointer] • ["follow", the statement tag of the next ELSEIF/ELSE/ENDIF]
"IMPLIED_GOTO"	structured change of control flow • ["st", "IMPLIED_GOTO"] • ["follow", the statement tag to which control flows]
"INQUIRE"	I/O statement, check file info • ["st", "INQUIRE"] • ["s_control", {map of ["keyword",expr] pairs}]
"LABEL"	CONTINUE statement • ["st", "LABEL"] • ["label", integer value of the statement label]
"OPEN"	I/O statement, open a file • ["st", "OPEN"] • ["s_control", {map of ["keyword",expr] pairs}]

Table 12: Possible values for the type of a statement (Part 5).

"PAUSE"	I/O statement, suspend the program • ["st", "PAUSE"] • ["expr", an expr. pointer]
"PRINT"	I/O statement, write output • ["st", "PRINT"] • ["s_control", {map of ["keyword",expr] pairs}] • ["io_list", a comma expr. pointer]
"READ"	I/O statement, read input • ["st", "READ"] • ["s_control", {map of ["keyword",expr] pairs}] • ["io_list", a comma expr. pointer]
"RETURN"	RETURN, return from a subroutine • ["st", "RETURN"] • ["expr", an expr. pointer] (possibly omitted)
"REWIND"	I/O statement, rewind a file • ["st", "REWIND"] • ["s_control", {map of ["keyword",expr] pairs}]

Table 13: Possible values for the type of a statement (Part 6).

"STOP"	Terminate the program • ["st", "STOP"] • ["expr", an expr. pointer] (possibly omitted)
"UNCONDITIONAL_GOTO"	GOTO 100 • ["st", "UNCONDITIONAL_GOTO"] • ["target", a statement tag]
"WHERE"	WHERE (A(1:N) .GT. 0) • ["st", "WHERE"] • ["expr", an expr. pointer] • ["follow", the next ELSEWHERE/ENDWHERE statement tag]
"WHILE"	DO WHILE (.TRUE.) • ["st", "WHILE"] • ["expr", an expr. pointer] • ["follow", the matching ENDDO statement tag]
"WRITE"	I/O statement, write output • ["st", "PRINT"] • ["s_control", {map of ["keyword",expr] pairs}] • ["io_list", a comma expr. pointer]

Table 14: Possible values for the type of a statement (Part 7).

The original name of the statement function is not kept in this representation. It is expected that either the statement function call will be inlined, or a new temporary statement function will be created on output.

An "ARRAY_CONSTANT" is an extension to Fortran77 adopted from Fortran90. This expression creates a constant array. The argument to this expression is a comma expression which lists the elements of the array. The elements of the array can be constant expressions, such as "INTEGER_CONSTANT" or "REAL_CONSTANT" or in implied-do loop. An implied-do loop has as the root of the expression a node of type ["op", "DO"]. The two arguments to this expression are: a comma node containing the expressions which will be part of the "ARRAY_CONSTANT", and an "=" node specifying the variable to iterate over and the range of the iteration. The "=" node has two arguments as well. The first argument is the "ID" node specifying the name of the dummy variable. The second argument is a comma node containing three entries. The three entries are the initial value, the limit value, and the stride of the iteration.

"op"	The opcode of the expression node. Possible values are listed in Tables 16 to 23.
"type"	The datatype of the subexpression.
"size"	The size of the datatype.
"name"	The symbol name of an "ID" node.
"args"	A tuple of arguments.
"data"	The value of a constant.

Table 15: Possible field names for an expression node.

The ''routine_type''

The type of the compilation unit is stored in this field. Possible values are from the list in Table 3.

The ''initial_statement''

The initial statement is stored here to allow for the use of a linked list representation of the statements rather than a linear list. Figure 9 shows an example of this field. An alternative format of the "statements" table would be to represent it as a tuple instead of a map. The relative position of the statements would then implicitly define the "next", "prev", and "initial_statement"

"INTEGER_CONSTANT"	integer constant • ["op", "INTEGER_CONSTANT"] • ["data", *integer value*] • ["type", "INTEGER"]
"REAL_CONSTANT"	floating point constant • ["op", "REAL_CONSTANT"] • ["data", *"floating point value"*] • ["type", "REAL" *or* "DOUBLE PRECISION"]
"STRING_CONSTANT"	character string constant • ["op", "STRING_CONSTANT"] • ["size", *length of string*] • ["data", *"'string'"*] • ["type", "CHARACTER"]
"LOGICAL_CONSTANT"	.TRUE. or .FALSE. • ["op", "LOGICAL_CONSTANT"] • ["data", ".TRUE." *or* ".FALSE."] • ["type", "LOGICAL"]
"HOLLERITH_CONSTANT"	hollerith constant 2HAB • ["op", "HOLLERITH_CONSTANT"] • ["data", "2HXY"] • ["type", "UNKNOWN"]

Table 16: Possible values for the opcode of an expression node (Part 1).

"ARRAY_CONSTANT"	(\ (I, I = 1, 10) \) • ["op", "ARRAY_CONSTANT"] • ["args", [*comma expr. pointer*]] • ["type", "UNKNOWN"]
"ARRAY_REF"	A(I), array reference • ["op", "ARRAY_REF"] • ["args", [*id expr. pointer*, *comma expr. pointer*]] • ["type", *type of the array*]
"SUBSTRING"	STR(1:10), character substring • ["op", "SUBSTRING"] • ["args", [*expr. pointer*, *colon expr. pointer*]] • ["type", "CHARACTER"]
"FUNCTION_CALL"	FUNC(A,B,C), external function • ["op", "FUNCTION_CALL"] • ["args", [*id expr. pointer*, *comma expr. pointer*]] • ["type", *return type of the function*]
"INTRINSIC_CALL"	IFUNC(A,B,C), intrinsic function • ["op", "INTRINSIC_CALL"] • ["args", [*id expr. pointer*, *comma expr. pointer*]] • ["type", *return type of the function*]

Table 17: Possible values for the opcode of an expression node (Part 2).

"LAMBDA_CALL"	Statement function reference • ["op", "LAMBDA_CALL"] • ["args", [*id expr. pointer*, *comma expr. pointer*]] • ["type", *return type of the statement function*]
"ARG#"	Dummy argument to statement function • ["op", "ARG#"] • ["data", *position in the argument list*] • ["type", *type of the formal argument*]
"COMPLEX"	(0.0,1.0), complex constant • ["op", "COMPLEX"] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "COMPLEX"]
"+"	A+B, add two expressions • ["op", "+"] • ["args", [*expr. pointer*, ..., *expr. pointer*]] • ["type", *type of the result*]
"-"	A-B, subtract two expressions • ["op", "-"] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", *type of the result*]

Table 18: Possible values for the opcode of an expression node (Part 3).

`"*"`	A*B, multiply two expressions • ["op", "*"] • ["args", [*expr. pointer*, ..., *expr. pointer*]] • ["type", *type of the result*]
`"/"`	A/B, divide two expressions • ["op", "/"] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", *type of the result*]
`"ID"`	Symbol table reference • ["op", "ID"] • ["name", "*symtab entry*"] • ["type", *type of the symtab entry*]
`"()"`	Pass by value CALL F(A, (B), C) • ["op", "()"] • ["args", [*expr. pointer*]] • ["type", *type of the argument*]
`"U+"`	+A, unary plus • ["op", "U+"] • ["args", [*expr. pointer*]] • ["type", *type of the argument*]

Table 19: Possible values for the opcode of an expression node (Part 4).

`"U-"`	-A, unary minus • ["op", "U-"] • ["args", [*expr. pointer*]] • ["type", *type of the argument*]
`".EQ."`	A .EQ. B, A == B • ["op", ".EQ."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
`".NE."`	A .NE. B, A != B • ["op", ".NE."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
`".LT."`	A .LT. B, A < B • ["op", ".LT."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
`".LE."`	A .LE. B, A <= B • ["op", ".LE."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]

Table 20: Possible values for the opcode of an expression node (Part 5).

".GT."	A .GT. B, A > B • ["op", ".GT."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
".GE."	A .GE. B, A >= B • ["op", ".GE."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
".AND."	LA .AND. LB, LA && LB • ["op", ".AND."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
".OR."	LA .OR. LB, LA || LB • ["op", ".OR."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
".NOT."	.NOT. LA, ! LA • ["op", ".NOT."] • ["args", [*expr. pointer*]] • ["type", "LOGICAL"]

Table 21: Possible values for the opcode of an expression node (Part 6).

