CENTRE DE RECHERCHE EN INFORMATIQUE

How to Add a New Phase in PIPS: the Case of Dead Code Elimination

Corinne Ancourt
Fabien Coelho
Béatrice Creusillet
Ronan Keryell

December, 11^{th} 1996

Document EMP–CRI A/300-CRI

Sixth Workshop "Compilers for Parallel Computers" (CPC'96) December 11-13, 1996, Aachen, Germany.

How to Add a New Phase in PIPS: the Case of Dead Code Elimination

Corinne ANCOURT Fabien COELHO Béatrice CREUSILLET
Ronan KERYELL

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

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 and how to add a new phase in PIPS, a dead code elimination, using the management of data structures, of dependences between the various analysis and transformation phases, and using the mathematical libraries in PIPS. PIPS is available to the community and it is what motivates this article.

Keywords: programming environment, workbench, make, interprocedural parallelizer, interactive parallelizer, optimizer, dead code elimination, reverse engineering, CASE.

1 Introduction

Together with the computer 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 factorise 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).

This project aims at combining advanced interprocedural and semantical analyses [10] 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.

PIPS is a source to source and 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 to deal with some of these features to add a new phase in PIPS and a dead code elimination phase will serve as a running example through out the paper. Such a description could seem somewhat cumbersome but it gives an idea of how the PIPS features are organized for people wanting to use PIPS as a compiler development tool since it is publicly available.

We first present the general design of PIPS in Section 2. In Section 3 we sketch a dead code elimination phase and we explain how it is added to PIPS in Section 4.

2 PIPS 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 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.

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 [1], CMF, CRAFT polyhedral method [9] or message-passing code from HPF [2]).

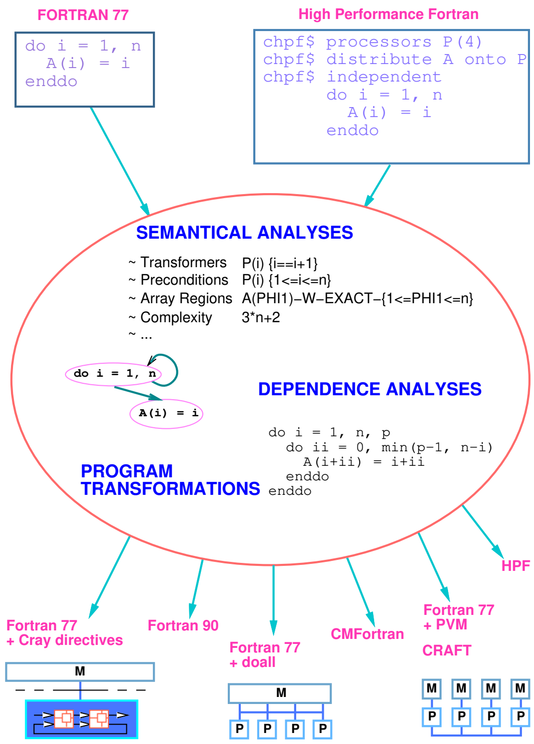

Figure 1: User view of the PIPS system.

In PIPS, the translation process is broken into smaller modular independent operations called phases that communicate only by sharing resources (such as the module code, semantical information, etc.) stored in a resource manager PIPSDBM. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are indifferently referred to as *modules* in the sequel. For the user and the programmer, the phases can be classified into various categories according to their usage.

A first class of phases are called analyses and compute 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 such as some predicates on some integer variables of the program [5] or the regions that describes some array elements [4].

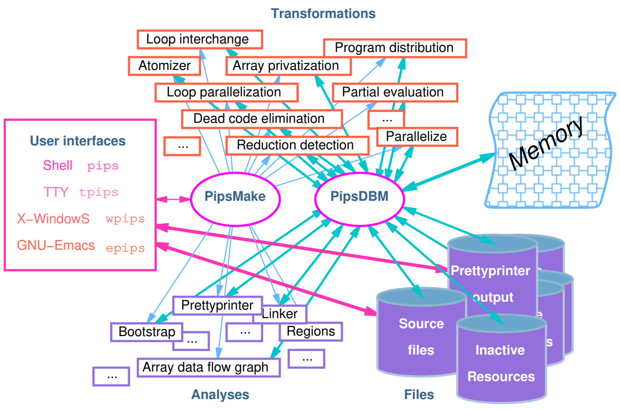

Figure 2: Programmer view of the PIPS system.

