PIPS: a Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers

Technical paper

Ronan KERYELL (presenting author)
Corinne ANCOURT Fabien COELHO
Béatrice CREUSILLET François IRIGOIN
Pierre JOUVELOT

Centre de Recherche en Informatique,
École Nationale Supérieure des Mines de Paris,
35, rue Saint-Honoré,
F-77305 FONTAINEBLEAU CEDEX FRANCE.
{ancourt,coelho,creusillet,irigoin,jouvelot,keryell}@cri.ensmp.fr

May 14, 1996

Abstract

PIPS is an experimental tool to implement and evaluate various interprocedural compilation, parallelization, analysis and optimization techniques. This paper focuses on the workbench used to build these compilers. It includes the management of data structures, of dependences between the various analysis and transformation phases, and the mathematical libraries used to implement some of the most advanced analyses.

Keywords: Parallelizing environment, programming environment, workbench, make, interprocedural parallelizer, interactive parallelizer, optimizer, reverse engineering, CASE.

1 Introduction

Detecting the maximum level of parallelism in sequential programs has been a major source of interest during the last decade. This process is of utmost importance when trying to cope with the increasing number of vector and parallel supercomputers and, perhaps unfortunately, the huge number of already existing "dusty deck" sequential programs. Nowadays, since the development of high-end parallel computers has not fulfilled the wildest hopes of entrepreneurial companies in the parallel supercomputing business, we can expect more down-to-earth developments of work-group parallel servers and workstations with few but powerful processors.

Together with this architectural evolution, three main directions in compiler developments have been pursued: compilers for sequential machines (with superscalar processors), parallelizers of sequential programs and compilers for explicit parallel programs. All these approaches benefit from deep global program analyses, such as interprocedural and semantical analyses, to perform optimizations, vectorization, parallelization, transformations, restructuring and reverse-engineering of programs. Since these approaches often require the same or share a common ground of analyses, it is interesting to factorize them out in a common development tool to get the benefit of code re-use and modular programming.

PIPS is such a highly modular workbench for implementing and assessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, etc. without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988 as the PIPS project (Interprocedural Parallelization of Scientific Programs). PIPS has been developed through several research projects funded by DRET (French ARPA), CNRS (French NSF) and the European Union (ESPRIT programs). The initial design was made by Rémi TRIOLET, François IRIGOIN and Pierre JOUVELOT. It has proved good enough since then not to require any major change.

Our project aims at combining advanced interprocedural and semantical analyses [19] with a requirement for compilation speed. The mathematical foundations of the semantical analysis implemented in PIPS are linear programming and polyhedra theory. Despite such advanced techniques, PIPS is able to deal with real-life programs such as benchmarks provided by ONERA (French research institute in aeronautics), or the Perfect Club in quite reasonable time. An excerpt of a fluid dynamics code for a wing analysis (Figure 2) will be used as a running example in this paper.

PIPS is a multi-target parallelizer. It incorporates many code transformations and options to tune its features and phases. PIPS offers interesting solutions to solve programming design problems such as generating high-level data structures from specifications, co-generating their documentation, defining user interface description and configuration files, dealing with object persistence and resource dependences, being able to write parts of PIPS in various languages, etc.

This paper details how PIPS achieves these goals. We first present a general overview of PIPS in the next section. In Section 3 we describe the mechanisms that deal with PIPS data structures, their dependences and their interprocedural relationships. The general design is discussed in Section 3. The programming environment, including the data structures generator and the linear library, and the documentation process is presented in Section 4. At last, the different user interfaces in Section 5 are presented before the related work in Section 6.1.

2 Overview

PIPS is a source-to-source Fortran translator (Figure 1), Fortran still being the language of choice for most scientific applications. Beside standard Fortran 77, various dialects can be used as input (HPF) or output (CMF, Fortran 77 with Cray directives, Fortran 77 with HPF parallelism directives, etc.) to express parallelism or code/data distributions. Fortran can be seen here as a kind of

