Interprocedural Analyses of Fortran Programs

Béatrice CREUSILLET and François IRIGOIN*

Centre de recherche en informatique, École des mines de Paris

35, rue Saint-Honoré, 77305 FONTAINEBLEAU cedex, FRANCE

Abstract

Interprocedural analyses (IPA) are becoming more and more common in commercial compilers. But research on the analysis of Fortran programs is still going on, as a number of problems are not yet satisfactorily solved and others are emerging with new language dialects. This paper presents a survey of the main interprocedural analysis techniques, with an emphasis on the suitability of the analysis framework for the characteristics of the original semantic problem. Our experience with the PIPS interprocedural compiler workbench is then described. PIPS includes a `make`-like mechanism, PipsMake, which takes care of the interleavings between top-down and bottom-up analyses and allows a quick prototyping of new interprocedural analyses. Intensive summarization is used to reduce storage requirements and achieve reasonable analysis times when dealing with real-life applications. The speed/accuracy tradeoffs made for PIPS are discussed in the light of other interprocedural tools.

Key words: Interprocedural analysis, parallelization, re-engineering, PIPS.

*E-mail: {`creusillet,irigoin`}@`cri.ensmp.fr`

Introduction

Real life applications are split into several procedures to factorize code as much as possible and to improve readability. Traditional compilers only provide separate compilation, and procedure boundaries prevent many analyses and optimizations, such as constant propagation for instance, although they are of paramount importance in parallelizing compilers [12, 66] to achieve reasonable performances. They are also more and more necessary for sequential machines. *Interprocedural* techniques have been introduced to cope with this problem.

To our knowledge, the oldest article about interprocedural analyses was written more than twenty years ago by Allen [2]. Since then, research has focused on three aspects of interprocedural techniques: Frameworks (*how to perform these analyses*), semantic problems (*what information to propagate*), and the representation of problem solutions, which is critical because the amount of gathered information is much larger than with intraprocedural analyses.

Today, many commercial compilers include some interprocedural optimizations. In 1991, the CONVEX Application Compiler was the first on the market [58]; it includes several complex analyses, such as interprocedural pointer tracking or array section analysis. The acknowledged origins for this compiler have to be found in several research projects: PTRAN [72], ParaScope [20], Parafrase-2 [68], and PAT [74]. Other commercial products, such as the product line developed by Applied Parallel Research, the `xlf` compiler from IBM (option `-qipa`), or the SGI, FORESYS, KAP, and VAST compilers, have since included various interprocedural analyses.

But most interprocedural techniques are still in the research domain. Several advanced analyses are too costly to be integrated in commercial products. Also, new language dialects such as HPF are emerging, and with them, new interprocedural semantic problems. Finally, as problems are being solved, new applications are considered.

This paper is organized as follows. In Section 1, we survey the main approaches proposed to solve problems due to procedure calls. We emphasize the relationship between the interprocedural framework and the characteristics of the semantic problem. This study is restricted to the analysis of Fortran-like programs, but we also cite papers about other languages when the presented techniques can readily be applied. As an example, we use in Section 2 the interprocedural framework of PIPS, the open compiler workbench developed at École des mines de Paris. We first explain how interprocedural analyses are performed, and how they are supported by an interprocedural engine, PipsMake. The main analyses used for program parallelization or re-engineering are then described, as well as recent and planned developments. Finally, PIPS is evaluated along with similar research tools.

1 A Survey of Interprocedural Techniques

When dealing with a semantic problem whose solutions involve information from several procedures, two decisions have to be made. The first one is the choice of the interprocedural technique: Inlining or interprocedural analysis or a mix of them (Section 1.1). Secondly, an interprocedural framework must be chosen (Section 1.3), depending on some of the characteristics of the semantic problem (Section 1.2).

Many semantic problems benefit from interprocedural techniques. The most important ones are described in Section 1.4.

1.1 Inlining? Interprocedural Analysis? Or Cloning?

Since procedure boundaries are handled very conservatively by traditional compilers, the first and most intuitive approach to handle procedure calls is to expand them *inline*, that is to replace each call by the corresponding code, in a recursive top-down or bottom-up manner. The main advantage is that usual intraprocedural analyses and transformations can be applied on the resulting code, hopefully with the same precision. However, there are several drawbacks [41, 61]:

1. Inlining is not always possible, due to array reshaping[1], or recursive procedures in languages other than FORTRAN 77.

2. The resulting code is usually much larger than the original[2]. If a procedure `a` calls procedure `b` t_{ab} times, and procedure `b` calls procedure `c` t_{bc} times, then `c` is expanded $t_{ab} \times t_{bc}$. The growth of the code is thus potentially exponential, and this increases the complexity of subsequent analyses.

3. Even when the resulting code is not overwhelmingly larger, the complexity of subsequent analyses may grow dramatically because their complexity is higher than $O(n)$. In an automatic parallelizer for instance, if a loop contains n_1 references to array A, and a procedure which contains n_2 such references is called within, then $n_1 \times n_2$ additional dependence tests are necessary to analyze the loop.

4. And finally, the translation mechanisms may reduce the efficiency of subsequent intraprocedural analyses, for example by generating nonlinear terms in array subscript expressions [16].

Partial inlining is more suitable to actually optimize programs. User-controlled inlining is usually available in compilers. Another solution is to let the compiler automatically inline some functions or some call sites, according to a heuristic. The most advanced compilers seem to be the CONVEX Application Compiler and the IBM `xlf` compiler. However, the best heuristics are usually based on results from previous interprocedural phases [22].