".EQV."	LA .EQV. LB, LA == LB • ["op", ".EQV."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
".NEQV."	LA .NEQV. LB, LA != LB • ["op", ".NEQV."] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "LOGICAL"]
"**"	A**B, pow(A,B) • ["op", "**"] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", *type of the result*]
"//"	CA // CB, concatenate CA, CB • ["op", "//"] • ["args", [*expr. pointer*, *expr. pointer*]] • ["type", "CHARACTER"]
"IO*"	WRITE(*,*), console or list I/O • ["op", "IO*"]
"RETURN*"	SUBROUTINE FOO(X,Y,*,Z) place holder • ["op", "RETURN*"]

Table 22: Possible values for the opcode of an expression node (Part 7).

```
    ["expression",[
1    {["op","ID"],["name","N"],["type","INTEGER"]},
2    {["op","ID"],["name","N"],["type","INTEGER"]},
3    {["op","ID"],["name","N"],["type","INTEGER"]},
4    {["op","ID"],["name","N"],["type","INTEGER"]},
5    {["op","ID"],["name","N"],["type","INTEGER"]},
6    {["op","ID"],["name","N"],["type","INTEGER"]},
6    {["op","ID"],["name","A"],["type","REAL"]},
7    {["op","ID"],["name","B"],["type","REAL"]},
9    {["op","ID"],["name","C"],["type","REAL"]},
10   {["op","ID"],["name","N"],["type","INTEGER"]},
11   {["op",","],["args",[7,8,9,10]]},
12   {["op","ID"],["name","I"],["type","INTEGER"]},
13   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
14   {["op","ID"],["name","N"],["type","INTEGER"]},
15   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
16   {["op","ID"],["name","J"],["type","INTEGER"]},
17   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
18   {["op","ID"],["name","N"],["type","INTEGER"]},
19   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
20   {["op","ID"],["name","A"],["type","REAL"]},
21   {["op","ID"],["name","I"],["type","INTEGER"]},
22   {["op","ID"],["name","J"],["type","INTEGER"]},
23   {["op",","],["args",[21,22]]},
24   {["op","ARRAY_REF"],["type","REAL"],["args",[20,23]]},
25   {["op","REAL_CONSTANT"],["data","0.0"],["type","REAL"]},
26   {["op","ID"],["name","K"],["type","INTEGER"]},
27   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
28   {["op","ID"],["name","N"],["type","INTEGER"]},
29   {["op","INTEGER_CONSTANT"],["data",1],["type","INTEGER"]},
30   {["op","ID"],["name","A"],["type","REAL"]},
```

Figure 6: Example VDL code for the expression table (Part 1).

"DO"	(I, I = 1, 10), top level op • ["op", "DO"] • ["args", [*comma expr. pointer*, = *expr. pointer*]]
"="	implied DO assignment • ["op", "="] • ["args", [*id expr. pointer*, *comma expr. pointer*]]
","	comma operator • ["op", ","] • ["args", [*expr. pointer*,...,*expr. pointer*]]

Table 23: Possible values for the opcode of an expression node (Part 8).

```
31   {["op","ID"],["name","I"],["type","INTEGER"]},
32   {["op","ID"],["name","J"],["type","INTEGER"]},
33   {["op",","],["args",[31,32]]},
34   {["op","ARRAY_REF"],["type","REAL"],["args",[30,33]]},
35   {["op","ID"],["name","A"],["type","REAL"]},
36   {["op","ID"],["name","I"],["type","INTEGER"]},
37   {["op","ID"],["name","J"],["type","INTEGER"]},
38   {["op",","],["args",[36,37]]},
39   {["op","ARRAY_REF"],["type","REAL"],["args",[35,38]]},
40   {["op","ID"],["name","B"],["type","REAL"]},
41   {["op","ID"],["name","I"],["type","INTEGER"]},
42   {["op","ID"],["name","K"],["type","INTEGER"]},
43   {["op",","],["args",[41,42]]},
44   {["op","ARRAY_REF"],["type","REAL"],["args",[40,43]]},
45   {["op","ID"],["name","C"],["type","REAL"]},
46   {["op","ID"],["name","K"],["type","INTEGER"]},
47   {["op","ID"],["name","J"],["type","INTEGER"]},
48   {["op",","],["args",[46,47]]},
49   {["op","ARRAY_REF"],["type","REAL"],["args",[45,48]]},
50   {["op","*"],["type","REAL"],["args",[44,49]]},
51   {["op","+"],["type","REAL"],["args",[39,50]]}]]
```

Figure 7: Example VDL code for the expression table (Part 2).

```
["routine_type","SUBROUTINE"]
```

Figure 8: Routine type for Code in Figure 3.

```
["initial_statement","S12"]
```

Figure 9: Initial statement for Code in Figure 3.

Dependences

Information that is derived from the compilation unit can either be recomputed when needed or stored along with the compilation unit. Dependences are an example of a type of derived information that is expensive to compute and thus it is natural to want to remember this information. Dependences also have another characteristic. They are uniform in structure and potentially large in number. These two characteristics imply that the full generality of the self-descriptive nature of the intermediate language is inappropriate for dependences. We compromise by allowing some flexibility in the content of some fields but have the layout of the dependence be fixed by the key at the beginning of each dependence record.

Data and Control dependences will be stored as a major component of the program unit. The keyword `"dependences"` will be used. The value of this field is a map from statement tags to sets of dependence arcs. For example: `["dependences", {["S1", {`*d1*`,`*d2*`,`*d3*`}]}]`. In this example we assume a loop is at `"S1"` with three dependences, *d1*, *d2*, and *d3*. The format of the dependences is conditional on the type of the dependence and the specify of the dependence.

Data Dependences come in three common types, (`"f"`, `"o"`, `"a"`). These correspond the *may* data dependences for flow, output and anti. An alternate set of symbols such as (`"F"`, `"O"`, `"A"`) could be used to indicate *must* dependences. The remaining possible fields for these dependences are listed below.

1. `["f", "S1", "S2"]`
2. `["f", "S1", "S2", "X"]`
3. `["f", "S1", "S2", "X",` *e1*`,` *e2*`]`
4. `["f", "S1", "S2", "X",` *e1*`,` *e2*`, ["<", "<=", ">"]]`
5. `["f", "S1", "S2", "X",` *e1*`,` *e2*`, [-1, -1, 0]]`

As you can see from this example, the length of a dependence may be variable to only include the information that is currently needed. The first example indicates a flow dependence from statement "S1" to "S2". This form can be used to summarize statement level dependences. If slightly more information is needed then the second form can be used. This form includes the name of the variable involved in the dependence. The third form also includes two expression pointers *e1*, and *e2* to indicate the actual expressions involved in the dependence. The fourth form adds a direction vector as the last component of the dependence. Finally, dependence distances can be substituted into the places occupied by the directions. The form of the dependence can be determined by the length of the dependence tuple and the datatype of the components.

This form provides flexibility while still maintaining a reasonably compact format. The storage requirements for a dependence range from 8 to 55 bytes depending on the number of items in the dictionary for the compilation unit and the amount of information included in the dependence.

Binary Representation of the Components

The previous section described the logical structure of the interchange format. However, this format is more verbose than it needs to be, and it more difficult to read that it needs to be.

The binary representation uses a tagged format. Each element is preceded by a tag that describes the datatype of the element and optionally a field which describes the length of the element.

The tags are called form_code_tags which are allocates one byte in the binary representation and are listed below:

- ft_omega: An undefined value.
- ft_true: The boolean value TRUE.
- ft_false: The boolean value FALSE.
- ft_integer: An integer value. The value of the integer is stored in the field following the form_code.
- ft_real: A real value. The value of the real is stored in the field following the form_code in 64bit IEEE representation.
- ft_string: The string form_code is followed by a field storing the length of the string in bytes. The characters in the string are stored immediately after the length field.
- ft_tuple: A tuple is represented as a field containing the number of elements in the tuple followed by the binary representations of each element in the tuple.
- ft_set: The same representation as a tuple.
- ft_map: The same representation as a tuple.
- ft_dict: The field following the form_code is an integer which indicates a position in the data dictionary.
- ft_header: A marker to specify the length in bytes of the data dictionary. The header also includes version information. Mainly useful for specifying I/O buffer sizes.
- ft_body: A marker to specify the length in bytes of the compilation unit. Mainly useful for specifying I/O buffer sizes.

The default variants of these form_codes assume that the fields following the form_code are all four bytes in size. We also define variants of these form_codes that assume the fields are two bytes and that assume the fields are one byte. These latter variants are semantically the same as the default form_codes, except that they require less space to store the information in the intermediate form.

A unique feature of the binary representation of the intermediate form is the use of a data dictionary. The data dictionary stores elements that are used repeatedly in the intermediate form. For example, statement labels, field names, variables, etc. The ft_dict form code is a reference to an entry in the data dictionary. Since the ft_dict element can be much shorter than a ft_string element, the replacement of the ft_string by an ft_dict reduces the size of the intermediate form without losing any of the self describing properties or the extensibility properties.

Field Name	code	layout
ft_omega	'A'	'A'
ft_true	'B'	'B'
ft_false	'C'	'C'
ft_real	'D'	'D', 8 byte IEEE double
ft_integer	'E'	'E', 1 byte signed integer
	'F'	'F', 2 byte signed integer
	'G'	'G', 4 byte signed integer
ft_string	'H'	'H', 1 byte unsigned length, up to $2^8 - 1$ characters
	'I'	'I', 2 byte unsigned length, up to $2^{16} - 1$ characters
	'J'	'J', 4 byte unsigned length, up to $2^{32} - 1$ characters
ft_tuple	'K'	'K', 1 byte unsigned length, up to $2^8 - 1$ elements
	'L'	'L', 2 byte unsigned length, up to $2^{16} - 1$ elements
	'M'	'M', 4 byte unsigned length, up to $2^{32} - 1$ elements
ft_set	'N'	'N', 1 byte unsigned length, up to $2^8 - 1$ elements
	'O'	'O', 2 byte unsigned length, up to $2^{16} - 1$ elements
	'P'	'P', 4 byte unsigned length, up to $2^{32} - 1$ elements
ft_map	'Q'	'Q', 1 byte unsigned length, up to $2^8 - 1$ 2-tuples
	'R'	'R', 2 byte unsigned length, up to $2^{16} - 1$ 2-tuples
	'S'	'S', 4 byte unsigned length, up to $2^{32} - 1$ 2-tuples
ft_dict	'T'	'T', 1 byte unsigned dictionary index
	'U'	U', 2 byte unsigned dictionary index
	'V'	V', 4 byte unsigned dictionary index
ft_header	'#'	'#', 4 byte version number, 4 byte unsigned length
ft_body	'$'	'$', 4 byte version number, 4 byte unsigned length

Table 24: Storage layout of the binary representation

The dictionary is a tuple of strings. The position of a string in the tuple is the index for the ft_dict element.