Another important class is the set of the code generation phases that produce a new code from the information given by the analyses. Important code generators are the parallelizer, the code distributor [1] and the HPF compiler [3].

A third class contains the prettyprinters that transform and merge some internal PIPS resources (code, preconditions, etc.) in a user viewable file.

The last phases are called transformations because they modify the program code. The available transformations are loop distribution, scalar and array privatization based on regions [4], etc. In this article, we present a dead code elimination transformation.

All the resources about a program are kept in a *workspace* database with the help of PIPSDBM and the global coherence is insured with the interprocedural PIPSMAKE manager. More details can be found in [8].

3 Defining a dead code elimination phase

Semantical information known at compile time can be used to simplify the code in many situations, such as for code specialization when a library function is used in a specific context, when all the code is not used and thus can be removed, for simplifying old code with useless parts or as back-end optimizer after some generic code transformations or compilation phases.

In PIPS for example, the preconditions give some affine predicates on integer variables of the program. These predicates can be used to simplify the code locally by partially evaluating it.

The simplest rule is to remove a statement when its predicate has no integer solution such as in an IF statement with a condition never satisfied such as $\{0 == -1\}$ on Figure 3.

```
      J = 3
C  P(J) {J==3}
      DO J = 1, 3
C  P(I,J) {1<=J, J<=3}
         IF (J.GE.5) THEN
C  P(I,J) {0==-1}
            I = 5
         ENDIF
      ENDDO
```

Figure 3: Simple code with dead code highlighted by preconditions

In fact this can be done further: since the IF is then empty, it can be removed itself if the condition computation as no side effect such as doing an I/O or touching a global variable. This last information is abstracted in PIPS by the proper effects resource that give information about the variables read or written for each statement.

For the loop optimizations, if the preconditions can prove that a loop cannot be executed, it can be removed. But if the computation of the loop bounds has some side effects, the loop must be replaced with these equivalent side effects, such as setting the index variable to its lower bound value if it is used later.

In the same way, if a loop can be proved to be executed only once, the loop can be replaced by its body and if the loop statement itself has a side effect, the same side effect should be kept. The information needed to implement this kind of transformation is the preconditions, the proper effects and of course the code itself.

Many rules such as the previous one can be easily added and implemented in a dead code elimination transformation in PIPS. In the following part we do not focus on the kind of optimizations that can be made but on how to use the PIPS framework to add such a transformation.

4 Adding a new phase: dead code elimination

In this section, we are explaining the different tasks needed to add a new phase in PIPS. First we describe the global directory hierarchy and phases basic files before describing the use of the PIPSMAKE, PIPSDBM and NEWGEN tools.

4.1 Directory hierarchy

The PIPS main directory hierarchy is mainly organized with the following concepts [8]:

Production: contains the last coherent version of the PIPS software (`Bin`, `Runtime`), documentation (`Doc`, `Html`), examples of validation (`Validation`) and sources (`Src`). The production directory is pointed by `$PIPS_ROOT`;

Development (`PIPS_DIR/Development`) is used to develop all the pieces of PIPS. By default, a part of PIPS in development is tested with all the other parts from the `Production` directory to avoid as much as possible the interactions between concurrent developments of different parts at the same time (`$PIPS_DEVEDIR`). Important directories of `Development` are `Libs` (the internal parts of PIPS, such as each phase), `Passes` (the various PIPS executable with the different user interfaces `pips`, `tpips` and `wpips`), `Documentation` (the documentation itself of course but also many configuration files), `Runtimes` (support for HPF, message passing and graphics) and `Scripts`. Installing a part of PIPS (with a classical GNU `make install`) is done in fact by copying the needed files and directory in the *ad hoc* directories of `Production`;

For portability purpose, an architectural hierarchy *à la* PVM has been reused in PIPS with the `$PIPS_ARCH` variable to compile and store different binary architectures.