[1] This occurs when an array is not similarly declared in the caller and the callee.

[2] Before any subsequent optimization such as dead code elimination.

A second approach[3], is to compute *summaries* of the procedure behaviors, to translate them as accurately as possible into the callers' or callees' name space depending on the direction of the analysis, and to use the translated summaries as abstractions of procedures. This method has the opposite advantages and consequences of inlining. It does not have its complexity limitations, since no code is duplicated, and since summaries are usually more compact than the code they abstract. Basically, if procedure sizes are bounded, and if summaries have a constant or bounded size, interprocedural analyses are linear versus the sizes of the programs. Moreover, there is no restriction on the input code (see Item 1 on Page 4). However, much precision may be lost, due to (1) summarization techniques, (2) translation processes, and (3) simplification of the program structure representation.

Between these two approaches, a recent technique lessens the drawbacks of interprocedural analysis, but preserves a reasonable complexity. *Selective cloning* [41, 27] duplicates procedures on the basis of calling *contexts*. For example, a heuristic based on integer variable values has proved useful in an automatic parallelizer, without increasing the program size dramatically [41]. However, even if this approach alleviates some precision problems (see Section 1.3), its decision heuristics are usually based on another interprocedural information, and subsequent program analyses and transformations may require an interprocedural propagation of information.

It would seem obvious that the best solution certainly is a mix of these techniques. In the CONVEX Application Compiler, interprocedural analysis results are first used to decide to inline some functions, and then to duplicate some others. But whatever solution is chosen, apart from an impractical systematic inlining, some interprocedural analyses are necessary. The main existing techniques are surveyed in the next sections.

[3] This second approach is less widely spread at least in commercial compilers

1.2 Characteristics of Interprocedural Problems

Some interprocedural problems are intrinsically much more difficult to solve than other ones. For instance it is easier to precisely compute the set of variables that *may* be modified by a procedure call, than the set of variables that are *always* modified [64] because in the last case the internal control flow of the called procedures must be taken into account.

Two main criteria have been proposed to classify interprocedural problems: Flow sensitivity and context sensitivity (also called *path specificity*) [8, 64, 60].

An interprocedural problem is said to be *flow sensitive* if it cannot be solved precisely without considering the internal control flows of procedures. The above mentioned *may be modified* problem is flow insensitive, while the *must be modified* problem is flow sensitive. For example, in the procedure `bar` in Figure 1, `j` and `k` both belong to the *may be modified* variable set because there exists at least one instruction which modifies them. But only `j` belongs to the *must be modified* set because `k` is modified only along one control flow path (the *true* test branch) and not along the other one (the *false* test branch).

```
procedure bar(i,j,k)
if (i.le.3) then
   j=5
   k=6
else
   j=4
endif
end
```

Figure 1: Flow sensitivity

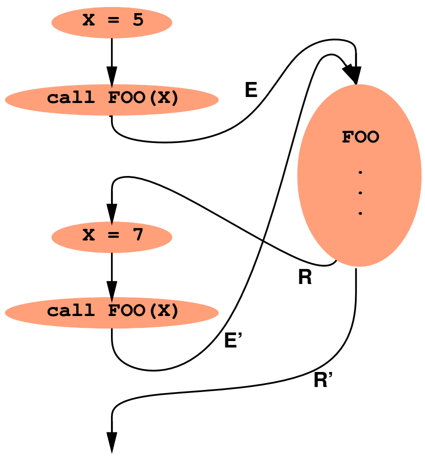

Figure 2: Unrealizable paths

Interprocedural *context sensitive* (or *path specific*) analyses are analyses whose results depend on the execution path taken to reach a point in the program. Suppose we want to compute the value of `x` after each call to `foo` in the example of Figure 2. Let us assume that `foo` increments its parameter. A context insensitive algorithm does not distinguish the two paths E-R and E'-R' through `foo`, and considers that the value of `x` before the calls is *either 5 or 7* because of arcs `E` and `E'`, and therefore that it is *either 6 or 8* after *each* call. In fact, this algorithm takes into account two *unrealizable paths*: One from the first call to the return node for the second call (`E-R'`), and another one from the second call to the return node for the first call (`E'-R`). However there are only two possible paths, namely `E-R` and `E'-R'`.

Of course, these two criteria must be taken into account when chosing or implementing an interprocedural analysis framework.

1.3 Interprocedural Analysis Frameworks

Many interprocedural analysis frameworks have been defined (some examples can be found in [64, 21, 46, 79]). They differ in several ways: The degree of summarization and the accuracy of the translation across procedure calls are two important issues, and directly depend on the specific semantic problem and the representation of its solutions. But the representation of the program structure on which the information is conveyed and the ability to perform or not flow sensitive or context sensitive analyses are also of importance. These last two problems are related [48, 46], as shown in this section.

Interprocedural analyses can be performed on the program *call graph*, where nodes represent individual procedures, and edges represent call sites[4]. Each edge is labeled with the actual parameters associated to the corresponding call site.

[4] Mathematically, this is a *multigraph*, since two nodes can be connected by several edges.

Call graphs can be constructed very efficiently [71, 45], and can provide sufficient information for many program optimizations, including parallelization. However, they do not allow flow sensitive interprocedural analyses, because they do not take into account the intraprocedural control flow of individual procedures.