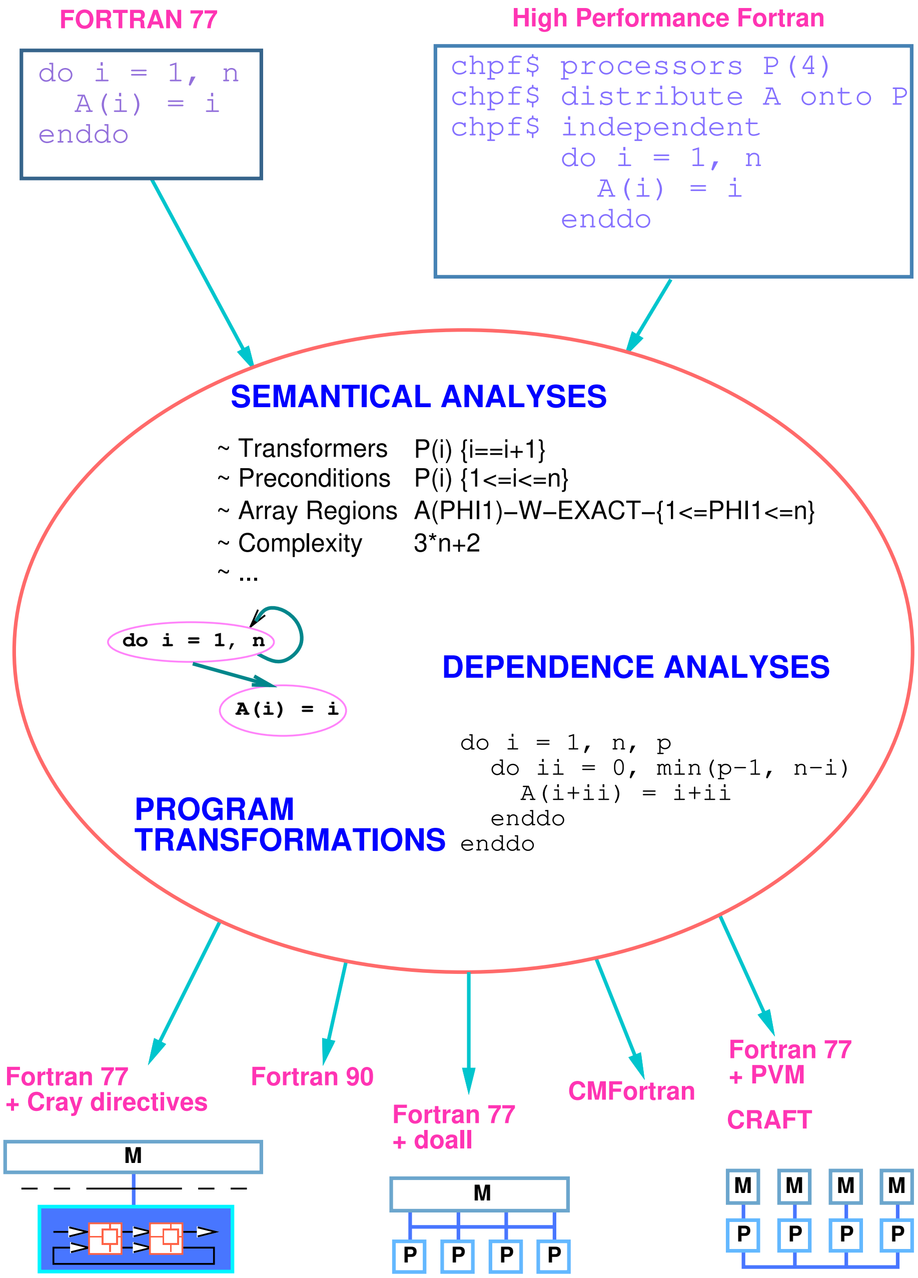

Figure 1: User view of the PIPS system.

portable language to run on various computers without having to deal with an assembly code back-end, although the PIPS infrastructure could be used as a basis to generate optimized assembly code. Other parsers or prettyprinters could be added and the internal representation could be extended if need be to cope with other imperative languages such as C.

```
c
      PROGRAM EXTRMAIN
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      DATA N1,N4,N7,N17 /1,4,7,17/

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
c
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(N7,N17)
         CALL EXTR(N4,N17)
         CALL EXTR(N1,N17)
      ENDIF
      END
c
c
c Compute the extrapolation coefficients
c for all the wing area (K=K1)
c
      SUBROUTINE EXTR(NI,NC)
c
      DIMENSION T(52,21,60)
```

```
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JM=J1+J2-J
         T(JM,1,NC+3)=T(J,1,NC+3)
         T(JM,1,NC+4)=T(J,1,NC+4)
         T(JM,1,NC+5)=T(J,1,NC+5)
 300  CONTINUE
      END
c
c
c Compute D=DISTANCE
c
      REAL FUNCTION D(J,K,JP,KP)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
c
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 2: Code excerpt from an ONERA benchmark.

Following the field research history, PIPS has first been used to improve the vectorization of Fortran code for parallel vector computers with shared memory (Fortran 77 with `DOALL`, Fortran 77 with Cray micro-tasking directives, Fortran 90). It is now mainly targeted at generating code for distributed memory machines, using different methods (processor and memory bank code for control distribution [3], CMF, CRAFT polyhedral method [17] or message-passing code from HPF [5]).

As we see in Section 3, the translation process is broken into smaller modular operations called phases in PIPS. For the user, these phases can be classified into various categories according to their usage. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are referred to as *module* in the sequel.

2.1 Analyses

An analysis phase computes some internal information that can be later used to parallelize (or generate) some code, some user information (such as a program complexity measure) to be later displayed by a prettyprinter, etc. The most interesting analyses available in PIPS are the semantical analyses described below.

2.1.1 Effects

Since an imperative language such as Fortran deals with memory locations, the basic semantical analyses in PIPS deal with the effects of instructions on memory. First, the *proper* effects are computed for each sub-statement of a module: a read or write operation on an array element, the update of an index variable in a `DO` loop, etc. These effects can be displayed to the user *via* a prettyprinter like:

```
C      <must be read    >: J NC S1 S2 S3
C      <must be written>: T(J,1,NC+5)
          T(J,1,NC+5) = S1*S2/((S3-S1)*(S3-S2))
```

Then, since the abstract syntax tree is recursively defined in PIPS, the *cumulated* effects are recursively aggregated from all the proper effects of the sub-statements in a statement. For the loop `300` on Figure 2 this leads to:

```
C      <may be read     >: J J2 JH K L NC S1 S2 S3 T(*,*,*)
C      <may be written >: JH S1 S2 S3 T(*,*,*)
C      <must be read    >: J1 JA
C      <must be written>: J
       DO 300 J = J1, JA
          ...
```

Since having information about effects on internal variables in a function is irrelevant to calling procedures, the *summary* effects of a procedure are constructed by removing these local effects from the cumulated effects of the procedure. Interprocedural propagation will be commented in Section 3.4.

2.1.2 Transformers

To improve the accuracy of the effect information, to provide more information for dependence testing and to select better program transformations, statements are labeled with preconditions that express constraints on integer scalar variables. Since such variables are often used in array references and loop bounds, preconditions can often precisely define which subpart of an array is referenced by a memory reference expression, thus refining cumulated effects into regions (Section 2.1.4).

Mathematically speaking, PIPS preconditions are computed via transformers that are abstract commands mapping a store to a set of stores [14, 13]. They are relations between the values of integer variables in an initial store and those in a final store. In PIPS, the considered relations are polyhedra, represented by systems of linear equalities and inequalities, because they provide a sound and general mathematical framework for reasoning about array accesses which are mostly linear in practice. Below, two statement transformers are given by the prettyprinter.

```
C  T(J1) {J1==J1#init+1}
          J1 = J1+1
C  T(J2) {J2==2JA+1}
          J2 = 2*JA+1
```

Constant propagation, inductive variable detection, linear equality detection or general linear constraints computation can be performed in this workbench using different operators to deal with PIPS basic control structures. To deal with unstructured control graphs, some fancier operators requiring a fix point computation are needed, although they may slow down the analysis. To provide a better trade-off between time and precision to the user, options are available to tune the precondition construction algorithms.

Unlike most analysis in the field, we are dealing with abstract commands instead of abstract stores for two reasons . First, variables appearing in references cannot be aggregated to build cumulated and summary effects unless they denote the same values, that is they refer to the same store. The module's initial store is a natural candidate to be the unique reference store and, as a result, a relationship between this store and any statement's initial store is needed. This relationship is called a precondition [13] in PIPS and is constructed for a statement by aggregating all the transformers of the statements on the paths from the beginning of the module up to that statement. On the same example than the transformers we have:

```
C  P(J1,N) {N==1, 1<=J1, ...}
         J1 = J1+1
C  P(J1,N) {N==1, 2<=J1, ...}
         J2 = 2*JA+1
C  P(J1,J2,JA,N) {J2==2JA+1, N==1, 2<=J1, ...}
```

Second, dependence tests are performed between two statements, Although a relationship between each statement store would be more useful than two stores, each relative to one statement, this would not be realistic because too many predicates would have to be computed. Thus, a common reference to the same initial store seems to be a good trade-off between accuracy and complexity.

2.1.3 Dependence test

Since the dependence graph [21] is used within many parts of PIPS, such as the parallelizers, the validity check for some transformations, the use-def elimination phase, etc., a multi-precision test has been implemented to provide various degrees of precision and speed. First, a crude by fast algorithm is used and if no positive or negative conclusion is found, a more precise but more compute-intensive test is used, and so on [20].

PIPS dependence tests compute both dependence levels and dependence cones. Dependence cones can be retrieved to implement advanced tiling and scheduling methods.

2.1.4 Regions

Array regions [9, 10] are used in PIPS to summarize accesses to array elements. READ and WRITE regions represent the effects of statements and procedures on sets of array elements. They are used by the dependence test to disprove interprocedural dependences. For instance, in our example, the READ regions for the sole statement of function D are:

```
C       <T(φ1,φ2,φ3)-R-EXACT-{φ1==J,K<=φ2<=K+1,L<=φ3<=L+2}>

             D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1     +        (T(J,K,L+1)-T(JP,KP,L+1))**2
     2     +        (T(J,K,L+2)-T(JP,KP,L+2))**2)
```

where the ϕ variables represent the three dimensions of `A`. To compute this region, interprocedural preconditions must be used to discover the relations between the values of `J` and `JP`, and `K` and `KP`.

However, since READ and WRITE regions do not represent array data flow, they are insufficient for advanced optimizations such as array privatization. IN and OUT regions have thus been introduced for that purpose: for any statement or procedure, IN regions contain its imported array elements, and OUT regions contain its exported array elements. The possible applications are numerous. Among others, IN and OUT regions are already used in PIPS to privatize array sections [4, 8], and we intend to use them for memory allocation when compiling signal processing specifications based on dynamic single assignments.

Another unique feature of PIPS array regions lies in the fact that, although they are over-approximations of the element sets actually referenced, they are flagged as exact whenever possible. Beside a theoretical interest, there are specific applications such as the compilation of HPF in PIPS.

2.1.5 Array Data Flow Graph

This sophisticated data flow graph extends the basic DFG with some kind of array regions information with linear predicates. Thus, it is able to finely track array elements accesses according to different parameters and loop indices in programs. However, this is only valid when the whole control can be represented in a linear framework [18, 17].

2.1.6 Complexities

When optimizing some code, or assessing whether to parallelize some program fragment or apply a transformation, it is quite useful to have some estimation of the program time complexity. A static complexity analysis phase has been added to PIPS to give polynomial complexity information about statements and modules [22] from preconditions and other semantical information.

2.2 Code generation phases

The result of program analyses and transformations is a new code. These phases generate (1) parallel code from data such as sequential code, semantical informations and the dependence graph, or (2) a message passing code for distributed memory architectures.

2.2.1 HPF Compiler

HPFC is a prototype HPF compiler implemented as a set of PIPS phases. From Fortran 77, HPF static (`align distribute processors template`), dynamic (`realign redistribute dynamic`) and parallelism (`independent new`) directives, it generates portable PVM-based SPMD codes which have run on various parallel architectures (SUN NOW [5], DEC alpha farm, TMC CM5, IBM SP2).

Implementing such a prototype within the PIPS framework has proven to be a fine advantage: First, the integration of the various phases (Figure 3) within the PipsMake system allows to reuse without new developments all PIPS analyses, and to benefit from the results of these analyses for better code generation and optimizations; Second, the Linear C^3 library has provided the mathematical framework [1, 2] and its implementation to develop new optimizations techniques targeted at generating efficient communication codes. It has been applied to I/O communications [6] and general remappings [7].

Phase	Function
`hpfc_filter`	preprocessing of the file
`hpfc_parser`	Fortran and HPF directive parser
`hpfc_init`	initialization of the compiler status
`hpfc_directives`	directive analysis
`hpfc_compile`	actual compilation of a module
`hpfc_close`	generation of global runtime parameters
`hpfc_install`	installation of the generated codes
`hpfc_make`	generation of executables
`hpfc_run`	execution of the program under PVM

Figure 3: Hpfc phases in Pips

2.2.2 Parallelization

The primary parallelization method used for shared-memory machines is based on Allen&Kennedy algorithm and on dependence levels. This algorithm was slightly modified to avoid loop distribution between statements using the same private variable(s). The resulting parallel code can be displayed in two different formats based on Fortran 77 and Fortran 90. Parallel loops are distinguished by the `DOALL` keyword.

Another parallelization algorithm was derived from Allen&Kennedy's to take into account the Cray specificities, the vector units and the micro-tasking library. Two levels of parallelism are selected at most. The inner parallelism is used to generate vector instructions and the outer one is used for task parallelism.

2.2.3 Code distribution

This parallelization method emulates a shared memory machine on a distributed architecture with programmable memory banks. It splits the program in two parts: a computational part and a memory part. More precisely, two programs are generated by using a method based on linear algebra: one SPMD program that does the computations in a block distributed control way and another SPMD program that manages the memory accesses.

The development run-time is implemented over PVM but the target is Inmos T9000 with C104 transputer-powered parallel machines with an hight speed interconnection network. This prototype was built as part of the European Puma project [3].

2.2.4 Parallelization with a polyhedral method

By using the Array Data Flow Graph, it is possible to track the movement of each value in a program where control flow can be statically represented in a linear way. Using this information, a schedule is generated to exploit the maximum parallelism and a placement is selected to reduce the number of communications according to their nature without discarding all the parallelism [18, 17].

2.3 Transformations

A transformation in PIPS takes an object, such as the internal representation of the code, and generates a new version of this object. Transformations currently implemented in PIPS are:

- loop distribution;
- scalar and array privatization based on regions;
- atomizer to split statements in simpler ones such as `A=B op C`;
- loop unrolling;
- loop strip-mining;
- loop interchange using an unimodular transformation;
- loop normalization;
- partial evaluation;
- dead-code elimination based on preconditions;
- use-def elimination;
- control restructurer;
- reduction detection.

2.4 Pretty-printers

Pretty-printers are used to transform some PIPS internal representation into a human readable file that can be displayed by the user interface. There are pretty-printers to display the original parsed code, the sequential code, the parallel code in some parallel dialect, the dependence graph, the call graph, etc. Other specialized pretty-printers exist to add some informative decoration to the previous ones, such as displaying the code with regions, preconditions, complexities, etc. There are implemented by using hooks in the more classical pretty-printers.

3 PIPS general design

For a PIPS developer, PIPS can be viewed as in Figure 4. Of course the user interfaces control PIPS behavior but, internally, all the compiling, analyzing and transformation operations are split into basic independent *phases*. They are scheduled by PipsMake, an *à la* `make` interprocedural mechanism that deals with *resources* through the PipsDBM data base manager. An input program is first split into modules that correspond to its subroutines and functions.

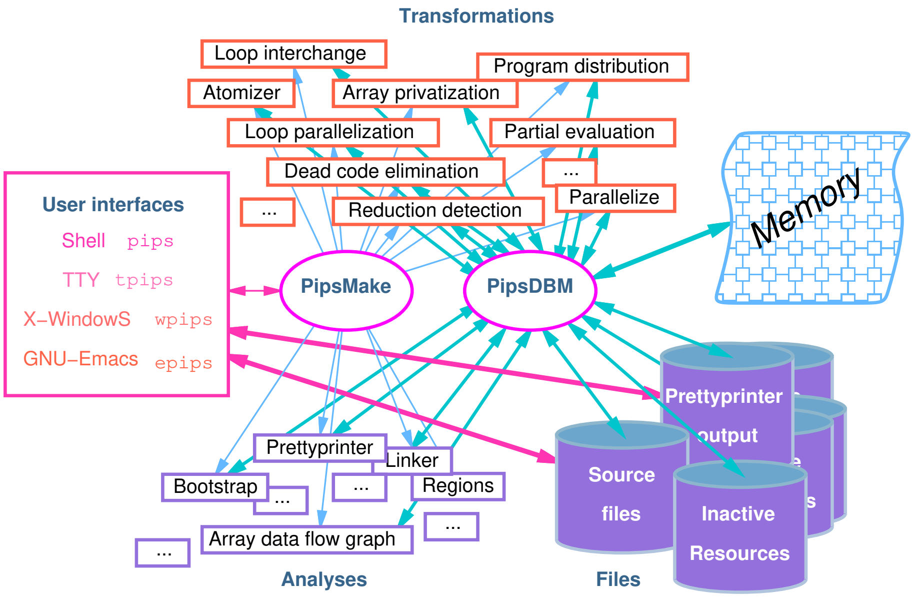

Figure 4: Overview of the PIPS architecture.

PipsMake can then apply the transformation phases on these modules according to some predefined rules. The result is a streamlined programming interface with as few side effects as possible between phases. This allows a highly modular programming style with several developers at the same time. To add a phase to PIPS, only a C function (which may be a wrapper if the execution part of the phase is written in another language) is added (which may of course call other functions) that is called by PipsMake with a module name as argument.

3.1 Resources

All the data structures that are used, produced or transformed by a phase in PIPS are called resources. A phase can request some data structures in PipsDBM, use or modify them, create some other resources, and then store with PipsDBM the modified or created resources to be used later. Examples of resources are the input code, abstract syntax tree, control flow graph, use-def chains, output code, semantic information such as *effects*, *preconditions*, *regions*, etc. These resources are described with the NEWGEN description language (Section 4.1) that allows transparent access to persistent data in PipsDBM from different languages.

3.2 Workspace

Since all the resources about a user code may be stored on disk (Section 3.5), resources are located in a workspace directory. This workspace is created from the source file given by the user and also contains a log file and a PipsMake data structure (Section 3.4 that encodes a coherent description of its content.