4.2 Basic file definitions

Since all the phases are kept in the PIPS library, a new phase p needs a new directory d in `$PIPS_DIR/Development/Libs` or at least should be put in an old one d. The generic structure of a PIPS source directory and a library in particular contains a `config.makefile`, a d`-local.h` file and various source files.

A phase p itself is a function compatible with the C declaration of Figure 4. Thus if one wants to write a new phase in another language such as Fortran or Common Lisp, a C wrapping function should be used to call the phase kernel itself in this particular language and return the success flag. A phase returns false when it fails and PIPSMAKE stops executing the various phases.

The `config.makefile` is aimed to derive a read-only GNU `Makefile` through the PIPS make file generator `pips-makemake` that will be used to compile the sources in the directory d to build a library `lib`d`.a` that will be linked later with other parts of PIPS to have a working executable. Many PIPS-specific generic rules are generated by `pips-makemake` (for C, Fortran, `lex` and `yacc` files, HTML and LaTeX documentation, various binary architectures, etc) and often the `config.makefile` is quite short, typically declaring the source files to use.

In the same way, a local include file d`-local.h` is used by the `Makefile` to generate a file d`.h` that declares all the functions and variables to be known from outside to use the d library. d`.h` is the concatenation of d`-local.h` and all the non-static objects of the local C files extracted by `cproto`.

To sum up the directory creation and declaration for a new phase:

1. create the directory d if necessary;
2. declare the list of the useful files in a `config.makefile`;
3. run `pips-makemake`;
4. write a p function that does the phase job;
5. run GNU `make`
6. debug it and iterate on previous points, eventually by doing a local pass as described later;
7. once the previous code has been compiled, the library is ready to be added to PIPS with editing the file `$PIPS_DIR/Development/Scripts/dev/define_libraries.sh` and typing `make install`.

In fact, the last points are a little bit interleaved since it rather hard to debug a new or modified phase without installing it, but it is dangerous to install some unstable code. For that purpose, there exist the rules `test`, `ttest` and `wtest` to locally build the passes `pips`, `tpips` and `wpips` respectively by linking with the local library instead of the one installed in the production version `$PIPS_ROOT/Lib`. Since often several libraries are co-developed and need to be linked together to be debugged, one can add all the other library d' by doing a `ln -s ../`d'`/lib`d'`.a`.

```
bool p(char * module_name)
{
bool good_result_p = TRUE;

 /* Some computation about phase p */

 return good_result_p;
}
```

Figure 4: Basic phase interface

In our case, we want to add a dead code elimination phase that is a transformation. Since there are many transformations with a rather small code, these are put all together in the `transformations` library. Since this library already exists, it is useless to create a new `transformations` directory in `$PIPS_DEVEDIR/Libs`. Just create the file `dead_code_elimination.c` and add its name in the `LIB_CFILES` of the `config.makefile` for the file to be considered when building the library.

Our dead code elimination has no reason to fail since we only apply legal optimization if necessary, thus the `suppress_dead_code()` function always returns `TRUE`, as we will see on Figure 6.

4.3 Using the interprocedural 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.* pretty-printers) and operators (*e.g.* loop unrolling and other transformations). 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 the corner stone of PIPS interprocedurality. 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 that use and produce resources. The ordering of the computation is demand-driven, dynamically and automatically deduced from the `pipsmake-rc` specification file.

In our dead code elimination example, we need the code of the current module of course, but also its preconditions and its proper effects for the semantical information. The entities of all the program are also needed to have access to variables, `COMMON`s and so on. The global effect of this transformation is to rewrite the code of the current module. These dependences are abstracted by the `pipsmake-rc` syntax on Figure 5.

```
suppress_dead_code    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.preconditions
```

Figure 5: Example of `pipsmake-rc` rules.

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 describs how the resource can be used: `<` for 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 another rule before executing the rule.

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. Two interesting owners are `CALLEES` (the set of modules called by the current one, used to describe bottom-up analyses on the call-graph) and `CALLERS` (the set of modules calling the current one that can expresses top-down analyses). They are the mechanism that deals with interprocedurality in PIPS. In our small example, the interprocedurality is not highlighted because if the dead code elimination uses interprocedural information (the preconditions and the effects), it is not interprocedural *per se*.