To perform flow sensitive analyses, such as array region analyses[5], interprocedural representations of the program must therefore provide information about the internal control flow of procedures, as well as the external control flow. The most precise approach is to take into account the whole internal control flow graph of every procedure. However, resulting data flow analyses may be as complex as on a fully inlined program.

To reduce this complexity, one solution is to use a less precise abstraction of the program structure, in order to reduce its size. This can be done either at the interprocedural level, or even at the intraprocedural level. In this last case, the size of the representation of each procedure is reduced. Several sparse interprocedural representations have been designed for particular classes of problems, such as the *program summary graph* [19], the *system dependence graph* [34, 10], ... Intraprocedural sparse representations include the SSA form [3, 32, 76] and the *sparse data flow evaluation graph* [24]. The *unified interprocedural graph* [48] provides a demand-driven unified representation which combines the advantages of the sparse representations without restricting the scope of the possible analyses. The rationale of these approaches is to avoid computing irrelevant intermediate results, while still performing a global analysis.

Reducing the cost of interprocedural data flow analyses can also be achieved by *demand-driven* techniques, such as those presented in [33] and [69]. *Incremental* analysis [15, 59, 75] addresses the problem of avoiding unnecessary recomputations of data flow solutions in case of local program changes; but it requires an initial exhaustive solution. Another solution to reduce the cost of interprocedural analyses is to perform

[5] Array region analyses collect information about the way array elements are used and defined by the program.

them in *parallel* [39, 56].

When the representation of the program interprocedural control flow is approximative, and when the problem is context sensitive, the ability to avoid taking into account *unrealizable* paths [55, 70] is an issue. Several solutions have been proposed for this problem, the most common approach being to tag solutions with path history or path specific information [64, 73, 55]. However, this may result in an exponential growth of the number of solutions at each node, and thus in overwhelming execution times and memory requirements. Selective cloning [41, 27] also appears as a partial solution, since it reduces the number of unrealizable paths taken into account. From our experience with the PIPS parallelizer, combining results from several analyses can also lessen the problem. This is the case with *preconditions* and *transformers* in PIPS (see Section 2.3). They correctly handle the case of Figure 2 although neither analysis is context sensitive.

Many other problems may also be addressed by interprocedural tools, but are not developed here: Recursivity, formal procedures[6], or unavailable source code for called procedures.

1.4 Some Usual Interprocedural Problems

For the last two decades, the interprocedural analysis of scientific programs has been chiefly driven by research on automatic parallelization, parallelizing large loops containing procedure calls being of the utmost importance to achieve good performances.

At first, the main purpose was therefore to analyze interprocedural dependences. And alias analysis [16, 29, 72, 62] as well as *summary data flow information* or *interprocedural side-effects* (SDFI) [28] on scalar variables were among the main analyses. To enhance the results of these analyses, and to enable other useful code transformations

[6]Recursivity and formal procedures are not often used in scientific programs.

such as partial redundancy elimination [63], many other interprocedural scalar analyses have also been introduced. They range from constant propagation [18, 36, 10, 40, 23], to subexpression availability and variable values [47], ranges [13], or preconditions [53, 52] propagation. To handle arrays more accurately than SDFI, flow insensitive array region analysis was introduced by TRIOLET [77], followed by many others [21, 7, 50, 57].

Today, many commercial products include some interprocedural flow insensitive analyses, as complex as array region analyses, or pointer tracking in the most advanced tools such as the CONVEX Application Compiler. But research prototypes are still ahead, in particular for symbolic analyses [40, 54] and flow sensitive array region analyses [37, 42, 78, 31] which are mainly used for array privatization in parallelizing tools. Also, the compilation of FORTRAN dialects such as HPF raises new interprocedural problems [1, 44, 9].

1.5 Conclusion

A wide variety of interprocedural frameworks exists. They are more or less adapted to a specific interprocedural problem. The choice mainly depends on the characteristics of the problem, such as flow sensitivity or context sensitivity, but also on the desired complexity/precision ratio, given that the quality of the underlying intraprocedural analysis sets an upper bound on the precision of the interprocedural analysis.

Many flow sensitive analyses are still too complex for commercial products, which only implement flow-insensitive interprocedural analyses. But as good experimental results are published [43, 11], and as the power and the memory sizes of computers increase, such analyses will undoubtedly appear in commercial tools. Not long ago, even flow insensitive array region analyses were considered too time and space consuming. They are now implemented in all leading research parallelizer projects, and can be found in some commercial compilers [58].

2 An Example: The PIPS Compiler Workbench

It is more and more widely acknowledged that not only program optimizations for parallel and sequential target machines, but also program maintenance and reverse-engineering benefit from interprocedural techniques. To experiment various applications without having to build a new tool each time, a common infrastructure is necessary. PIPS is such a source-to-source FORTRAN open compiler workbench. The project began in 1988, with the generation of code for parallel vector computers with shared memory. Its infrastructure has since proven good enough not to require any major change. Today, its main use is code generation for distributed memory machines. Another current research track is the restructuring and reverse-engineering of production codes.

In this section, we focus on PIPS interprocedural characteristics. In Sections 2.1 and 2.2, we describe its interprocedural infrastructure. We then present in Section 2.3 the main interprocedural analyses available in PIPS. A fourth section deals with the recent and planned developments. Finally, the fifth section is devoted to an informal comparison with similar interprocedural tools.