Appendix C: Automatic Array Privatization

Introduction

Enhancing parallelism, balancing loads and reducing communication are among the major tasks of today's parallelizing compilers. Memory-related dependences can severely limit the potential parallelism of a program. Privatization is a technique that allows each concurrent thread to allocate a variable in its private storage such that each thread accesses a distinct instance of the variable. By providing a distinct instance of a variable to each processor, privatization can eliminate memory related dependences. Previous studies on the effectiveness of automatic program parallelization show that *privatization* is one of the most effective transformations for the exploitation of parallelism [EHLP91]. A related technique called *expansion* [PW86] transforms each reference to a particular scalar into a reference to a vector element in such a way that each thread accesses a different vector element. When applied to an array, expansion creates a new dimension for the array.

Because the access to a private variable is inherently local, privatization reduces the communication and facilitates data distribution. Since private instances of a variable are spread among all the active processors, privatization provides opportunities to spread computation among the processors and improve load balancing [TP92].

We present an algorithm for automatically generating an annotated parallel program from a sequential program represented by a control flow graph. For each loop in the target program, privatizable arrays are annotated; last-value assignment statements for live privatizable arrays after the loop are also annotated. The algorithm has been implemented in the POLARIS parallelizing compiler.

Previous work on eliminating memory-related dependences focused on scalar expansion [Wol82], scalar privatization [BCFH89], scalar renaming [CF87], and array expansion [PW86] [Fea88]. Recently there have been several papers on array privatization [Li92][MAL92][TP92].

Our work on automatic array privatization presents the following new results:

- We use data flow-based analysis for array reference. Compared with the dependence analysis-based approach [MAL92], which has to employ parametric integer programming in its most general case, our approach is more efficient and can handle nonlinear subscripts that cannot be handled by integer programming.

- We distinguish private arrays whose last value assignment can be determined statically from those whose last values have to be assigned dynamically at runtime. This work can potentially identify more private arrays than other algorithms can identify. Our algorithm for identifying static last value assignments can handle complicated subscripts such as coupled subscript; hence the need for run-time synchronization is relatively small.

- The algorithm proceeds from the bottom up, which allows us to easily extend the algorithm to program call trees for interprocedural analysis. Our experience shows this interprocedural array reference analysis is necessary in many cases for successful array privatization in real applications.

- To evaluate its effectiveness, we test the algorithm on the programs in the Perfect Benchmarks. We compare the automatic privatization with manual privatization described in a previous study[EHLP91] and find that we need more sophisticated techniques to determine the relationship between symbolic values for further improvement.

- To facilitate further improvement, we present a goal-directed technique to determine symbolic values in the present of conditional statements, loops, and index arrays.

The rest of the paper is organized as follows. Section 2 is an overview of the issues in automatic array privatization and gives an example that motivates this work. Section 3 presents the algorithm. The algorithm can be divided into two parts: private array identification and last-value assignment resolution. Section 4 contains the experiments of automatic privatization of the Perfect Benchmarks and presents a comparison of automatic privatization with manual privatization. Section 5 presents a goal-directed technique that uses the SSA form of a program to determine symbolic values in the presence of conditional statements, loops, and index arrays. Section 6 presents the conslusion.

Background

Data dependence [Ban88] specifies the precedence constraints in the execution of statements in a program due to data producer and consumer relationships. *Anti-dependences* and *output dependences* are memory-related or *false* dependences because they are not caused by the flow of values from one statement to another, but by the reuse of memory locations. Consider the loop:

```
S1:  DO I = 1, N
S2:     A(1) = X(I,J)
S3:     DO J = 2, N
S4:         A(J) = A(J-1)+Y(J)
S5:     ENDDO
S6:     DO K = 1, N
S7:         B(I,K) = B(I,K) + A(K)
S8:     ENDDO
S9:  ENDDO
```

Because every iteration of loop *S1* accesses the same elements of array A, loop *S1* cannot be executed in parallel. However, there is no flow of value across iterations. The conflict can be resolved by declaring A to be private to each iteration of loop *S1*. We add the followin directives to the loop:

```
C$DIR  INDEPENDENT
C$DIR  PRIVATE A(1:N)
C$DIR  LAST VALUE A(1:N) WHEN (I.EQ.N)
```

The *INDEPENDENT* directive is borrowed from HPF [For93]. It specifies that the iterations of loop $S1$ are independent. There are two directives for a private array. The *PRIVATE* directive associates the privatizable arrays with each iteration of a

loop. The *LAST VALUE* statement specifies the conditions when a processor should copy its private array value to the global array. The interpretation of the directives is as follows:

- Each processor cooperating in the execution of the loop allocates the private arrays in its local storage before executing any statement in the loop.
- During the entire execution of an iteration, all the references to a private array are directed to the instance local to the processor.
- After the execution of an iteration has completed, the processor checks the last-value assignment condition. If the condition is satisfied, the processor copies the private array to the corresponding global array. This operation is called *copy-out*.

The results of our research on manual array privatization of Perfect Benchmarks didn't provide any case where a privatizable array needs both a local value and a global value. Hence our model does not have *copy-in*, that is, we do not allow values to be copied from the global array to the private array. Under this assumption, our definition of privatizable array is as follows.

Definition 1 Let A be an array that is referenced in a loop L. We say A is privatizable to L if the following conditions are satisfied.

1. Every fetch to an element of A in L must be preceded by a store to the element in the same iteration of L.
2. Different iterations of L may access the same location of A. □

The conditions for copying out the value of a private array to a global array are also determined by the compiler. In simple cases such as the one above, the algorithm can find a closed form for the condition. We call these cases *static last-value assignment*. In the more complicated cases, such as in the following loop, the last-value assignment has to be determined at run time:

```
C$DIR  INDEPENDENT
C$DIR  PRIVATE A(1:N)
C$DIR  LAST VALUE A(1) WHEN (I.EQ.N)
C$DIR  LAST VALUE A(2:N) WHEN DYNAMIC
S1:  DO I = 1, N
S2:     A(1) = X(I,J)
S3:     IF (A(1).GT.0) THEN
S4:        DO J = 2, N
S5:           A(J) = A(J-1)+Y(J)
S6:        ENDDO
S7:        DO K = 1, N
S8:           B(I,K) = B(I,K) + A(K)
S9:        ENDDO
S10:    ENDIF
S11:  ENDDO
```

In this example, the array section `A(2:N)` is conditionally assigned. `A` is still privatizable because it satisfies the privatizability conditions, but its last-value assignment cannot be determined at compile time. We use the key word *DYNAMIC* to specify that run-time resolution techniques such as *synchronization variable* [ZY87] will have to be used for the array section `A(2:N)`. These cases are termed *dynamic last-value assignment*. For instance, the compiler can associate the subarray `A(2:N)` with a synchronization variable *last-iteration*, which stores the last iteration that was written to `A(2:N)`. Every iteration that defines `A(2:N)` will atomically compare its iteration number with the last iteration. If its iteration number is larger than the last iteration, the processor stores its iteration number into the last-iteration variable and copy-out `A(2:N)`. Otherwise, the assignment is ignored, because a later iteration has already written to `A(2:N)`. Note that because all the iterations are independent, the number of copy-out operations can be reduced by scheduling the loop backward from the last iteration to the first iteration.

Algorithm

We consider the problem of identifying privatizable arrays in a data flow framework. From this point of view, to determine if an array is privatizable in a loop, we need to determine if all the definitions that are used come from the same iteration of the loop.

Data Flow Framework

Problem Formulation

Data flow analysis examines the flow of values through a program and solves data flow problems by propagating information along the paths of a control flow graph. Because private arrays are associated with *DO* loops in the program, we must extend the traditional program flow graph with information about the scope of do loops.

Definition 2 Let $G = (N, E, s)$ be a program flow graph where N are nodes, E are arcs, and $s \in N$ is the initial node. Let L be a subflowgraph corresponding to a do loop (including all loops nested within it). We define as $control(L) \subset L$ the subset of nodes in L corresponding to the loop entry, increment and test of the loop index. $control(L)$ identifies the loop index, its limits and its step. We define the $body(L)$ as $L - control(L)$. □

Note that when a program has nested loops, the control and body of the inner loop are included in the loop body of the outer loop. When the control flow of an inner loop is not important for the analysis of an outer loop, we can use abstraction for the inner loop and simplify the flowgraph by collapsing the subflowgraph corresponding to the inner loop into one node.

Definition 3 Let $G = (N, E, s)$ be a flow graph and L be a do loop in G. The $COLLAP(G, L)$ is a flow graph with the subflowgraph L collapsed into one node. □

Given a subflowgraph L corresponding to a loop, we want to decide if for every

iteration of the loop, all reaching definitions to an array use come from the same iteration. We can do this through *def-use* analysis. The data values to be analyzed include both scalar values and array values. They are *scalar variable*, *subscripted variable*, and *subarray*. A subscripted variable consists of an array identifier and a list of subscript expressions. It is a special case of scalar variables. A subarray consists of a subscripted variable and one or more *ranges* for some of the indices in the subscript expression. A range includes expressions for the lower bound, upper bound, and stride. The notion of subarray we use in this paper is an extension at the *regular section* used by others[CK88a]. Using subarray, we can represent the triangular region and banded region, as well as the strip, grid, column, row, and block of a array. For instance, the following examples respectively represent a dense upper triangle, grids in the upper triangle, and diagonal of array A.

```
(A(I,I:N),[I=1:N])
(A(I,I:N:2),[I=1:N:2])
(A(I,I),[I=1:N])
```

A range in a subarray is interpreted as a $FORALL$, for instance, `(A(I,I:N),[I=1:N])` is interpreted as `FORALL (I=1:N) A(I,I:N)`. Here the $FORALL$ is just an assertion, no operational constraint, such as the order of assignment to different elements, is imposed.