An API is available for advanced usage to call PIPSMAKE directly to build a resource inside a user interface for example.

4.4 Using the PIPS data structures with PIPSDBM database manager

Every useful phase needs at least to read and use some data structures describing the user program or some other data and also needs to write some results or the modified user program for example. In PIPS, data management is done through PIPSDBM, a system able to read and write any NEWGEN [7] data structure or even non-NEWGEN ones (with user-provided read, write and copy functions) in a portable way (architecture and memory location independent) and persistent way. Such a resource can be on disk or memory but when PIPS exits, all the resources that are flagged as useful are stored on disk and when the programmer asks for a resource, it is loaded from disk if it is not in memory.

In such a way, the PIPS design is more modular, with many small independent modules. It is preemptive at each phase boundary since then all the data are in a coherent state with PIPSDBM. It is useful to have more reactive user interfaces. Further more, a PIPS session can be shutdown and brought up later by retrieving the former state from disk.

```
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 6: Excerpt of the main function of the dead code elimination phase.

4.4.1 Loading and storing resources

Since the only useful operations are to get a resource from or to put a resource into PIPSDBM, the programmer interface is reduced to

```
string db_get_resource(string resource_name, string module_name, bool genuine)
void db_put_resource(string resource_name, string module_name, void* value)
```

The first function returns a pointer to the data structure `ressource_name` for module `module_name`. If `genuine` is `TRUE`, the programmer gets the genuine resource. That means that modifying it will modify the resource used by subsequent users. The `genuine` flag with a value `TRUE` is useful to modify a resource or to use it read-only for example. When using it as a scratch-pad or modifying it to transform it in another resource, one must ask for a copy of the genuine resource with the value `FALSE` for flag `genuine`.

For our dead code elimination, we need to modify the module code, thus we need to get the genuine resource and put it back into PIPSDBM. The information about the code used to simplify it is used on a read only basis, that means that the genuine resources can be accessed to avoid useless copies since we do not modify them. `DBR_CODE` (the abstract syntaxic tree of the module), `DBR_PROPER_EFFECTS` (the proper effects of the statements of the module) and `DBR_PRECONDITIONS` (the preconditions for the statements of the modules). We can now write the main function for our dead code elimination as shown on Figure 6 with the use of PIPSDBM.

4.4.2 Creating new resources

Our dead code elimination phase is a simple code transformation example and in fact all the needed resources already exist in PIPS and we do not produce some new resources. However, if we try to add some new semantical analyses, some new languages features, etc. some new resources to store these information is necessary. This new resource creation process is explained in the PIPS development environment guide [8].

To avoid conflicts with already defined data structures in the PIPS sources, a resource *`resource`* is named internally `DBR_`*`RESOURCE`* with big caps.

Since we do not use dynamic linking and UNIX `nlist()` for portability reasons, adding a new resource with the related methods require a recompilation of PIPS. But since adding a new resource does not occur very often and implies that a new phase is also added or at least modified to use it and that it requires also a recompilation, it is not a big deal.

The new resources must be added in the `$PIPS_DIR/Validation/Libs/pipsdbm` library that is the core of PIPSDBM. The methods to be used for loading, unloading, freeing and verifying are added in the `method_map[]` array of the file `method_io.c`. The file `module.c` must be modified to reflect the unloading order of the resource hierarchy. For file resources (used for example for prettyprinters), the file `disk.c` need also to be updated. After this modification, the PIPSDBM library must be installed and PIPS recompiled (with `make-pips -r`). Further explanation are found in the development environment guide [8].

4.5 Avoiding global variables

Global variables in big projects such as PIPS should be avoided as much as possible since they are often the source of big trouble when many different modules are interacting and when the global state has not been proved to be safe. Two encapsulations are used in PIPS to provide global state keeping and enabling global options.

4.5.1 Hidden global variables

Nevertheless, some information such as "what is the statement of the current module" is often used in many places and it is cumbersome having to pass this information through some recursions. For that purpose, some interesting global variables exist but they have been encapsulated in `set_` and `reset_` methods to insure that a variable will not be left with an old value when leaving a phase. The `get_` method is used to use the value.

In the main function of dead code elimination on Figure 6, the current module statement and entity are stored by using the `set_current_module_statement()` and `set_current_module_entity()`.

`local_name_to_top_level_entity(n)` gets the entity corresponding to the module name. The module statement and entity are useful to directly access data structures that are global to the current module (such as variables, etc.) when doing some code analysis or transformation.

Since the dead code elimination needs to know some semantical information about the current module, such as the proper effects of a statement to deal with I/Os and the preconditions to have informations on variable values, it is stored with `set_proper_effects_map()` and `set_precondition_map()` for later use. At the end of the procedure, the global variables are reset accordingly.

4.5.2 Properties

The other object in PIPS that allows parameterization is called a property. Properties are used for various purposes: adding some information or selecting a style in a prettyprinter, asking to compute some statistics, selecting a special feature, etc.

The user interfaces can be used to modify the current properties and if a local `properties.rc` exists, the last one modifies the properties defined in the main PIPS `properties.rc` file.