2.1 PIPS Interprocedural Framework

PIPS uses an *implicit* call graph. It allows flow and context sensitive analyses, since the program representation contains the individual control flow graphs of all the procedures. Summarization is used to keep a linear complexity for all interprocedural analyses when the program size increases. This is achieved by eliminating control information, and by avoiding list data structures whose size could increase with the procedure height in the call graph. Translation across procedure boundaries is performed at each call site using the correspondences between formal and actual parameters, and between `common` declarations[7].

For *downward analyses* (see Figure 3), the callers are analyzed first; the information at each call site is propagated to the called subroutine to form a summary; when there are several call sites for a single procedure, the summaries corresponding to the call sites are merged, after being translated into the callee's name space, to form a unique summary. This summary is then used to perform the intraprocedural analysis of the called procedure.

For *upward analyses* (see Figure 4), the leaves of the call tree are analyzed first. The procedure summaries are then used at each call site, after the translation process, during the intraprocedural analysis of the callers.

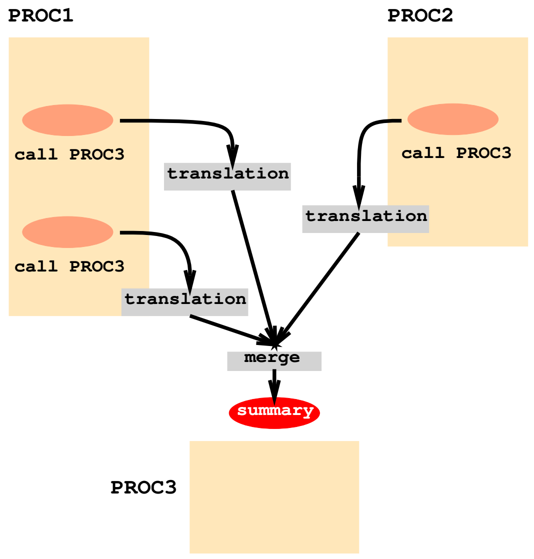

Figure 3: Downward propagation

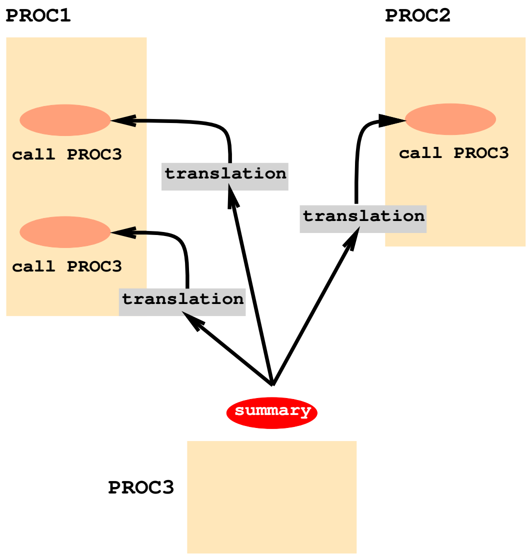

Figure 4: Upward propagation

[7]When the translation is not possible (because the variable is declared in a `SAVE` statement, or because it belongs to a `COMMON` not declared in the caller) a unique global identifier is used.

2.2 PIPS Interprocedural Engine

With PIPS infrastructure, the programmer does not have to worry about the orchestration of various interdependent analyses (or *phases*), about the call graph traversal order, and about the availability of other analysis results (or *resources*). A `make`-like mechanism, called PipsMake, automatically determines the order in which the different phases must be scheduled for the different procedures to have the resources ready when needed initially or after some minor changes of sources[17] or analysis options. The ordering is automatically deduced from a specification file, written by the phase designers, using a call-by-need interpretation. For instance, the following production rule for array regions:

```
regions   > MODULE.regions
          < PROGRAM.entities
          < MODULE.transformers
          < MODULE.preconditions
          < CALLEES.summary_regions
```

specifies that the production of the resource `regions` for the current subroutine (or *module*) is performed by the function `regions`, and that it requires the program symbol table (*entities*), several other pieces of information about the current module (*transformers* and *preconditions*), and the *summary regions* of all the callees. This rule therefore describes a backward propagation. Similarly, a forward analysis would be described by a production rule depending on information from the callers.

The programmer of the new phase must solely provide the function `regions` which performs the intraprocedural propagation of the resource and translates the summary regions of the callees into the current module name space. PipsMake automatically calls the function `regions` after having computed the other necessary resources, or after having checked that they are available and up-to-date.

This allows a quick prototyping of both forward and backward analyses on the program call graph, provided that the program is non-recursive. In addition, the simplicity of the scheme from the designer point of view promotes reuse of analysis results. Another advantage of this architecture is that it is demand-driven at the procedure level: Resources are computed or re-computed only once and only when requested by other analyses or transformations. This simple scheme based on summaries provides the flexibility and speed necessary for interactivity when dealing with source changes and new option selections because each intraprocedural analysis is performed fast enough. A lower level of granularity for incremental recompilation would not improve the response time in a useful way.

2.3 PIPS Interprocedural Analyses

Since interproceduralityity was at the origin of the PIPS project, all its analyses are interprocedural but the dependence analysis[8]. This section provides a description of the analyses that have proven to be useful for program transformations such as the parallelization of FORTRAN codes or the understanding of program behavior, which is a crucial issue for the maintenance of application codes. Effects, transformers, preconditions and array region analyses are described below. No alias analysis is performed, because the FORTRAN 77 standard forbids interprocedural aliasing of modified variables (see [5], paragraph 15.9.3.6).