We now describe the algorithm to do def-use analysis involving arrays. We start by computing *outward exposed* definitions and uses for each basic block S in the loop body. A definition of variable v in a basic block S is said to be outward exposed if it is the last definition of v in S. A use of v is outward exposed if S does not contain a definition of v before this use [ZC91].

Definition 4 Let S be a basic block and VAR be the set of scalar variables, subscripted variables, and subarrays in the program. Henceforth these are called variables.

1. $DEF(S) := \{v \in VAR : v$ has an outward exposed definition in $S\ \}$
2. $USE(S) := \{v \in VAR : v$ has an outward exposed use in $S\ \}$
3. $KILL(S) := \{v \in VAR : v$ has a definition in $S\ \}$ □

Given a subflowgraph of a loop, we want to determine if for every use, there are definitions in the same iteration that *must reach* the use from all possible paths in subflowgraph. To this end, for a basic block S, we define $MRD_{in}(S)$ as the set of variables that are always defined upon entering S, $MRD_{out}(S)$ as the set of variables that are always defined upon exiting S, and $pred(S)$ as the set of immediate predecessors of S in the loop's flow graph ignoring all the back edges. $MRD_{in}(S)$ can be computed using the following equations:

$$MRD_{in}(S) = \cap_{t \in pred(S)} MRD_{out}(t)$$

$$MRD_{out}(S) = (MRD_{in}(S) - KILL(S)) \cup DEF(S)$$

We start from a conservative initial solution, with each MRD_{in} an empty set ϕ. The back edges in the graph are removed because $MRD(S)$ is only concerned with the

values that are defined in the statements prior to S in the flow graph. Because back edges are deleted, the algorithm actually works on the DAG of the flow graph. Since back edges for inner loops carry information for the analysis of the outer loop, they are handled by the aggregation. The abstraction and aggregation of inner loops will be explained in the next section.

The value of each set defined above such as DEF, USE, $KILL$, and MRD, is a subset of VAR. Hence the domain of the data flow information set is the powerset $\mathcal{P}(VAR)$. The effect of a $\cup$ (union) operation is to form a union of its operands. It is precise in the sense that it will not summarize two sets unless the summary set has exactly the same members as the two sets. For instance, $\{\mathtt{A(I)} \cup \mathtt{A(1:N)}\}$ will return $\{\mathtt{A(I)}, \mathtt{A(1:N)}\}$ unless $\mathtt{I} \in [\mathtt{1:N}]$, but $\{\mathtt{A(1:N:2)} \cup \mathtt{A(2:N-1:2)}\}$ will return `A(1:N)`. The effect of a $\cap$ (join) operation is to form a join of its operands. It is conservative in the sense it will return an empty set ϕ if it cannot determine the join of its operands. For instance, $\{\mathtt{A(I)} \cap \mathtt{A(1:N)}\}$ will return ϕ unless $\mathtt{I} \in [\mathtt{1:N}]$. Because the join is conservative, there will be some potential loss of information at each join point of the flow graph. The effectiveness of the algorithm will hence depend on the system's ability to determine the relationship between symbolic variables. This issue will be discussed in Section 4.

An iterative algorithm for solving the MRD equation is shown in Figure 1 as phases 1 and 2. Phases 3 and 4 are explained below.

Abstraction for Inner Loops

When the algorithm finds a loop nested inside a loop body, it will recursively call itself on the inner loop. To hide the control flow of an inner loop, we introduce some abstraction and extend the previous definition from a basic block to a complete loop. We start by defining the information for one iteration of the loop.

Definition 5 Let L be a loop and VAR be the variables in the program. We define the following set as *summary set* for $body(L)$.

1. $DEF_b(L) := \{v \in VAR : v$ has a MRD reaching all exits node of $body(L)\}$
2. $USE_b(L) := \{v \in VAR : v$ has an outward exposed use in $body(L)\}$
3. $KILL_b(L) := DEF_b(L)$
4. $PRI_b(L) := \{v \in VAR :$ every use of v has a reaching MRD in $body(L)\}$ □

The summary set is an abstraction of the effect of a loop iteration on the data flow values. Using the summary set, we can ignore the structure of the inner loops in the analysis of the outer loop. The trade-off is that we have to make a conservative approximation and may lose information in the process.

- $DEF_b(L)$ is the *must define* variables for one iteration of L; i.e. the must define variables upon exiting the iteration:

$$DEF_b(L) = \cap(MRD_{out}(t) : t \in exits(L))$$

.

Algorithm Privatize
  $privatize := func(L)$
  Input: subflowgraph for loop L with back edges removed
  Output: $DEF(L), USE(L), PRI(L)$

Phase 1: Collect local information
  foreach statement $S \in body(L)$ in rPostorder do
    if $S \in control(M)$ for some loop M nested in L then
      ! `S` is an inner loop, visit `S` first
      $[DEF(S), USE(S)] \leftarrow privatize(M)$
      !collapse all nodes in M onto S
      $L \leftarrow COLLAP(L, M)$
    else
      compute local $DEF(S), USE(S)$
    endif
  endfor

Phase 2: Solve the MRD Data Flow Equations for each statement
  forall $S \in body(L)$ initialize $MRD(S) \leftarrow \phi$
  foreach $S \in body(L)$ in rPostorder do
    $MRD_{in}(S) \leftarrow \cap_{t \in pred(S)} MRD_{out}(t)$
    $MRD_{out}(S) \leftarrow (MRD_{in}(S) - KILL(S)) \cup DEF(S)$
  end

Phase 3: Compute Summary Sets for the Loop Body
  $DEF_b(L) \leftarrow \cap_{t \in exits(body(L))}(MRD_{out}(t))$
  $USE_b(L) \leftarrow \cup_{t \in body(L)}(USE(t) - MRD_{in}(t))$
  $PRI_b(L) \leftarrow (\cup_{t \in body(L)} USE(t)) - USE(L)$
  $PRI_b^{st}(L) \leftarrow DEF_b(L) \cap PRI(L)$
  $PRI_b^{dy}(L) \leftarrow PRI_b(L) - PRI_b^{st}(L)$

Phase 4: Return aggregated set $DEF(L)$ *and* `USE(L)`
  test if it is profitable to privatize $PRI_b(L)$
  determine last value assignment
  $[PRI^{st}(L), PRI^{dy}(L)] \leftarrow aggregate(PRI_b^{st}, PRI_b^{dy}, control(L))$
  $[DEF(L), USE(L)] \leftarrow aggregate(DEF_b(L), USE_b(L), control(L))$
  return $[DEF(L), USE(L)]$

Figure 1. Algorithm for Identifying Privatizable Arrays

- $USE_b(L)$, the *possibly outward exposed use* variables, is the set of variables that are used in some statements of L, but do not have an MRD_{in} in the same iteration:

$$USE_b(L) = \cup(USE(t) - MRD_{in}(t)) : t \in body(L)$$

.

- The *privatizable variables* are the variables that are used and not exposed to definitions outside the iteration:

$$PRI_b(L) = \cup\{USE(t) : t \in body(L)\} - USE_b(L)$$

In the analysis of the outer loop, we must consider the total effect of an inner loop on data flow values. That is, we need to account for the effect of back edges and index domain of the loop. We can do this by listing the summary set for each iteration of the loop. We will use an approximation called *aggregated set* to compute $DEF(L)$, $USE(L)$, $KILL(L)$, and $PRI(L)$. The aggregation computes the region spanned by each array reference in $USE_b(L)$, $DEF_b(L)$, $KILL_b$, and $PRI_b(L)$ across the iteration space. Because we only consider do loops, the aggregation is a relatively straightforward interpretation of loop index and boundaries in the do-entry of the loop. In our representation of variables, a subarray is represented as a subscripted variable together with a subscript range. To aggregate a subarray, we just need to concatenate the loop index and boundaries with the subscripted variable of subarray. For instance, if `I` is a loop index or an induction variable with value `[1:N:1]`, then `A(I,J)` will be aggregated as `(A(I,J),[I=1:N]) = A(1:N,J)` and `A(I,1:I)` will be aggregated as `(A(I,1:I),[I=1:N])`.

Because one iteration's use may only be exposed to the definitions in some previous iterations of the same loop, a naive aggregation of $USE_b(L)$ may exaggerate the exposed use set. The reason is that the uses covered by the definitions in previous iterations are not exposed to the outside of the loop, and therefore they should be excluded from the aggregated $USE(L)$ set. For instance, in

```
   DO I = 2, N
S1:   A(I) = A(I-1) + B(J)
   ENDDO
```

the information for one iteration is $USE_b(L) = \{\texttt{A(I-1)}, \texttt{B(J)}, \texttt{J}\}$ and $DEF_b(L) = \{\texttt{A(I)}\}$, the region aggregately defined in all iterations prior to the ith iteration is `A(2:I-1)`, and `A(I-1)` is exposed to definitions outside the loop only in the first iteration, that is, $USE(L) = \{\texttt{A(1)}\}$.