```
      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* ;
sequence = statements:statement* ;
statement = label:entity x number:int x ordering:int
          x comments:string x instruction ;
instruction = sequence + test + loop + goto:statement
            + call + unstructured ;
test = condition:expression x true:statement x false:statement ;
```

Figure 7: Excerpt of the PIPS abstract syntax tree.

Note that since the PIPS behavior can be highly altered by the properties, the properties used at the time of a workspace creation is stored in the workspace itself to be sure to always use the same properties along the life of a workspace. In such a way, even if the main or local `properties.rc` is modified, the PIPS behavior is coherent with the information persistence through different runs.

4.6 NewGen data structures in PIPS

An other key point of PIPS is the NewGen [7] description language that is used in PIPS to define all the data structures. Domains are defined NewGen to express complex data structures such as sums (`+` like a C-union) or products (`x` like a C-struct) of other domains, lists (`*`), sets (`{}`), array (`[]`) or maps (`->`) of domains. These domains are translated by NewGen in type definitions for different languages (C and Common-Lisp for now) with different methods to deal with this data: creators, destructors, accessors, readers and writers, etc. The last ones allow persistence and having PIPS written in different languages but sharing the same data. In this way, NewGen adds an object oriented layer that abstracts the target language data structures.

In our dead code elimination example, we do not define any new type but only use the internal representation of the abstract syntax tree. An excerpt of its definition is on Figure 7. All the NewGen data structures are defined in the LaTeX files from `$PIPS_DIR/Development/Documentation/newgen` that are used to generate the documentation about the types and of course the target language specifications.

NewGen defines a generic mono- or multi-domain iterator (`gen_recurse` and `gen_multi_recurse` in C) that recursively applies functions on a list of domain top-down and then a bottom-up on a data structure. In our example on Figure 8 in `suppress_dead_code_statement()` we iterate on all the statements of the code and apply on them `dead_statement_filter()` top-down to eliminate some dead code, recurse if this function returns `TRUE`, and then `dead_statement_rewrite()` bottom up to remove some code that has become useless because of the previous elimination of nested statements. All the accessors such as `statement_instruction()` to get or set an instruction of a statement or the `is_instruction_loop` tag to test if an instruction is a loop are generated by NewGen.

4.7 Using the C^3 linear algebra library

To test if an instruction is useful, we need to evaluate the preconditions to know if there is an integer solution to the predicate. For this purpose, we use the C^3 linear algebra library that provides support for vector and matrix operations, feasibility tests with simplex of Fourier-Motzkin, etc. In this example, we use in `dead_statement_filter()`

```
bool
statement_weakly_feasible_p(statement s)
{
    transformer pre =
                load_statement_precondition(s);
    /* Get the precondition linear constraint
       system associated with s */
    Psysteme sc =
   predicate_system(transformer_relation(pre));
    /* Test if there is at least a solution */
    bool feasible_p = !sc_empty_p(sc);
    return feasible_p;
}