Memory Effects

This abstraction describes the memory operations performed by statements or procedures. An *effect* on a memory reference is a *read* or *write* effect. PIPS also provides information about the likeliness of the reference: *must* effects are always performed by

[8] PIPS can automatically parallelize loop nests with procedure calls, but the interprocedural part is carried out by effect or region analysis.

all executions, while *may* effects may be performed by some execution, or even never. Procedure summaries are obtained by eliminating effects on local variables. These summaries are then translated in the caller's name space for each call site, describing the effects of each call statement.

This information can be used to parallelize loops containing procedure calls thanks to the translation process. Effects are also used to support other analyses, such as *preconditions*, to avoid propagating information about a variable if it is neither used nor modified in a subroutine and its descendents.

Transformers

Summary effects are the sets of variables read or written by procedures. But they do not describe *how* these variables are modified. In PIPS, this is the role of *transformers*. Transformers abstract the effects of instructions and procedure calls upon the values of integer scalar variables by affine approximations of the relations that exist between their values before and after any execution of a statement or procedure call. For example, the dummy procedure `foo`:

```
subroutine foo(i,j)
if (i.le.10) then
  i = 10
else
  i = i+1
endif
j = j+1
end
```

is abstracted by the transformer `T(i,j)` {`10<=i, i#init<=i, j==j#init+1`}. It means that the value of `i` after any call to `foo` is greater than 10 and greater than its value before any call, and that the value of `j` is incremented by 1.

Real, logical and string variables are not taken into account because the control flow at the loop level is mainly derived from integer values. Moreover, dusty decks and validation programs provided by users do not often use logical and character strings, and deriving information about floating point values is very difficult.

Transformer analysis is a backward analysis, and its results are mainly used by two other types of analyses: *preconditions* and *array region analyses*. Transformers are called *return functions* in [18].

Preconditions

Many program analyses and transformations can be improved when the relations between variable values are known. This is the case for loop parallelization which benefits from the knowledge of loop bound values, as well as array subscript symbolic constants. Dead code elimination and code specialization are also greatly enhanced by this type of information.

In PIPS *precondition* analysis labels each statement with a condition on the variable current values holding true before any of its executions. This condition involves equality constraints as well as inequalities. Therefore, values of symbolic constants and inequalities from test conditions are both preserved. This type of analysis greatly benefits from interprocedural techniques. In many programs, many variables used either in loop bounds, in tests conditions or in array subscripts, are initialized in a separate procedure. Tests also often restrict the conditions under which procedures may be called.

Precondition analysis is a forward analysis, and relies on transformers to derive post-conditions for statement or procedure calls from their preconditions. The information available in initialization routines is moved upwards to the main procedure using transformers, and propagated downwards towards the leaf procedures by preconditions. The whole analysis is thus completed in two traversals of the call graph.

Summary preconditions are similar to *jump functions* [18].

Array Region Analyses

Effects often fail to give precise information about the way array element sets are used and defined by procedures. And this lack of accuracy often prevents subsequent program transformations such as loop parallelization.

To obtain better accuracy, and to enable advanced program transformations such as array privatization, several flow sensitive *array region* analyses have been implemented in PIPS [31, 30]. READ and WRITE regions represent the effects of statements and procedures on array elements as sets. They are mainly used by the dependence analysis phase, but also by HPFC, the PIPS HPF compiler [25]. In addition, two other types of regions have recently been introduced: IN regions contain the array elements *imported* by the current statement[9], while OUT regions contain the elements *exported* towards the program continuation[10].

IN and OUT array regions ease the program comprehension by hiding internal details of procedures and composite statements (such as loops or tests), and by providing more relevant information to the user than READ and WRITE regions. They are also used to privatize array regions in loops [30]. New types of program transformations based on IN and OUT regions are currently being implemented, such as the re-engineering of procedure declarations. This transformation spots global arrays used as temporary variables in procedures and reallocates them as stack-allocated local variables. It is useful for the maintenance and parallelization of dusty deck programs.

[9] IN regions are similar to upward exposed read regions computed by others [72, 42, 37, 78].

[10] OUT regions are different from downward exposed regions, because they also depend on the future of the computation.

2.4 Recent and Planned Developments

We have only described some of the many analyses available in PIPS. Other analyses include interprocedural symbolic complexity and reduction detection, or intraprocedural dependence and array data flow analyses [67]. PIPS also includes an impressive set of intraprocedural program transformations, including source code generation, which heavily rely on the previously described interprocedural analyses.

The latest developments essentially focus on these program transformations: Array privatization for loops and procedures (see Section 2.3); or partial evaluation, dead code elimination [6] and code restructuring based on preconditions or use-def chains, for example. The recursive application of transformations and their interaction with the analyses on which they rely, creates a new problem in an interprocedural setup.

For example, restructuring a monoprocedural program A using its preconditions can result in more accurate preconditions, which can themselves be used to restructure A, and so on. This is one of the simplest schemes. Suppose now that A is called by B. Restructuring A modifies its code, and thus invalidates its transformers, as well as the transformers of B, since transformer analysis is a backward analysis. This may result in more precise preconditions in B, and therefore in more precise preconditions in A, which can be now be used to restructure A, and so on.

Handling such problems involves an extension of PipsMake to deal with the fix points of analysis and transformation phases.