Profitability of Privatization

After an array is identified as privatizable in a loop, we need to determine if different iterations of a loop will access the same location of the array. For instance, in the following loop:

```
S1:  DO I = 1, N
S2:     A(I) = ...
          ...
```

```
Si:          = A(I)
           ...
Sn:  ENDDO
```

the algorithm will identify that `A(I)` is privatizable. We can privatize `A(I)` using a private scalar as follows:

```
C$DIR INDEPENDENT
C$DIR PRIVATE X
C$DIR LAST VALUE A(I) = X
S1:  DO I = 1, N
S2:     X  = ...
           ...
Si:           = X
           ...
Sn:  ENDDO
```

This transformation is useful for conventional compiler optimization. Today's optimizing compilers usually will not allocate a register to a subscripted variable `A(I)` in the original program because they have very limited capability to disambiguitize the array reference. In the transformed program, it is easy for them to allocate a register to a scalar `X`. The transformation can also reduce the amount of *false sharing* in multiprocessor caches. In a distributed memory system with *owner computes* rule [ZBG88][CK88b] [RP89], the transformed program effectively transfers the ownership of `A(i)` to iteration i; hence the processor scheduled to execute the iteration i can execute operations in `S2` even if it does not own `A(i)`. This transformation can facilitate data distribution to reduce communication and improve load balance [TP92].

For the purpose of eliminating memory-related dependences in this paper, the array `A` in the previous example need not be privatized. The condition for privatization exists when different iterations of the loop access same location. This can be determined by examining $PRI_b(L)$. We will call the test the *profitability test.* Let *A(r)* be a reference to array A where r is a subscript expression if $A(r)$ is a subscripted variable, or a range list if $A(r)$ is a subarray.

If $A(r)$ is a subscripted variable and r is a monotonic function of loop index i, then different iterations of i will access different locations of $A(i)$; hence it is not profitable to privatize $A(r)$, otherwise it is profitable. When there is more than one subscript of A in $PRI_b(L)$, we need to test if there are dependences among each pair of subscripted variables. We can use the Banerjee Test [Ban88] to determine if within the loop boundaries two references referred to the same location. If $A(r)$ is a subarray, we need to determine if there is an iteration $j \neq i$ such that $A(r) \cap A(r[i/j]) \neq \phi$, where $r[i/j]$ represents r after we substitute each appearance of loop index i with j. Again one has to test for each pair of occurrences if there is more than one occurrence of subarrays. This discussion is summarized in the algorithm shown in Figure 2.

Last Value Assignment

Live Analysis

Live analysis is needed to determine if a privatizable variable is live after exiting the loop. If it is live, the last-value assignment will be necessary to preserve the seman-

Algorithm Profitability Test

Input: PRI_b for loop L: with index $i \in [p:q:t]$

Output: PRO, arrays profitable for privatization

$PRO \leftarrow \phi$
foreach $A \in PRI_b$ do
  $ALL_A \leftarrow \{A(r) : A(r) \in PRI\}$
  foreach pair $A(x), A(y) \in ALL_A$ — where x and y can be the same
    let $X \leftarrow$ set of values in x
    let $Y \leftarrow$ set of values in y
    if $(\exists j \in [p:q:t] | j \neq i, X[i/j] \cap Y \neq \phi)$
      $PRO \leftarrow PRO + A$
    !Notice that if $x = y$ and x does not contain i, the test is satisfied.
  endfor
endfor

Figure 2. Profitability Test

tics of the original program; otherwise no last-value assignment is needed for that variable. A last-value assignment statement can be ignored when the private array is not used after the loop, or there are subsequent definitions of the array before any use.

Definition 6 Let S be a node in the flowgraph. The live variables at the bottom of S is the set of variables that may be used after control passes the bottom of S. We define

1. $LVBOT(S) := \{v \in VAR : v$ may be used after $S\}$
2. $LVTOP(S) := \{v \in VAR : v$ may be used after S or in $S\}$ □

Let $succ(S)$ be the set of immediate successors of S in the program flowgraph. The equations for $LVTOP$, $LVBOT$ are

$$LVBOT(S) = \cup_{t \in succ(S)} LVTOP(t)$$

$$LVTOP(t) = (LVBOT(S) - KILL(S)) \cup USE(S)$$

The algorithm traverses the flow graph backward and uses the aggregated set for each loop. This algorithm is just the natural extension of scalar live analysis to include array references.

Static and Dynamic Last-Value Assignment

After live analysis, we can ignore the last-value assignments for private arrays that are not live at the bottom of the loop. However, the remaining live private arrays have to be copied to their global counterparts. Two problems prevent static determination of iteration that copies its private array to the global array. One, as shown in our early example, is due to conditional definition. Without information about

Algorithm Write Back Set
Input: PRI_b^{st} for loop L: with index $i \in [p:q:t]$
Output: WBS, for iteration i

$WBS \leftarrow \phi$
foreach array $A \in PRI_b^{st}$ do
$ALL_A \leftarrow \{A(r) : A(r) \in PRI_b^{st}\}$
$WBS \leftarrow ALL_A - \cup_{j \in [i+t:q:t]} ALL_A[i/j]$
endfor

Figure 3. Compute Write Back Set

which branch the program will take at runtime, it is impossible to determine which iteration shall assign the last value. Another problem is that some complicated subscript expressions make it inefficient to compute at compile time which iteration will assign the last value. In these cases, we will use well-known run-time techniques such as [ZY87] to resolve the output dependences.

Our first step is to identify the private arrays that need dynamic last-value assignments because of conditional definition. PRI_b contains all the array uses that are covered by some definition in the same iteration of the loop; some of the uses are conditional, where they are covered by some conditional definition. DEF_b contains all the variables that must be defined in every iteration of the loop. Therefore, $PRI_b^{st} = PRI_b \cap DEF_b$ contains the privatizable arrays that are unconditionally defined. Hence $PRI_b^{dy} = PRI_b - PRI_b^{st}$ contains the conditionally defined privatizable arrays.

Because of the profitability test, at least one element of the array in PRI_b^{st} is defined in several iterations. To determine for each iteration what element has to be copied back to the global array, we define a *write back set* as the sections of private array that have to be copied back to the global array for iteration i.

Definition 7 Let L be a loop body and PRI^{st} be the static private arrays. The write back set (WBS) of L for iteration i is defined as the sections of arrays in PRI^{st} that are written in the ith iteration, but are not written thereafter. □

¿From the definition we can compute the WBS by comparing the set defined in iteration i and the set defined in the iterations after i. The algorithm is shown in Figure 3.

Note that the last iteration of loop L will always write back all its static private arrays. When we cannot find a closed form for WBS, we can move the array to PRI_b^{dy} and use run-time resolution. Actually the algorithm itself can be linked into the program to perform a run test for each iteration. In most cases, the algorithm will find a closed form and therefore WBS can be determined at compile time. The following example shows how the algorithm in Figure 3 works in two different situations.

```
S1:  DO I = 1, N
S2:     DO J = 1, M
S3:        A(J) = ...
S4:        B(I+J) = ...
S5:     ENDDO
        ...
Sn:  ENDDO
```

For loop S1, $PRI_b^{st} = \{A(1:M), B(I+1:I+M)\}$. A(1:M) will be accessed in all iterations after a given I<N because A(1:M) does not depend on I. Hence WBS for A in iteration $I \neq N$ is ϕ, the empty set. Only the last iteration of loop S1 will copy out A(1:M). For B, B(I+1:I+M) is in ALL_B for iteration I, B((I+1)+1:M+N) is modified in iterations from I+1 to N, hence the WSB for B is B(I+1).

Interprocedural Analysis of Privatizable Arrays

In many cases, we need to do interprocedural analysis for array privatization. We can find more deeply nested loops by looking at the loops in the subroutines. To use the algorithm for interprocedural analysis, we generalize the loop flow graph to incorporate subroutine bodies.

Definition 8 Let R be a subroutine and VAR be the variables in the subroutine. We define the *subroutine summary set* for R as follows:

1. $DEF(R) := \{v \in VAR : v$ has a MRD reaching all exits node of $R\}$
2. $USE(R) := \{v \in VAR : v$ has an outward exposed use in $R\}$
3. $KILL(R) := DEF(R)$ □

The algorithm to find the *subroutine summary set* is the same used above to compute DEF_b, USE_b, and $KILL_b$. The input to the algorithm is now the flowgraph of the subroutine. We run the algorithm in bottom-up order on the program call tree, such that each time we encounter a subroutine call in a program, the summary set for the subroutine has already been computed. When the algorithm finds a subroutine call node, it reads the summary set of the subroutine, simplifies the summary set to get rid of variables that are not visible to the caller, and maps the formal parameters and common variables in the summary set to the corresponding actual parameters and common variables at the caller. Because array reshaping usually occurs in programs written in Fortran, the array defined in the subroutine may have a different shape. Our interprocedural mapping program will linearize an array in the subroutine if it has a different number of dimensions as in the caller. After the specialization, the algorithm will use the subroutine summary set for the call statement.

We implemented the algorithm with interprocedural analysis in the POLARIS system. It allows us to do automatic array privatization in loops with subroutine calls.

Automatic versus Manual Array Privatization

To evaluate the effectiveness of the algorithm, we ran the automatic array privatization on the Perfect Benchmarks. We compared the number of private arrays found by the algorithm with that of the manual array privatization reported in [EHLP91]. The result is shown in Table 1. The first column reports the number of private arrays identified by both manual and automatic privatization. The second column reports the number of private arrays identified by manual privatization but not by automatic privatization. The third column reports the number identified by automatic privatization but not by manual privatization.