```
bool suppress_dead_code(string module_name)
{
  /* get the resources */
  statement module_stmt = (statement)
      db_get_memory_resource(DBR_CODE, module_name, TRUE);
  set_proper_effects_map((statement_mapping)
      db_get_memory_resource(DBR_PROPER_EFFECTS, module_name, TRUE));
  set_precondition_map((statement_mapping)
      db_get_memory_resource(DBR_PRECONDITIONS, module_name, TRUE));

  set_current_module_statement(module_stmt);
  set_current_module_entity(local_name_to_top_level_entity(module_name));

  debug_on("DEAD_CODE_DEBUG_LEVEL");
  /* really do the job here: */
  suppress_dead_code_statement(module_stmt);
  debug_off();

  /* returns the updated code to Pips DBM */
  DB_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, module_stmt);
  reset_current_module_statement();
  reset_current_module_entity();
  reset_proper_effects_map();
  reset_precondition_map();

  return TRUE;
}
```

Figure 5: Excerpt of the function of the dead code elimination phase.

3.3 Phases

Each phase in PIPS use the same interface and simply is a C function taking a module name as argument and returning a boolean describing its completion status (success or failure). Figure 5 shows the example of the dead code elimination phase.

Since PIPS core is written in C, a wrapper function is needed to program PIPS in other programming languages. Currently, the only phase written in Common-Lisp is the reduction detection phase: it calls a Common-Lisp process to perform the job. Communications are handle through PipsDBM files.

3.4 PipsMake consistency manager

From a theoretical point of view, the object types and functions available in PIPS define an heterogeneous algebra with constructors (*e.g.* parsers), extractors (*e.g.* prettyprinters) and operators (*e.g.* loop unrolling). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and even experienced users and it was deemed necessary to assist them by providing default computation rules and automatic consistency management.

The PipsMake library – not so far from the Unix make utility – is intended for interprocedural use. The objects it manages are resources stored in memory or/and on disk. The phases are described in a `pipsmake-rc` file by generic rules