Also, recent developments in the array region area have shown that representation independence could be improved [30]. For instance, different array region representations such as intervals, convex polyhedra, list of convex polyhedra or Presburger formulae, have been made usable by a generic interprocedural array region engine. Although the saving expressed in lines of code is limited because the programming effort is more at the operator level than at the engine level, and because operators obviously depend on the representation, this is very useful for experiments. This flexibility could be used in commercial products to let the user make speed/accuracy tradeoffs. However, the dependence between different kinds of analyses makes such clear interface non trivial. For instance, region analysis uses precondition analysis. The combination of a polyhedral representation for regions and of a Presburger-based representation for preconditions, as well as all three other different possible combinations are not obvious and would require additional conversion operators.

Representation independence could be used to move PIPS away from its original linear algebra framework and into polynomial representations as in Parafrase-2 (see next section) or into Presburger-based representations. Such a move has been shown necessary in [14], but we feel that linear algebra and polyhedra are an interesting tradeoff in the accuracy/speed space and that their potential has not yet been measured.

2.5 PIPS and Other Research Tools

Many research Fortran optimizing tools include some kind of interprocedural analyses, but the closest ones to PIPS certainly are FIAT/SUIF [46, 47], ParaScope [26] and the D system [44], Polaris [11], Parafrase-2 [68], and Panorama [65, 37, 38].

FIAT/SUIF

SUIF is an intraprocedural compiler for parallel machines developed at Stanford. To enable the parallelization of loops containing procedure calls, FIAT [46], an interprocedural engine, has been added on top of it. FIAT is a framework which allows prototyping of flow insensitive and flow sensitive interprocedural data flow analyses. The programmer has only to provide the initialization functions, the meet operator, the transfer function, and the direction of the analysis. Much like PipsMake, FIAT is demand-driven: The programmer does not have to worry about the ordering of interprocedural analyses.

Several interprocedural analyses are available in FIAT/SUIF. The first backward analysis computes a transfer function which bears some resemblance with PIPS transformers, but in its handling of test conditions. Then some kind of preconditions are computed, which give each variable value in terms of loop invariants and indices of enclosing loops. An additional phase propagates inequality constraints from test conditions. Several backward interprocedural array regions analyses are finally performed.

The support of automatic selective procedure cloning and, hence, of some context sensitivity is a distinctive advantage of this framework. Unfortunately, FIAT/SUIF had not yet been released in the public domain for trial when we wrote this paper.

Parascope and the D System

Parascope and the D system are being both developed at Rice University. They use FIAT as an interprocedural engine, like FIAT/SUIF.

Parascope provides several interprocedural analyses: MOD/REF analysis (i.e. effects), array region analyses based on *regular section descriptors (RSD)* [21], alias analysis, constant propagation and symbolic value analysis.

The D system is built in the context of Parascope, and aims at compiling Fortran D programs. Fortran D is an HPF-like dialect, and requires specific interprocedural analyses, such as *reaching decomposition* or *overlap* analyses.

Parafrase-2

Parafrase-2 is an optimizing and parallelizing source-to-source compiler developed at the University of Illinois at Urbana-Champaign. It includes many phases, some of which are interprocedural, especially a powerful symbolic analysis based on polynomials [40].

Polaris

Polaris too is being developed at the University of Illinois. Most Polaris phases are intraprocedural and require inline expansion to parallelize loops efficiently. However, automatic inlining is provided, and some interprocedural phases, such as constant propagation, have been implemented lately or are planned, such as array region analyses based on lists of *regular section descriptors (RSDs)*.

Panorama

The Panorama parallelizing compiler has been initiated at the University of Minnesota, to support systems with memory hierarchies. It performs several interprocedural analyses, such as use-def chains [49] and flow-sensitive array region analysis based on lists of *guarded regular section descriptors (gRSDs)*. Interprocedural analyses are performed on the program *hierarchical supergraph*, which is an extension of MYERS' *supergraph*. As such, it provides the necessary framework to perform flow and context sensitive analyses. But no interprocedural engine such as FIAT or PipsMake is used, and each analysis must redefine its own traversal of the hierarchical supergraph.

Relationships with PIPS

It is impossible to compare different interprocedural tools in a scientific way. They do not provide exactly the same sets of analyses. And when two tools support the same interprocedural analysis, its precision depends on several factors, such as the quality of preliminary analyses, the quality of the intraprocedural propagation, the characteristics of the chosen representation (whether it can include symbolic context information or not, for example), and the precision of the translations across procedure boundaries.

In this context, PIPS interprocedural framework is unique compared to the other approaches. Firstly, it provides an interprocedural demand-driven engine which also ensures the consistency of the database for a given option selection and source editing. This feature is unique among interprocedural tools because PIPS was designed primarily to study interprocedural issues. Along with the program representation as a call graph, this engine allows flow sensitive analyses. Secondly, PIPS offers a comprehensive set of interrelated interprocedural analyses (*effects*, *transformers*, *preconditions* and *array regions*) and of transformations exploiting the symbolic information gathered, e.g. partial evaluation.

These features do not come at the expense of execution time and PIPS can be used interactively as well as in batch mode. As long as no code transformation is applied, each analysis is applied only once on each module. At a given average module size, summarization results in a linear complexity against the program size for all interprocedural analyses.