Program	Automatic and Manual	Manual Only	Automatic Only
ADM (AP)	2	12	0
ARC2D (SR)	0	2	0
BNDA (NA)	12	3	4
DYFESM (SD)	0	1	11
FLO52 (TF)	0	0	4
MDG (LW)	17	1	1
MG3D (SM)	1	4	0
OCEAN (OC)	4	3	0
QCD (LG)	22	7	0
SPEC77 (WS)	25	14	0
TRACK (MT)	20	2	0
TRFD (TI)	4	0	0

Table 1. Number of Private Arrays

By comparing the results of automatic privatization and manual privatization, we found that the algorithm is sufficient to discover most of the privatizable arrays. The lattice for array reference is also adequate for representing the array use and definition in the programs of Perfect Benchmarks. Where our algorithm failed, we found that most instances were due to lack of information about the loop boundaries at compiler time. Some of the ambiguities can be resolved by a more powerful forward substitution algorithm than that available in the current system. For instance, in *DYFESM* our algorithm failed to identify `XE` in the subroutine *solvh* because the array boundary passed as common variables. `XE` is defined in the *geteu* subroutine as a two-dimensional array `XE(NDDF,NNPED)` and used in *solvhe* as a one dimensional array `XE(NDFE)`. It turns out that `NDFE = NDDF*NNPED` after interprocedural forward substitution.

Some of the ambiguities can be resolved by enhancing the traditional scalar constant propagation and forward substitution. For instance, a common difficulty is conditionally defined loop boundaries. Such a situation occurs in subroutine *initia* of *ARC2D* code. Two common variables `JLOW` and `JUP` are defined as follows:

```
IF (.NOT.PERIDC) THEN
   JLOW = 2
```

```
      JUP = JMAX - 1
   ELSE
      JLOW = 1
      JUP = JMAX
   ENDIF
```

These two common variables are then used in subroutine *filerx, filery* as loop boundaries:

```
L1: DO N = 1, 4
L2:    DO J = JLOW, JUP
          DO K = KLOW, KUP
             WORK(J,K,1) = ...
          ENDDO
       ENDDO
       IF (.NOT.PERDIC) THEN
L3:       DO K = KLOW, KUP
             WORK(1,K,1) = WORK(2,K,1) + ...
             WORK(JMAX,K,1) = WORK(JMAX-1,K,1)
          ENDDO
       ENDIF
       ...
    ENDDO
```

For the array `WORK` to be privatized in loop `L1`, we need to determine the reference `WORK(2,K,1)` and `WORK(JMAX-1,K,1)` in loop `L3` are defined in `L2`. If we inspect the condition in the *IF* statement, we can determine that when `L3` is executed the condition is `(.NOT.PERDIC)`. Under the same condition, `JLOW=2, JUP=JMAX-1`; hence the use is covered by the definition in `L2`. A simpler method is to propagate the upper bound of `JLOW=2` and lower bound of `JUP=JMAX-1`, and we do not need to interpret the condition of *IF* statement.

A problem of similar nature happens in *BDNA*. It involves the bound of an induction variable L:

```
L1: DO I = 2, NSP
L2:    DO J = 1, I-1
          XDT(J) = ...
          YDT(J) = ...
          ZDT(J) = ...
       ENDDO
       L = 0
L3:    DO J = 1, I-1
          IF (IND(J).NE.0) THEN
             L = L+1
          ENDIF
       ENDDO
L4:    DO J = 1, L
          ... = XDT(J)
          ... = YDT(J)
```

```
          ... = ZDT(J)
       ENDDO
    ENDDO
```

For XDT,YDT,ZDT to be private to loop L1, the use in L4 must be defined in L2. The condition is L<=I-1. Computing the upper bound for L in loop L3, the validity of this condition in L4 can be confirmed.

The idea of keeping more than one possible value for scalars can be generalized to propagate the value for each element of an array. This is very useful in the case of subscripted subscripts. For instance, in *ARC2D*, array JPLUS and JMINU are used to store the neighboring element on a ring of size JMAX. That is, JPLUS(I) = I+1 mod JMAX, and JMINU(I) = I-1 mod JMAX. These values are used throughout the program. By propagating the value of the whole array, we can use our algorithm to compute the range defined and used.

Another interesting case arises in MDG, where we need to inspect the condition of an IF statement to determine whether the array RL can be privatized in subroutine *poteng* and *interf*:

```
    DO I = 1, NMOL1
L2:    DO J = I+1, NMOL
          KC = 0
          DO K = 1, 9
             RS(K) = ...
C1:          IF (RS(K).GT.CUT2) THEN
                KC = KC + 1
             ENDIF
          ENDDO
          DO K = 2, 5
C2:          IF (RS(K).LE.CUT2) THEN
                RL(K+4) = ...
             ENDIF
          ENDDO
C3:       IF (KC.EQ.0) THEN
             DO K = 11, 14
                ... = RL(K-5)
             ENDDO
          ENDIF
       ENDDO
    ENDDO
```

Note that whenever (KC.EQ.0) in C3 is true, (RS(K).LE.CUT2) must also be true, because if (RS(K).GT.CUT2), then KC must be greater than 0 due to the increment in C1. Hence RS(11:14) is privatizable in L2 because whenever RS(11:14) is used, it refers to the values defined in the same iteration.

In some cases, user direction is needed to determine if an array is privatizable. This happens in OCEAN, where it cannot be statically determined if the array C and CA are defined in subroutine *in*. The *in* writes to the C and CA, but the definitions are surrounded by an error condition test. Without knowing whether the error condition will abort the program, the algorithm has no way of knowing those arrays are defined whenever the program returns from calling *in*.

Determine the Relationship between Symbolic Variables

In the last section, we showed that to determine the region of an array that is used in a program, a major task is to determine the relationship between symbolic variables. In this section, we present a demand-driven technique to determine the value relationship between symbolic variables. This technique is based on the Static Single Assignment (SSA) form. SSA is an intermediate representation of a program which has two useful properties:

1. Each use of a variable is reached by exactly one definition to that variable.
2. The program contains `PHI` functions that merge the values of a variable from a distinct incoming control-flow graph.

[AWZ88, RWZ88, WZ91] present various applications of SSA, and [CFR+91] deals with efficiently transforming programs into SSA form.

Determine Symbolic Value on Demand

The SSA form can be used to track the value of symbolic variables on demand. For instance, in the following SSA representation of a piece of code from *DYFESM* we can determine that the value of `NDFE_1` used in `L3` is `NDDF_1*NNPED_1` since `NDFE_1` is assigned in `S1`.

```
S1: NDFE_1 = NDDF_1 * NNPED_1
    ...
    DO K_1 = 1, N_1
L1:    DO I_1=1, NDDF_1
L2:       DO J_1=1, NNPED_1
             XE(I_1,J) = ...
          END DO
       END DO
       ...
L3:    DO I_2 = 1,NDFE_1
          ... = XE(I_2)
       END DO
    END DO
```

The loop boundaries for `L1,L2` are the same `NDDF_1` and `NNPED_1`. Hence the `XE(1:NNDF_1,1:NNPED_1)` defined in `L1` covers the `XE(1:NDFE_1)=XE(1:NDDF_1*NNPED_1)` used in `L3`. Because in the SSA representation, each use of a variable can be reached by exactly one assignment to that variable, it is very easy to track the value of a variable by its name.

In the traditional forward substitution, because it is difficult to know which variable should be forward substituted beforehand, it usually substitutes all the variables in a program and rolls back variables later to reduce redundant computation. In contrast, the backward tracking of a value through the variable name is done on demand.

Dealing with Conditionals

As shown in the previous section, to determine the value of a symbolic variable, the conditional statement must be handled properly. One difficulty in dealing with a conditional statement is to propagate the condition from the assignment of a variable to the use of the variable. For the example code from *ARC2D*, it is difficult to know that the condition guarding the assignment to JLOW_1 is the the same condition guarding the use of JLOW_1. In this case, the unique variable name in the SSA representation can be used as a handle to link the scattered information.

The SSA representation for the *ARC2D* example is as follows (we only show the SSA for variables involved in the loop boundaries):

```
    IF (.NOT.PERIDC) THEN
       JLOW_1 = 2
       JUP_1 = JMAX - 1
    ELSE
       JLOW_2 = 1
       JUP_2 = JMAX
    ENDIF
    JLOW_3 = PHI(Cond(.NOT.PERIDC),JLOW_1,JLOW_2)
    JUP_3 = PHI(Cond(.NOT.PERIDC),JUP_1,JUP_2)

L1: DO N = 1, 4
L2:    DO J = JLOW_3, JUP_3
          DO K = KLOW, KUP
             WORK(J,K,1) = ...
          ENDDO
       ENDDO
S1:    IF (.NOT.PERDIC) THEN
L3:       DO K = KLOW, KUP
             WORK(1,K,1) = WORK(2,K,1) + ...
             WORK(JMAX,K,1) = WORK(JMAX-1,K,1)
          ENDDO
       ENDIF
       ...
    ENDDO
```

Note that the PHI functions for JLOW_3 and JUP_3; it is inserted in the program to distinguish values of a variable from different branches of the control flow graph, in this case the different branches of a conditional statement. We use an extension of PHI function to include the predicate for the conditional statement, that is, Cond(.NOT.PERIDC) specifies the condition in the IF statement, if Cond(.NOT.PERIDC) is satisfied, JLOW_3 will take the value of the second parameter of the PHI function which is JLOW_1, if Cond(.NOT.PERIDC) is false, it will take the value of the third parameter JLOW_2.