```
proper_effects    > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects

summary_effects   > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

hpfc_close          > PROGRAM.hpfc_commons
    ! SELECT.hpfc_parser
    ! SELECT.must_regions
    ! ALL.hpfc_static_directives
    ! ALL.hpfc_dynamic_directives
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MAIN.hpfc_host

use_def_elimination > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.chains
```

Figure 6: Example of `pipsmake-rc` rules.

that use and produce resources. Some examples are shown on Figure 6. The ordering of the computation is demand-driven, dynamically and automatically deduced from the `pipsmake-rc` specification file. Since it is a quite original part of Pips, it is interesting to describe a little more the features of PipsMake.

3.4.1 Rule format

For each rule, that is in fact the textual name of a phase C function, it is possible to add a list of constraints composed with a qualifier, an owner name and a resource name. Each constraint has the following syntax:

$$< \textit{qualifyer} > < \textit{owner-name} > . < \textit{resource-name} >$$

The qualifier describing the resource can be for each resource:

<: needed;

>: generated;

#: destroyed by the rule;

=: asserts that the resource remains valid even if other dependence rules would suggest to invalidate the resource;

!: asks for applying an other rule before executing the rule. Thus, this constraint has a special syntax since it composed with a rule instead of a resource name. In Figure 6 for instance `hpfc_close` needs an alternate parser for HPF, the must regions and the directives of all the modules of the program.

This qualifier is thus more complex than an Unix `make` but allows finer tuning to avoid useless computation (for example with the =) and be able to expand the rule dependence behavior with !.

An owner name specifies the owner of the needed resource. For example for the code resource, only the code of this module, or the code of all the modules, etc:

MODULE: the current module itself;

MAIN: the main module if any (that is the Fortran `PROGRAM`);

ALL: all the modules

CALLEES: the set of the called modules by the current one; it allows to describe bottom-up analyses on the callgraph.

CALLERS: the set of the calling modules of the current one; it expresses top-down analyses.

PROGRAM: a resource that is attached to the whole program, such as the entities of a program. This concept is orthogonal to the module concept;

SELECT: select a rule by default.

A rule (that is a phase for a PIPS user) is conventionally called a transformation when a resource name appears in the rule with both < and > qualifiers. For example dead code elimination which transforms the code both needs the code and writes it.

It is possible to chose between several rules to make a resource. By default the first one found in `pipsmake-rc` is selected but the programmer can choose to activate an other default rule through the API (see Section 3.4.2 or with the **SELECT** owner). This feature is used for example in the user interfaces to select the kind of decorations in the prettyprinters. For instance the sequential code can be built by several phases, one for the plain code, another one for the plain code with regions, etc.

3.4.2 Up to date resources & API

PipsMake is responsible for dynamically activating rules to build up to date resources and iterates through the transitive closure of the resource dependences (something Unix's `make` is unfortunately unable to do). A small programming interface is available to drive PipsMake from user interfaces (Section 5) or other phases:

- `make(string resource_name, string owner_name)` builds the resources `resource_name` for the set of modules or the program described by `owner_name`;
- `apply(string phase_name, string owner_name)` applies the phase `phase_name` to the set of modules or the program described by `owner_name`;
- `activate(string phase_name)` activates this rule as the new default one when more than one rule could be used to build some resources. Since with this new default rule these generated resources are no longer up to date, the resources that can generated by the new default rule and that existed before are recursively deleted.

3.4.3 Interprocedurality

Interprocedurality is dealt with by the PipsMake **CALLEES** and **CALLEES** owner features: A bottom-up algorithm on the interprocedural call graph such as effects or interprocedural transformers computation will have some rules using **CALLEES**, while a top-down algorithm such as interprocedural preconditions will have some rules using **CALLERS** so as to properly order the phase execution and propagate resources interprocedurally. The correct ordering of the computation is deduced from the `pipsmake-rc` file.

For instance, effects computation relies on the three dependence rules shown on Figure 6: First the proper effects are computed from the code and the summary effects of the **CALLEES** to get the summary effects of the modules called in the current module. Second these proper effects, that only relate to each simple statement, are transformed into the summary effects that take into account the hierarchical structure of the abstract syntax tree. Third, the summary effects are computed to describe the effects that are only non local to the current module and that may later be used to compute the effects of a caller.

3.4.4 Persistence and interruption

Since PipsMake itself is implemented with NewGen, the PipsMake data structure describing the state of Pips is saved on disk when Pips exits. Thus, a Pips operation can continue from a previous saved state.

Furthermore, at a phase boundary, all Pips data structures are managed with PipsDBM. Thus the Pips process can be stopped cleanly at a phase boundary when the control is under PipsMake. Of course the `make()` operation asked to PipsMake fails, but next time the user asks for a `make()`, PipsMake will only launch the phases required to build the lacking resources. This mechanism is heavily used in the user interfaces to insure interactivity.

3.5 PipsDBM resource manager

All the data structures used by the phases to seamlessly communicate with each other are under the responsibility of the PipsDBM resource manager.

PipsDBM can decide to store a resource in memory or on disk. Typically, resources that have been modified are written on disk when Pips exits but the programmer can ask to immediately create a file resource to be viewed by a user interface. The programmer can also declare some resources as unloadable or as memory-only.

PipsDBM also manages a logical date for each resource for the dependence mechanism of PipsMake. Logical dates have been introduced to overcome the lack of Unix time stamp precision (1 second under SunOS4) and the difficulty to have very well synchronized client-server operations in an NFS environment. Of course, the Unix date is also used in order to detect resource files modified by the user (typically an edit through a user interface) where the 1 second precision is good enough.

The programming interface is reduced to

```
string db_get_resource(string ressource_name, string module_name, bool pure)
void db_put_resource(string ressource_name, string module_name, void* value)
```

`pure` is used to specify if the programmer wants the genuine resource or only a copy (for example to modify it and give back another resource).

3.6 PIPS properties

Global variables to modify or finely tune PIPS behavior are quite useful but unfortunately are often dangerous. Thus these variables are wrapped in PIPS as properties. A property can be a boolean, an integer or a string. Using the fact that properties are centralized, they are initialized from a default property file and possibly from a local `property.rc`. Since the behavior is modified by the properties, they are stored in the workspace in order to restart a PIPS session later in the same way even if the global property file has been modified.

4 Programming environment

The PIPS workbench is implemented to form a programming environment. Typically, each phase has its own directory and its own library so that it is easy to add a new phase or modify a phase without too much trouble-shooting. Furthermore, almost all the directory structure is duplicated to have both a production and a development version at the same time.

4.1 NewGen: data structure and method generation

In oder to ease data structure portability between different languages and also to allow data persistence, a tool and a language to declare data structures has been developed: NEWGEN [16]. Once a data description of the *domains* (that are the data structures that can be defined in NEWGEN) is written, NEWGEN constructs several methods to create initialized or not data values, access or modify certain parts of constructed values, write or read data from files, or recursively deallocate data values.

A domain can be a sum (`+` like a C-union) or a product (`x` like a C-struct) of other domains, list (`*`), set (`{}`), array (`[]`) or a map (`->`) of domains. Some domains can be declared as "persistent" to avoid being deleted and the "tabulated" attribute associates a unique identifier to each domain element, allowing unique naming through files and different program runs. The mapping of domains is used to attach semantic information to objects: `alignmap = entity->align`.

Of course, some external types can be imported (such as for using the linear C^3 library in PIPS). The user must provide a set of functions for these domains to write, read, free and copy methods. The NEWGEN environment includes also various low-level classes such as lists, hash-tables, sets, stacks etc. with their various methods (iterators, etc.) that ease the writing of new phases.