Apart from its interprocedural features, PIPS offers a number of advantages. It was designed as an open workbench to support student contributions and distributed development. Extensions can be developed without linking with PIPS source because every useful data structure can be stored on disk and retrieved by another program. And PIPS is a source-to-source tool and great care was taken to produce user-recognizable source code. The control structure of the code and comments are preserved as long as possible.

However, the use of summarization, to keep summaries small, combined with a pure interprocedural framework without cloning often prevents interprocedural path differentiation, whereas other parallelizers [35, 42] partially achieve this goal through automatic selective procedure cloning. Since PIPS is an interactive tool, manual cloning was tried and more accurate results were obtained, especially for forward interprocedural analyses, such as preconditions and OUT regions, and for intraprocedural transformations. Since PIPS does not use an *explicit* interprocedural control flow graph, cloning

Features	FIAT SUIF	Parascope D-System	Parafrase-2	Panorama	PIPS hpfc	Polaris
Cloning	X					X
Inlining			X			X
Alias Analysis		X	X		X	
Constants	X	X	X		X	X
Inequalities	X		X		X	
Polynomials	X		X			
Effects (SDFI)		X			X	
RSD		X				
RSDs	X					X
gRSDs				X		
Polyhedral					X	
Use-Def Chains				X	X	
Reaching Decomp.		X			X	
Overlap					X	
IO's		X			X	
Engine	FIAT	FIAT			PipsMake	

Figure 5: Features of some academic projects

and selective cloning would be easy to implement.

Features described in references and WEB pages about the projects presented above are shown in Figure 2.5. They are informally grouped into categories, some related to the environment, the store and the command domains of denotational semantics. This summary is not provided as a definite comparison between different academic projects which are still evolving, but as a set of pointers for readers interested in a given feature. Obviously we take responsibility for any mistake or missing information.

3 Conclusion

Interprocedural analyses have come of age. Thanks to CPU and memory technology improvements, futuristic approaches of the eighties are now implemented in commercial compilers. Time and space complexities of such analyses are still issues, but they are not perceived any longer as impossible to tackle.

Most research compiler prototypes take interprocedural issues into account and some classification effort has now been undertaken to better understand the relationship between the many interprocedural analyses published since 1974 [60]. But this effort must be pursued, in particular for interdependent analyses which may override the existing classification.

The current interprocedural frameworks for scientific programming in Fortran are surveyed in this paper. Few experimental results about effectiveness [51, 43, 11] have been published and additional research in the interprocedural analysis area is necessary before a really formal comparison can be made. In particular, the speed/accuracy ratio is still an issue, but has not yet been extensively studied [38]. For that purpose, more generic tools for the experimentation of interprocedural analyses, such as FIAT [46], PAG [4], or PIPS, should be made available.

Many issues linked to interprocedural analyses have been dealt with by PIPS, the interprocedural Fortran 77 source-to-source parallelizer developed at École des mines. It supports a comprehensive set of interprocedural analyses, each with a large number of accuracy options. Its interprocedural engine, PipsMake, is a unique feature among the other interprocedural tools briefly surveyed in this paper: FIAT/SUIF, Parascope and the D System, Parafrase-2, Polaris and Panorama. PIPS is a public-domain tool available for SunOS, Solaris, AIX, LINUX and OSF-1 at URL `http://www.cri.ensmp.fr/pips` designed to explore interprocedural techniques for Fortran programs. Future developments include array privatization at the procedure level, extension of PipsMake to deal efficiently with fixpoints of analysis and transformation phases, and speed/accuracy comparison of various array region analyses based on different representations.

References

[1] G. Agrawal and J. Saltz. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *International Conference on Programming Language Design and Implementation*, pages 258–269, June 1995.

[2] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress*, pages 398–402, 1974.

[3] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[4] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, Lecture Notes in Computer Science, pages 33–50. Springer-Verlag, 1995.

[5] American National Standard Institute. *Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980*, 1983.

[6] C. Ancourt, F. Coelho, B. Creusillet, and R. Keryell. How to add a new phase in PIPS: The case of dead code elimination. In *Sixth International Workshop on Compilers for Parallel Computers*, pages 19–30, December 1996.

[7] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *International Conference on Programming Language Design and Implementation*, pages 41–53, June 1989.

[8] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Symposium on Principles of Programming Languages*, January 1979.

[9] S. Benkner and H.P. Brezany, P.and Zima. Processing array statements and procedure interfaces in the PREPARE High Performance Fortran compiler. In *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 324–338, April 1994.

[10] D. Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *International Conference on Compiler Construction*, pages 374–388, April 1994.

[11] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *Computer*, 29(12):78–82, December 1996.

[12] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[13] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *International Conference on Supercomputing*, pages 528–537, November 1994.

[14] W. Blume and R. Eigenmann. Symbolic analysis techniques needed for the effective parallelization of the Perfect Benchmarks. In *International Conference on Parallel Processing*, 1994.

[15] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[16] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 162–175, July 1986.

[17] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[18] C. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 152–161, 1986.

[19] D. Callahan. The program summary graph and flow-sensitive interprocedural data-flow analysis. *International Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, 23(7):47–56, July 1988.

[20] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4), Winter 1988.

[21] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(10):517–550, 1988.

[22] P. Carini. Automatic inlining. Research Report RC 10286, IBM, November 1995.

[23] P. Carini. Flow-sensitive interprocedural constant propagation. Research report RC 20290, IBM, November 1995.

[24] J. Choi, R. Cytron, and J Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[25] F. Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, pages 102–109, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.