We will first show how to interpret the conditional statement and then show how to compute the upper and lower bounds for the variables. For loop L3 to be executed, the condition (.NOT.PERDIC) must be true since L3 is control dependent on the S1. Tracing the values of JLOW_3,JUP_3 to the PHI function we

know that they will have value JLOW_1,JUP_1. Tracing the value of JLOW_1,JUP_1 further, we have JLOW_3=JLOW_1=2,JUP_3=JUP_1=JMAX-1. Since now the value of JLOW_3=2,JUP_3=JMAX-1 matches the value for the subscript of the WORK(2,K,1) and WORK(JMAX-1,K,1), we do not need to trace back the value for JMAX any further. At this point we can compare the array region WORK(JLOW_3:JUP_3,KLOW:KUP,1) defined in loop L2 which is WORK(2:JMAX-1,KLOW:KUP,1) with the array regions WORK(2,KLOW:KUP,1) and WORK(JMAX-1,KLOW:KUP,1). The definition region covers the use region and WORK array is privatizable to loop L1.

As discussed before, in this example it is sufficient to prove WORK is privatizable just by showing the lower bound for JUP_3>=JMAX-1 and the upper bound for JLOW_3<=2. To compute the bounds for a variable, we can make a conservative choice at each PHI function and ignore the predicate for the conditional statement. We start by tracing back the values for variables until they are unified. Then we take the max or min on the second and third parameter of a PHI function and ignore the predicate.

```
max(JLOW_3) = max(PHI(Cond(.NOT.PERIDC),JLOW_1,JLOW_2))
   = max(PHI(Cond(.NOT.PERIDC),2,1)) = max(2,1) = 2
```

and

```
min(JUP_3) = min(PHI(Cond(.NOT.PERIDC),JUP_1,JUP_2))
   = min(PHI(Conc(.NOT.PERIDC),JMAX-1,JMAX)) = JMAX-1
```

In addition to being goal directed and on demand, the backward tracing scheme can stop the tracing at the correct point in the unification of variables; that is, when all the variables in the expressions are the same. After that, the additional unwinding of values may not gain any more information. In a forward propagation scheme, everything must start from the most primitive variables and in the case of several level of conditionals, the number of branchs may quickly explode and complexity may grow out of control. Because the backward tracing is goal directed and incremental, we can easily set complexity constraints such as setting the maximum backward tracing level to a fix number of nested PHI functions. After that, the algorithm can give up and degrade gracefully to reduce time spent on compiling the program.

Bounds for Monotonic Variables

Induction variables and other variables will have values dependent on the structure of the loop to which they are assigned. For an induction variable, its last value can be determined by technique in induction variable substitution such as presented in [Wol92]. In this section, we will show how to estimate bounds for monotonic variables.

Using the SSA form of loop L3 in the previous example from *BDNA*, we have:

```
        L_1 = 0
L3:     DO J = 1, I-1
           L_2 = PHI(L3,L_1,L_4)
           IF (IND(J).NE.0) THEN
```

```
            L_3 = L_2 + 1
         ENDIF
         L_4=PHI(Cond(IND(J).NE.0),L_3,L_2)
      ENDDO
```

The original loop L4 will use L_2 as the value of L. In this example, we further extend the PHI function to include loop label L3 in it to identify the loop control. Following the terminology used by Wolfe on induction variables[Wol92], L_2 will appear as a *Strongly Connected Region (SCR)* that includes a loop header PHI function and some conditional PHI functions. Because we need to know the upper bounds of L_2, we actually want to find the SCR with maximum increment to L_2. Similarly, for the lower bound of a monotonic variable, we need to find the SCR with minimum increment for L_2. This can be accomplished by backward tracing and compute bounds on PHI function as follows:

```
L_2 = PHI(L3,L_1,L_4) = PHI(L3,0,PHI(Cond(IND(J).NE.0),L_3,L_2)
   = PHI(L3,0,PHI(Cond(IND(J).NE.0),L_2+1,L_2))
```

Hence the longest SCR will have value:

```
max(L_2)=max(PHI(L3,0,PHI(Cond(IND(J).NE.0),L_2+1,L_2)))
   =(I-1)*(max(0,PHI(Cond(IND(J).NE.0),L_2+1,L_2)))
   =(I-1)*(max(0,max(PHI(Cond(IND(J).NE.0)),L_2+1,L_2)))
   = (I-1)*max(0,1)=I-1
```

The way to handle loop PHI function is to take its trip count and multiply the trip count to the maximum increment in the SCR. To find the maximum increment for a loop, we will choose the branch with maximum increment in a conditional PHI function. In the example, the monotonic variable is L_2 and its maximum increment for each iteration is 1. This technique can be generalized to compute lower bound for monotonic variables by taking a minimum value over the PHI functions.

Index Arrays

The use of the index array in the program makes it difficult to determine the array reference region in a program. ARC2D uses an index array JPLUS. It is assigned as follows:

```
L1:  DO J=1, JMAX
        JPLUS(J) = J+1
     END DO
     JPLUS(JMAX)=JMAX
     ...
L2:  DO J = 1, JMAX
        ... = ... FLUX(JPLUS(J),K,N)
     END DO
```

We can use the SSA representation to find out the value of JPLUS(J) in loop L2. We will extend the SSA representation to array in the following way: (1) create a

new array name for each array assignment; (2) use the subscript to identify which element is assigned; (3) replace the assignment with a special PHI function which will be written as MU((subscript),assignment,old). The assignment A(I) = exp is converted to A_1 = MU((I), exp, A_0), which is interpreted as element A_1(I) will take the value of exp in the assignment while other elements of A_1 will take the value in the A_0 as before the assignment. Using this extension, our example can be transformed into the following SSA form:

```
L1:  DO J=1, JMAX
        JPLUS_2 = PHI(L1,JPLUS_1,JPLUS_0)
        JPLUS_1 = MU((J),J+1,JPLUS_2)
     END DO
     JPLUS_3 = MU((JMAX),JMAX,JPLUS_2)
     ...
L2:  DO J = 1, JMAX
        ... = ... FLUX(JPLUS_3(J),K,N)
     END DO
```

For subscript expression JPLUS_3(J) in loop L2, it can be evaluated as follows.

```
   JPLUS_3(J) = (MU((JMAX),JMAX,JPLUS_2))(J)
      = (MU((JMAX),JMAX,PHI(L1,JPLUS_1,JPLUS_0)))(J)
      = (MU((JMAX),JMAX,PHI(L1,MU((J),J+1,JPLUS_2),JPLUS_0)))(J)
      = (MU((JMAX),JMAX,MU(([1:JMAX]),J+1,JPLUS_0)))(J)
```

The expression can be interpreted as

```
JPLUS_3(J) = IF J=JMAX THEN
                JMAX
             ELSEIF J in [1:JMAX] THEN
                J+1
             ELSE
                JPLUS_0(J)
             ENDIF
```

Note that the PHI function for loop L1 defining an aggregated region of the index array.

Conclusion

We presented algorithms to automatically identify privatizable arrays in sequential Fortran programs and use array privatization to eliminate memory-related dependences. The algorithms have been implemented in the POLARIS system to perform array privatization both intraprocedurally and interprocedurally. Our experiments have thus far indicated that the algorithms can privatize most of the arrays privatized by hand in [EHLP91]. To increase the effectiveness of the algorithms, it seems necessary to use more sophisticated techniques for determining the equivalence of symbolic variables and interprocedural values and bounds propagation of symbolic variables. To this purpose, we proposed a goal-directed technique which uses the

SSA form of a program to determine values and bounds of symbolic variables in the presence of conditional statements, loops, and index arrays. We are currently implementing our symbolic analysis algorithms and studying the application of array privatization to data distribution for greater local access and better load balancing.

Appendix D: Data-Dependence Testing in Polaris

Reference pattern analysis in real programs

Table 26 shows the results of an analysis of the array subscript expressions in the most time-consuming loops in the suite of Perfect Bechmarks® programs[1], that is, all the loops that were manually parallelized in order to achieve the performance previously reported[EHJP92]. This performance, and its comparison to the performance achieved by commercially available compilers, is displayed in Table 25. The last two columns of Table 26 show the programs that need be preprocessed by two transformation techniques, array privatization and reduction analysis. We are implementing these two transformation techniques in separate subprojects. For the purpose of this appendix we assume that they have been successfuly applied.

program	Automatically compiled		Manually improved	
	FX/80	Cedar	FX/80	Cedar
ARC2D	8.7	13.5	10.6	20.8
FLO52	9.0	5.5	14.6	15.3
BDNA	1.9	1.8	5.6	8.5
DYFESM	3.9	2.2	10.3	11.4
ADM	1.2	0.6	7.1	10.1
MDG	1.0	1.0	7.3	20.6
MG3D	1.5	0.9	13.3	48.8
OCEAN	1.4	0.7	8.9	16.7
TRACK	1.0	0.4	4.0	5.2
TRFD	2.2	0.8	16.0	43.2
QCD	1.1	0.5	2.0	20.8
SPEC77	2.4	2.4	10.2	15.7

Table 25: Speedups versus serial for Perfect Benchmarks programs on Alliant FX/80 and Cedar. Automatically compiled with Kap and manually optimized variants.