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

   ifdebug(9) {
       debug(9, "dead_statement_filter",
     "The current statement:\\n");
       print_text(stderr,
  text_statement(get_current_module_entity(), 0,
                                             s));
   }

   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 for example */
            ret = remove_dead_loop(s, i, l);
         else if (loop_executed_once_p(s, l)) {
     /* DO I=N,N for example */
            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
                                      here */
                statement_domain, /* on
                                 statements */
/* entry function (top-down): */
                dead_statement_filter,
/* exit function (bottom-up): */
                dead_statement_rewrite);
}
```

Figure 8: Excerpt of the dead code elimination phase.

`statement_weakly_feasible_p()` that calls the C^3 function `sc_empty_p()` to know if there is a solution to the system of constraints.

4.8 Debug support

In addition to fine debug support at the debugger level of NewGen and pips objects, higher level debug can be triggered using environment variables such as `$DEAD_CODE_DEBUG_LEVEL` on Figure 6 with the `debug_on()` called up to a matching `debug_off()`. For example, if `$DEAD_CODE_DEBUG_LEVEL` has the value 9 or more, `dead_statement_filter()` displays its current statement on Figure 8.

4.9 Non-regression validation tests

Managing a big project is very hard without appropriate test procedures. For this goal, it is quite useful to add some test examples to run PIPS every night on it and to compare the output with files containing the expected results. In this way, running a validation after modifying a part of PIPS give more confidence about the global status.

The typical test example is put in a directory in `$PIPS_ROOT/Validation`, for example in the `Transformation` directory. For each example Fortran code contained in *example*`.f`, a script *example*`.test` and a *example*`.result/test` should be written so that every night the script is executed and use *example*`.f` to generate an output compared to the `test` expected output.

4.10 User interface

The various user interfaces of PIPS use the `pipsmake-rc.tex` file to deduce some automatic name completion and graphical menus for WPIPS and EPIPS. For the graphical interface, the transformation menu layout is specified by the `\begin{PipsMenu}{Transformations}` ... `\end{PipsMenu}` LaTeX environment of the `pipsmake-rc` file.

Since the phase names are not very user-friendly, some aliases are defined also in the `pipsmake-rc` file to be used instead of the phase names in the `Options` menu for example.

4.11 Documentation

This section comes late but of course it is nevertheless one of the most important to deal with long-term usage and software maintenance.

The documentation system heavily rely on GNU `make`, LaTeX and LaTeX2HTML since they are portable, widely spread and easily integrated in an automatic documentation process *à la* UNIX. LaTeX is very nice to have a high quality printed document and LaTeX2HTML to add navigation capabilities for browsing.

When the `pipsmake-rc` file is updated to add a new phase, some documentation should also be added to present what the phase is about. In this way, the user manual can automatically be derived with a description of every phase accessible from a menu.

Once the documentation exists, it can automatically be published on the Internet with the World Wide Web protocol and retrieved using the standard meta-indexers as AltaVista by an interesting world-wide side effect...

5 Conclusion

The originality of PIPS is the conjunction of:

- modularity and easy adding of a new phase as stated in this article;
- interprocedurality and automatic coherency 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 [2], WP65 [1])
- automatic data mapping…

- the availability of different interfaces (shell, terminal TPIPS, graphical WPIPS).

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 [6], but it is also 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 [4] 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.

The PIPS software and documentation is available on `http://www.cri.ensmp.fr/pips`. The HTML version of this paper should be used to obtain many links to the PIPS project and related work.

5.1 Acknowledgment

We would like to thank many people that have contributed to the PIPS project: Bruno BARON, Denis BARTHOU, Pierre BERTHOMIER, François IRIGOIN, Pierre JOUVELOT, Arnauld LESERVOT, Guillaume OGET, Alexis PLATONOFF, Rémi TRIOLET, Yi-Qing YANG, Lei ZHOU.

References

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

[2] 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.

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

[4] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming special issue on LCPC*, 24(6):?, 1996.

[5] 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.

[6] 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.

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

[8] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report 289, CRI, École des mines de Paris, April 1996.

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

[10] 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.