[26] K. Cooper, M. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Waren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 81(2), February 1993.

[27] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *IEEE International Conference on Computer Language*, April 1992.

[28] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984.

[29] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Symposium on Principles of Programming Languages*, pages 49–59, February 1989.

[30] B. Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des mines de Paris, December 1996. Available at `http://www.cri.ensmp.fr/doc/A-295.ps.gz`.

[31] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996.

[32] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignement form and the control dependence graph. In *Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[33] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data-flow. In *Symposium on Principles of Programming Languages*, pages 37–48, January 1995.

[34] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[35] S. Graham, S. Lucco, and O. Sharp. Orchestrating interactions among parallel computations. In *International Conference on Programming Language Design and Implementation*, pages 100–111, June 1993. ACM SIGPLAN Notices.

[36] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump functions implementations. In *International Conference on Programming Language Design and Implementation*, pages 90–99, June 1993. ACM SIGPLAN Notices.

[37] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.

[38] J. Gu, Z. Li, and G. Lee. Experience with efficient data flow analysis for array privatization. In *Symposium on Principles and Practice of Parallel Programming*, pages 157–167, June 1997.

[39] R. Gupta, L. Pollock, and M. L. Soffa. Parallelizing data flow analysis. In *Workshop on Parallel Compilation*, May 1990.

[40] M. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* Kluwer Academic Publishers, 1995.

[41] M. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, Houston, Texas, April 1991.

[42] M. Hall, S. Amarasinghe, B. Murphy, S.-W. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing*, December 1995.

[43] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.

[44] M. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D. *Journal of Parallel and Distributed Computing*, 38(2):114–129, November 1996.

[45] M. Hall and K. Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[46] M. Hall, J. Mellor-Crummey, A. Carle, and R. Rodríguez. FIAT: A framework for interprocedural analysis and transformation. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[47] M. Hall, B. Murphy, S. Amarasinghe, S.-W. Liao, and M. Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer-Verlag, August 1995.

[48] M. J. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

[49] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

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

[51] M. Hind, M. Burke, P. Carini, and S. Midkiff. Interprocedural array analysis : How much precision do we need ? In *Third Workshop on Compilers for Parallel Computers*, pages 48–64, July 1992.

[52] F. Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, pages 333–350. North-Holland, September 1992.

[53] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[54] S. Johnson, M. Cross, and M. Everett. Exploitation of symbolic information in interprocedural dependence analysis. *Parallel Computing*, 22:197–226, 1996.

[55] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *International Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[56] Y.-F. Lee, B. Ryder, and M. Fiuczynski. Region analysis: a parallel elimination method for data flow analysis. *IEEE Transactions on Software Engineering*, 21(11):913–926, November 1995.

[57] Z. Li and P.-C. Yew. Efficient interprocedural analysis and program parallelization and restructuring. In *Symposium on Parallel Processing: Experience with Applications, Languages and Systems*, pages 85–99, 1988.

[58] J. Loeliger and R. Metzger. Developing an interprocedural optimizing compiler. *ACM SIGPLAN Notices*, 29(4):41–48, April 1994.

[59] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[60] T. Marlowe, B. Ryder, and M. Burke. Defining flow sensitivity in data flow problems. Research Technical Report lcsr-tr-249, Rutgers University, Laboratory of Computer Science, July 1995.

[61] D. Maydan. *Accurate Analysis of Array References.* PhD thesis, Stanford University, September 1992.

[62] H. Mayer and M. Wolfe. Interprocedural alias analysis: Implementation and empirical results. *Software : Practice and Experience*, 23(11):1201–1233, November 1993.

[63] E. Morel and C. Renvoise. Interprocedural elimination of partial redundancies. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*. Prentice-Hall, Inc., 1981.

[64] E. Myers. A precise inter-procedural data-flow algorithm. In *Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[65] T. Nguyen, J. Gu, and Z. Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 96–110. Springer-Verlag, August 1995.

[66] P. M. Petersen. *Evaluation of Programs and Parallelizing Compilers using Dynamic Analysis Techniques*. PhD thesis, CSRD, University of Illinois, January 1993. CSRD report No. 1273.

[67] A. Platonoff. Automatic data distribution for massively parallel computers. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 555–570, June 1995.

[68] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitionning, synchronizing and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, August 1989.

[69] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction*, pages 389–403, April 1994.

[70] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[71] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, May 1979.

[72] V. Sarkar. PTRAN — the IBM parallel translation system. In *Parallel Functional Programming Languages and Compilers*, pages 309–391. ACM Press Frontier Series, 1991.

[73] M. Sharir and A. Pnuelli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 189–233. Prentice-Hall, Inc., 1981.

[74] K. Smith and W. Appelbe. PAT - an interactive Fortran parallelizing assistant tool. In *International Conference on Parallel Processing*, volume 2, pages 58–62, 1988.

[75] V. Sreedhar, G. Gao, and Y.-F. Lee. A new framework for exhaustive and incremental analysis using DJ graphs. In *International Conference on Programming Language Design and Implementation*, pages 278–290, May 1996.

[76] E. Stoltz, M. Gelerk, and M. Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *International Conference on System Sciences*, pages 43–52, 1994.

[77] R. Triolet, P. Feautrier, and F. Irigoin. Direct parallelization of call statements. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, July 1986.

[78] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[79] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Symposium on Principles of Programming Languages*, pages 246–259, January 1993.