Simple index patterns

The programs generally show very simple reference patterns. Column one of Table 26 marks the programs that use *simple indices*. A simple index consists of the loop variable only, such as `a(i)` or `b(i,j)`. Such patterns are frequent in 5 of the 12 inspected codes. In BDNA, SPEC77, Flo52, and ARC2D simple indices are the only significant data dependence pattern that needs to be analyzed to parallelize the codes successfully. The code ADM exhibits simple subscript patterns in addition to more difficult situations that we will discuss below. Two of these five programs, namely Flo52 and Arc2D, are successfully parallelized by available compilers. The

[1] The Perfect Benchmarks serve as an important first test suite for the evaluation of our compilation techniques, as we can take advantage of previous experiences in analyzing and parallelizing these programs. We plan to extend our test suite to include codes from the Spec/Perfect II benchmarks as well as several Grand Challenge applications.

program	simple subscripts	ranges	subscripted subscripts	other	private variables	reductions
ADM	x	x	x		x	x
ARC2D	x				x	
BDNA	x				x	x
DYFESM		x	x		x	x
FLO52	x					
MDG		x			x	x
MG3D					x	x
OCEAN		x			x	
QCD		x	x	x	x	x
SPEC77	x			x	x	x
TRACK			x		x	x
TRFD		x	x		x	x

Table 26: Subscript expressions in the important program sections of the Perfect Benchmarks suite

reason for failing parallelization of current compilers in the other three programs are limitations in their capabilities to recognize data that can be privatized to a loop or that form reduction operations.

Range patterns

We use the term *range* patterns for situation where loop iterations access adjacent subranges of arrays. These subranges are typically accessed in the following forms (the objective is to parallelize the outer loop of the nests).

```
    DO 10 i=1,n,k
       DO 5 j=1,k
          a(i+j) = ...
5      ENDDO
10  ENDDO
```

Patterns of this type were found in the programs ADM and QCD. Sometimes these ranges are built by induction variables. Examples can be found in program TRFD:

```
    DO 10 i=1,n
       DO 5 j=1,k
          a(ind) = ...
          ind = ind + 1
5      ENDDO
10  ENDDO
```

In program OCEAN, ranges are defined in expressions such as:

```
    DO 10 i=1,n
       DO 5 j=1,k
          a(y*i + x*j) = ...
5      ENDDO
10  ENDDO
```

where it can be symbolically proven that $max(x*j) < y$. That is, the array range accessed in the inner loop is less than the step taken from iteration to iteration of the outer loop. A related pattern occurs in program MDG. This pattern illustrates the advantage of the techniques we will describe in the next section over existing data dependence tests: Test algorithms for the specific range patterns are implementable without excessive cost, even including the symbolic analysis and manipulation facilities needed to extract the relationship of the coefficients (y and x in the example) from the program text. On the other hand, existing data dependence test do not handle symbolic coefficients, and adding this capability might make the tests very slow.

More complex patterns of range accesses are found in program DYFESM, where loops of the following form are typical:

```
    DO 10 i=1,n
       is = start(i)
       il = length()
       DO 5 j=is,is+il
          a(j) = ...
5      ENDDO
10  ENDDO
```

¿From the initialization of the arrays `start` and `length` in program DYFESM, the relationship `start(i+1) > start(i)+length(i)` can be determined, which guarantees that adjacent ranges do not overlap. This is an example where we need both range analysis and subscripted subscript analysis, as discussed next.

Subscripted subscript patterns

Subscripted subscript patterns are the main reason why existing data dependence tests fail, even after successfully privatizing data and handling reduction operations. If an array is indexed with another array (such as `a(table(i))`) the tests conservatively assume dependence. There is no simple way around these problems, however our code pattern analysis has revealed that in a substantial amount of the important program segments there is a way for the compiler to detect independence. We have found that, in many programs, subscript arrays are actually initialized in the program. The values are either known at compiletime or enough can be determined about their relationship so that data dependence tests can prove loop parallelizability.

In programs TRFD and QCD we have found such situations. Currently we are not yet in the position to do the symbolic analysis necessary to furnish data dependence tests with the necessary information. However it is quite important to state the fact that these values are known at compiletime and from this point of view there is no obstacle to develop appropriate analysis techniques. In program DYFESM we

have also found subscript arrays that are initialized with known values, as in the example above. However, there are a few cases in DYFESM where it is difficult and perhaps impossible to gather enough information at compiletime and we may have to resort to runtime data-dependence tests. In two more programs, namely ADM and TRACK we will have to do additional studies of the situations. In both programs we have found variables that are computed and modified in non-trivial functions, including table lookup, before they are used as array indices. Further investigation will show to what extent we can prove loop independence at compile time.

Other patterns

Other patterns are sometimes important. Two examples are found in the programs SPEC77 and QCD. In Spec77 there is an important loop that reads an array some of whose elements are set to different values for part of the loop iteration and then set back. One can duplicate this array, one version having the altered values, which will result in read-only references of this array and thus independence.

Another interesting situation is in QCD where all data dependences seem detectable at compiletime, except for those introduced by a random number generator. From a high-level problem point of view there is no necessity for keeping the sequence of random numbers the same when going from a serial to a parallel execution of the program (except perhaps for test purposes). In fact, in some manual experiments we have replaced the generator with a parallel algorithm, which allowed the whole program to be executed in parallel. A compiler does not have the high-level knowledge necessary to make this replacement. However it is conceivable that such random number generators can be identified and the user gets queried whether the replacement is permissible.

Work in data dependence analysis

We are currently extending existing technology in data dependence analysis in two areas. First we will develop symbolic analysis facilities that will allow us to gather from the whole program text information necessary to perfom dependence tests. We have described some of the pertinent program patterns in Section 6.3. The description of our symbolic analysis and manipulation facilities is not object of this report.

Second we will add to the repertoire of available new test algorithms a method that we term *range test*. This test is devised to solve the simple subscript and range patterns identified and described in the previous section. The range test is not a test in the traditional sense in that it does not do a pairwise subscript analysis. Instead it performs an expensive analysis of loop subscripts for each variable once per loop. Consequently it does not produce a data-dependence graph and does not produce direction vector information. Direction vectors are important for synchronized doacross loops, which we have found in our hand-analysis of codes to be of minor importance. Thus, having our new test available, a driver for data dependence tests would try to detect independence using the range test, and if it failed, a full analysis would be done using a traditional test in order to generate synchronized loop schedules.

The range test works in two steps. For each non-privatized read-write variable of a given loop nest it first gathers all subscript expressions. They must be equal except for a loop-invariant terms, otherwise the test assumes dependence. This first step detects the simple index patterns as a by-product. Arrays indexed with such patterns do not have cross-iteration dependences. The second step of the test then computes the *difference* and *magnitude* of the subscript expression with respect to each enclosing loop. The difference is the change in the subscript value when the loop advances by one iteration. The magnitude is the change in the subscript value between the first and the last iteration. A loop is now independent if there is an ordering of the loops such that the difference is greater than the sum of the magnitudes of all lower-order loops. Intuitively this means, the array range accessed by the "inner" loops is less than the step of the enclosing loop; thus, adjacent ranges do not overlap. Difference and magnitude can be determined in many non-linear subscript functions, some of which arise from the substitution of induction variables in triangular loops. We have found such loops in important program sections. The analysis of non-linear subscripts is not possible with known data-dependence tests. The ability to handle non-linear and symbolic terms, and the fact that simple subscript and range patterns are most common, constitute the power of the range test. Our preliminary results indicate that this test is capable of handlling most of the situations arising in real programs. Specifically, our test was able to identify the simple index patterns shown in Table 26. Thus, in combination with our privatization and parallel reduction transformations our test will be able to parallelize the codes to the full extent of our previous manual optimization.

Conclusions

The study of subscript patterns in the time-consuming loops of a sizeable program suite has revealed that simple subscript patterns are very frequent, and where they are complex, they need advanced symbolic analysis and subscripted subscript handling capabilities rather than new tests that can solve the general integer equation problem more efficiently. Our analysis agrees with earlier studies done by Yew/Li[SLY90], who found a large number of simple subscript patterns. It also agrees with the findings of Petersen/Padua[PP93] who showed that new, more powerful linear subscript tests add little to the program performance.

Our study has not only shown what is needed to advance the situation in data dependence testing, it has also shown us how far we can get with improved methods. We have analyzed all loops of the programs that we have manually optimized in previous experiments and where we have achieved satisfactory performance. Our analysis now shows that in many programs we can do the full data dependence testing to match this performance. The fact that we found it to be possible to determine at compile-time information about the values and relations of many variables that are crucial for data dependence testing, gives strong indication that with future development of analysis methods this goal can be reached.

References

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[BCFH89] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[CF87] Ron Cytron and Jeanne Ferante. What's in a Name? or The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proc. 1987 International Conf. on Parallel Processing*, pages 19–27, August 1987.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

[CK88b] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[EHJP92] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, 1992.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Proc. 4-th Workshop on Programming Languages and Compilers for Parallel Computing*. Pitman/MIT Press, August 1991.

[Fea88] P. Feautrier. Array expansion. In *Proc. 1988 ACM Int'l Conf. on Supercomputing*, July 1988.

[For93] High Performance Fortran Forum. High performance fortran language specification (draft). Technical report, High Performance Fortran Forum, January 1993.

[Li92] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of ICS'92*, pages 313–322, 1992.

[MAL92] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proc. 5rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[PP93] Paul M. Petersen and David A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. In *Proc. of ICS'93, Tokyo, Japan*, July 1993.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. the SIGPLAN '89 Conference on Program Language Design and Implementation*, June 1989.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computation. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):350–364, July 1990.

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines, to appear on ACM SIGPLAN Notices 1993*, September 1992.

[Wol82] Michael Joseph Wolfe. Optimizing supercompilers for supercomputers. Technical Report UIUCDCS-R-82-1105, Department of Computer Science, University of Illinois, October 1982.

[Wol92] Michael Wolfe. Beyond induction variables. *ACM PLDI'92*, 1992.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.

[ZY87] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.