```
expression = reference + range + call ;
reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
statement = label:entity x number:int x ordering:int x
            comments:string x instruction ;
instruction = block:statement* + test + loop + goto:statement +
              call + unstructured ;
test = condition:expression x true:statement x false:statement ;
```

Figure 7: Excerpt of the PIPS abstract syntax tree.

An excerpt of the internal representation for Fortran in PIPS is shown on Figure 7. For example, a `call` is a `function entity` with a list of `arguments expressions`. An `expression` can be a `reference`, a `range` or a `call`, and so on. In fact an assignment is represented itself by a call to a pseudo-intrinsic function "=" with 2 arguments, the left hand side and the right hand side argument. And so on for all the intrinsics of the Fortran language. All the syntactic sugar of Fortran has been removed from the internal representation and in this way it is expected not to stick on Fortran idiosyncrasies. The representation is quite simple: there are only 5 cases of statements and 3 cases of expressions. This simplicity also benefits all the algorithms present in PIPS since there are very few different cases to test.

An important feature of NEWGEN is the availability of a general multi domain iterator function (`gen_multi_recurse`). From any NEWGEN domain instance (e.g., `statement`, `instruction`, `expression`...) one can start an iterator by providing the list of domains to be visited, and for each domain the function to be applied top-down, which returns a boolean telling whether to go on with the recursion or not, and a function to be applied bottom-up. The iterator is optimized so as to visit only the needed nodes, and not to follow paths on which no nodes of the required types might be encountered.

In the Figure 8 example, some entities are replaced with others. Entities may appear as variable references, as loop indices, as called functions and in program codes. The recursion starts from `statement stat`: on each `loop`, `reference`, `call` or `code` encountered top-down, the function `gen_true` is applied. This function does nothing but returning true, hence the recursion won't be stopped before having processed all the required domains that can be reached from `stat`. While going back bottom-up, the `replace` functions are applied and perform the required substitution. From the user point of view, only replace functions are to be written.

An example of how NEWGEN defined types are used is shown on Figure 9.

```
static void replace(entity * e)
{ /* replace entity (*e) if required... */}

static void replace_call(call c)
{ replace(&call_function(c));}

static void replace_loop(loop l)
{ replace(&loop_index(l));}

static void replace_reference(reference r)
{ replace(&reference_variable(r)); }

static replace_code(code c) // code_declarations() is a list of entity
{ MAPL(ce, replace(&ENTITY(CAR(ce))), code_declarations(c));}

// args: (obj, [domain, filter, rewrite,]* NULL);
   gen_multi_recurse(stat,
        reference_domain, gen_true, replace_reference,
        loop_domain,      gen_true, replace_loop,
        call_domain,      gen_true, replace_call,
        code_domain,      gen_true, replace_code,
                     NULL);
```

Figure 8: `gen_multi_recurse` example

It is a follow-up of Figure 5. `suppress_dead_code_statement()` begins with a NewGen generic iterator to walk down the module statements applying `dead_statement_filter` on them and walk up applying `dead_statement_rewrite` on all the `statement` domains encountered. In `dead_statement_filter()` one can see that `statement_instruction()` is the accessor to the `instruction` of a `statement` generated by NewGen from the internal representation description. `instruction_tag()` describes the content of an `instruction` (which is a "union"). For loops the result is `is_instruction_loop`. NewGen also generates predicates such as `instruction_loop_p()` to test directly the content of an `instruction`.

NewGen can output data structure declarations and methods for C and Common-Lisp languages since they are the languages used to build Pips. The translation environment is described in Section 4.4. Note that at the beginning of the Pips project, C++ was not considered stable enough so that all the object methods have been written in C, wrapped and hidden in macros and functions.

4.2 Linear C^3 library

An important tool in Pips is the Linear C^3 library that handles vectors, matrices, linear constraints and other structures based on these such as polyhedrons. The algorithms used are designed for integer and/or rational coefficients. This library is extensively used for analyses such as the dependence test, precondition and region computation, and for transformations, such as tiling, and code generation, such as send and receive code in HPF compilation or code control-distribution in wp65. The linear C^3 library is a joint project with irisa

```
static bool dead_statement_filter(statement s)
{
   instruction i = statement_instruction(s);
   bool ret = TRUE;

   if (!statement_weakly_feasible_p(s))       // empty precondition
      ret = remove_dead_statement(s, i);
   else
   {
      switch (instruction_tag(i)) {
      case is_instruction_loop:
      {
         loop l = instruction_loop(i);
         if (dead_loop_p(l))                    // DO I=1,0
            ret = remove_dead_loop(s, i, l);
         else if (loop_executed_once_p(s, l)) { // DO I=N,N
            remove_loop_statement(s, i, l);
            suppress_dead_code_statement(body);
            ret = FALSE;
         }
         break;
      }
      case is_instruction_test:
         ret = dead_deal_with_test(s, instruction_test(i));
         break;
      case is_instruction_unstructured:
         dead_recurse_unstructured(instruction_unstructured(i));
         ret = FALSE;
         break;
      }
   }

   if (!ret) // Try to rewrite the code underneath
      dead_statement_rewrite(s);

   return ret;
}

void suppress_dead_code_statement(statement module_stmt)
{
    gen_recurse(module_stmt,            // recursion from...
                statement_domain,       // on statements
                dead_statement_filter,  // entry function (top-down)
                dead_statement_rewrite);// exit function (bottom-up)
}
```

Figure 9: Excerpt of the function of the dead code elimination phase.

and PRISM laboratories, partially funded by CNRS. IRISA contributed an implementation of CHERNIKOVA algorithm and PRISM a C implementation of PIP

(Parametric Integer Programming).

4.3 Debugging support

Developing and maintaining such a project requires a lot of debugging support from the programming environment at many levels. At the higher level, it is nice to be able to replay a PIPS session that failed since a bug can appear after quite a long time. For this purpose there is a tool that transforms a WPIPS log file into a TPIPS command file to be re-executed later. Every night, a validation suite is run on hundreds of test cases to do some basic non-regression testing. It is not enough to do intensive debugging but is generally sufficient to ensure that a modification is acceptable.

At the programmer level there are some macros to enable or disable the debug mode in some parts of PIPS according to some environment variables that indicate the debug level. Often, there is such a variable per phase at least and the debug modes can be nested to avoid debugging all the functions used in a phase to debug only this phase for example. The debug levels can also be modified according to the property of the same names.

At the NEWGEN level, the recursive type coherence of a data structure can be verified with `gen_consistent_p()` or even at each assignment or at creation time by setting the variable `gen_debug` to an adequate value.

At last, some Emacs macros and key bindings have been defined for the `gdb` Emacs-mode to display the type of a NEWGEN object, the type of a PIPS entity, an expression of the PIPS abstract syntax tree, a PIPS statement, etc. to down the development effort.

4.4 Documentation and configuration files

Such a big project (over 200,000 lines of code for the compilers, mathematical libraries and tools) with developers on different sites) needs to manage a strong documentation support of course, but also as much automatically generated and non-redundant as possible to keep the incoherence level low.

The main idea is to write all the configuration files as technical reports and automatically extract the configuration files from them. Figure 10 is a simplified synopsis of the documentation flow. C headers are generated from `.newgen` files that come from some document describing the usage of the data structure, such as `ri.tex` for the internal representation . The menu files for the user interfaces are generated from `pipsmake-rc` and the property file. All the information needed to feed PipsMake and PipsDBM comes from the `pipsmake-rc` file. At last, LaTeX2HTML is used to generate most of `http://www.cri.ensmp.fr/pips` from all the LaTeX files.

5 User interfaces

Typically, a user interface will use the `make()` command to build a resource to display, use `apply()` to apply a transformation on the code and select default rules, that are options, with `activate()` (Section 3.4.2). There are as many as four interfaces to PIPS:

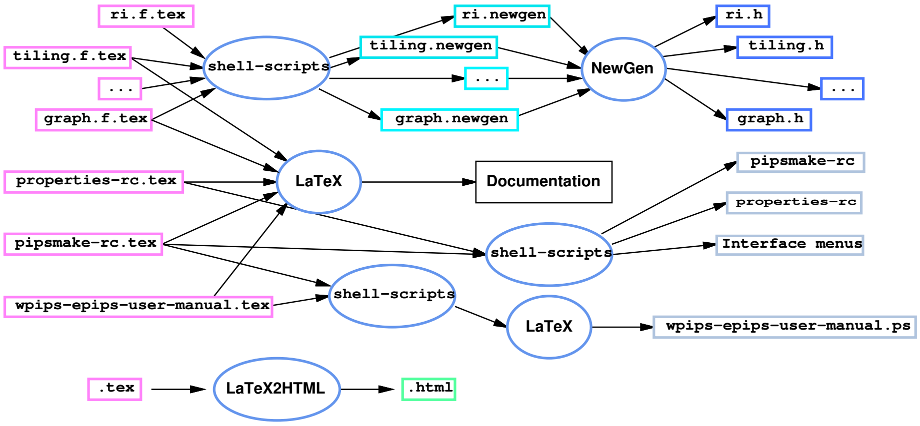

Figure 10: Documentation and configuration system.

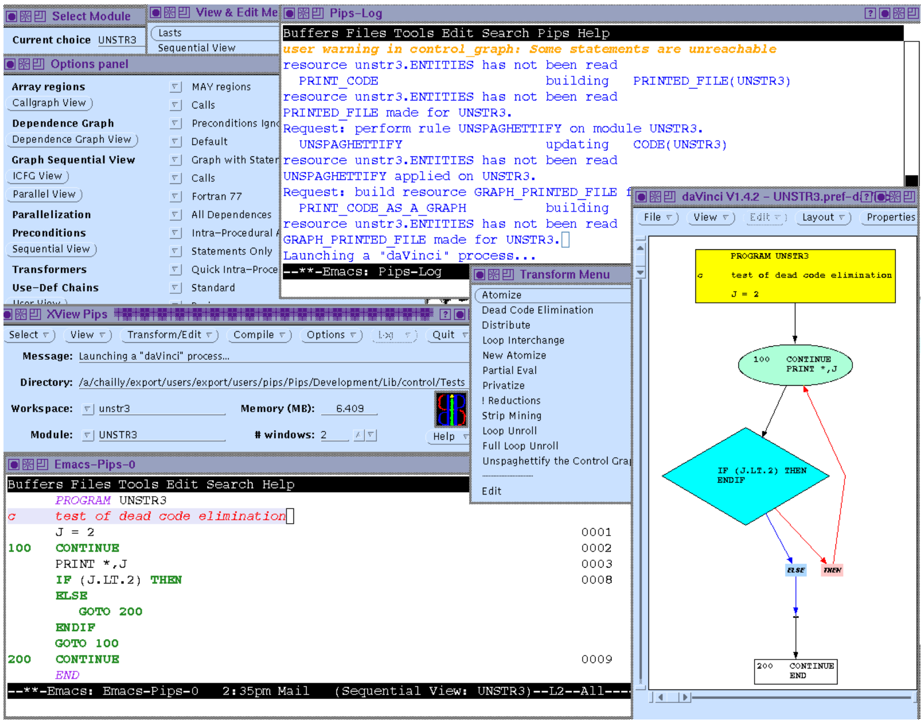

Figure 11: Emacs-Pips interface snapshot.

- `pips` is a shell interface, can do only one operation on a workspace and relies a lot on PipsDBM persistence since the `pips` process exits after each operation;
- `tpips` is a line interface with automatic completion. It's much faster to use for experimentation and debugging than a full graphical interface;

- `wpips` is an XView based easy interface interface;
- `epips` is an extended version of the previous one for GNU-Emacs users. The interface inherits the Fortran mode and color highlight features. Further more graphs are displayed with `xtree` and `daVinci`.

All these interfaces are configured according to the documentation 4.4: when a programmer adds a transformation, it automatically appears in all the interfaces after reconfiguration of the `pipsmake-rc` file.

When dealing with big programs, it may be necessary to interrupt PIPS if a user chooses too costly an option or applies an heavy analysis by mistake. To solve this issue, the user interfaces can interrupt PipsMake at the phase interface level, without compromising the coherence through PipsMake and PipsDBM (Section 3.4.4).

6 Conclusion

6.1 Related Work

Many compiler prototypes have been implemented to test analysis, transformation or code generation techniques.

Parafrase 2 (UIUC): This compiler is a source to source code restructurer for Fortran and C performing analysis, transformation and code generation. It allows automatic vectorization and parallelization.

Polaris (UIUC): This compiler is also a source to source restructurer developed for Fortran 77. It performs interprocedural analysis for automatic parallelization of programs, targeting shared-memory architectures.

SUIF (Stanford): The *Stanford University Intermediate Form* is a low level representation plus some high level additions that keep track of the program structure such as loops, conditions and array references. The input language is C and Fortran through the `f2c` translator. The compiler outputs machine code for different architectures. The compiler initial design and the current distribution of SUIF does not incorporate interprocedural analysis. Such an addition is being worked on [12, 11]. Another problem is that the consistency of analyses is not automatically inforced (for instance invalidation after code transformations).

The originality of PIPS with respect to these other prototypes is the conjunction of:

- Interprocedurality and coherency automatic management in a demand driven system combining database (PipsDBM) and dependence (PipsMake) functions.
- Advanced interprocedural semantic and data-flow analyses already implemented, and their mathematical foundation upon linear algebra (Linear C^3 library).

- Software foundation upon an high level description system (NewGen) which automatically generates functions (access, constructor, destructor) and can iterate on arbitrary structures. An additional by-product is that the code look and feel is homogeneous over the 150,000 lines of the compiler.
- Multi-target:
  - Automatic parallelization for shared-memory
  - Compilation for distributed-memory models (Hpfc, wp65)
  - Automatic data mapping...
- The availability of different interfaces (shell, terminal Tpips, graphical Wpips).

Compiler development is a hard task. Here are examples of many research prototypes. They would benefit a lot from already available infrastructures, and of the synergy of other analyses and transformations, especially interprocedural ones.

- (small) program analyzers and parallelizers: Bouclette (LIP, ENS-Lyon), LooPo (U. Passau), Nascent (OGI), PAF (prism, UVSQ), Petit (U. of Maryland), Tiny.
- Parallel Fortran (HPF spirit) compilers: Adaptor (GMD), Annai (CSCS-NEC), D system (Rice U.), Fx (Carnegie Melon U.), Paradigme (UIUC, based on Parafrase 2), SHPF (Southhampton), VFCS (Vienna)
- Parallel C compilers: Pandore (Irisa), EPPP (CRI Montréal)

The html version of this paper should be used to obtain many links to related work and the Pips project.

6.2 Remarks

Eight years after its inception, Pips as a workbench is still alive and well. Pips provides a robust infrastructure for new experiments in compilation, program analysis, optimization, transformation and parallelization. The Pips developer environment is described in [15], but it also is possible to develop new phases on top of but *outside* of Pips since all (in fact, most...) Pips data structures can be reloaded using NewGen primitives.

Pips can also be used as a reverse engineering tool. region analyses provide useful summaries of procedure effects, while precondition-based partial evaluation and dead code elimination reduce code size.

Pips may be less robust than other publicly available source-to-source compilers but the stress is put on the ability to quickly add new phases such as program transformations, semantic analyses, parsers, parallelizers, etc. or new user interfaces without spending time developing a cumbersome infrastructure.

All the on-line reports of the cri can be found at http://www.cri.ensmp.fr/rapports.html.

The Pips software and documentation is available on `http://www.cri.ensmp.fr/pips`.

6.3 Acknowledgment

We would like to thank many people that have contributed to the Pips project: Bruno Baron, Denis Barthou, Pierre Berthomier, Arnauld Leservot, Guillaume Oget, Alexis Platonoff, Rémi Triolet, Yi-Qing Yang, Lei Zhou.

References

[1] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, December 1993. To appear in Scientific Programming. Available on `http://www.cri.ensmp.fr`.

[2] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, ??(??):??–??, ?? 1996. To appear.

[3] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[4] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux. Master's thesis, Université Paris VI, September 1994. Available by WWW at `http://www.cri.ensmp.fr/ creusil`.

[5] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, April 1994.

[6] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[7] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. CRI TR A 277, CRI, École des mines de Paris, October 1995. To appear in JPDC, 1996.

[8] Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report A-279, CRI, École des mines de Paris, November 1995. Available by WWW at `http://www.cri.ensmp.fr/ creusil`.

[9] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Verlag, August 1995.

[10] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming special issue on LCPC*, ?(?):?, 1996. Extended version of [9].

[11] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer Verlag, August 1995.

[12] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT : A framework for interprocedural analysis and transformation. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[13] François Irigoin. Interprocedural analyses for programming environment. In Jack J. Dongara and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350, Saint-Hilaire-du-Touvet, September 1992. North-Holland, Amsterdam, NSF-CNRS.

[14] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, pages 144–151, June 1991.

[15] François Irigoin, Pierre Jouvelot, Rémi Triolet, Arnauld Leservot, Alexis Platonoff, Ronan Keryell, Corinne Ancourt, Fabien Coelho, and Béatrice Creusillet. *PIPS: Development Environnement.* CRI, École des mines de Paris, 1996.

[16] Pierre Jouvelot and Rémi Triolet. Newgen: A language-independent program generator. EMP-CRI 191, CRI, École des mines de Paris, December 1990.

[17] Alexis Platonoff. Automatic data distribution for massively parallel computers. In *Workshop on Compilers for Parallel Computers, Malaga*, pages 555–570, June 1995.

[18] Alexis Platonoff. *Contribution à la Distribution Automatique de Données pour Machines Massivement Parallèles.* PhD thesis, Université Paris VI, March 1995.

[19] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *Proceedings of the ACM Symposium on Compiler Construction*, volume 21(7), pages 176–185, July 1986.

[20] Yi-Qing Yang. *Tests des Dependances et Transformations de Programme.* PhD thesis, Université Paris VI, November 1993.

[21] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Language and Compilers for Parallel Computing*, July 1994. Also available as TR EMP A/266/CRI.

[22] Lei Zhou. *Statical and Dynamical Analysis of Program Complexity.* PhD thesis, Université Paris VI, September 1994. Available by WWW at `http://www.cri.ensmp.fr/doc/A-255.1.ps.Z`.