ABSTRACT
The PIPS project was started in 1988 to investigate the automatic detection of medium- and large-grain parallelism in scientific programs thanks to summarization techniques based on convex array regions. By 1992 the PIPS system had reached its original goals, but it has morphed into a comprehensive, open-source platform still in use today. What were the key scientific and engineering decisions that made this possible in spite of some inevitable shortcomings?

Original paper: http://dx.doi.org/10.1145/109025.109086

Categories and Subject Descriptors:
D.3.4 [Processors]: Compilers, Optimization

Keywords:
Automatic Parallelization; Interprocedural Analysis

1. INTRODUCTION
The goals of the PIPS project were (1) to find automatically medium and large grain interprocedural parallelism in Fortran 77 scientific programs and (2) to express it thanks to a source-to-source translation process targetting shared-memory multiprocessors.

PIPS was based on polyhedral techniques for command abstraction [16], procedure summarization with convex array regions [31] and hence dependence tests, and was to be the first polyhedral compiler.

2. ASSUMPTIONS AND KEY CHOICES
2.1 Six Key Assumptions
The necessary parallelism was to be found in loops dealing with arrays indexed by affine expressions only. Unlike data parallelism, task and instruction-level parallelisms were deemed of no use because bounded by the code size and the number of operators.

Recursive pieces of code did not require a precise analysis because they do not contain data parallelism.

Recursive functions were not used in scientific code and do not contain data parallelism either. Functional parameters are not common enough to be taken care of either.

Convex array regions are precise enough to find medium-grain parallelism and polyhedral operators are fast enough to analyze whole applications.

PIPS would perform whole program parallelization. The source code of all modules is assumed available.

Finally, source-to-source is sufficient to express parallelism and optimizations, while preserving key information for programmers and optimizers.

2.2 Scientific Choices
Because parallelism was to be found in DO loops, PIPS internal representation is not based on a usual control flow graph (CFG). Instead it uses a recursive combination of abstract syntax trees for structured components and CFGs for unstructured ones. This combination was called a hierarchical control flow graph (HCFG).

It was decided to use abstract commands to build abstract stores for interprocedural analysis with only two traversals of the call graph, and, later, to perform convex array region propagation. Abstract commands are also useful to shorten the analysis of nested loops, which are common in scientific codes.

2.3 Design Choices
The design was guided by our knowledge of Parafrase [20] and PTRAN [30] internals, and ended up with a minimal internal representation both for control constructs and for entities. No intrinsics, not even the assignment, are singled out: they all are calls with side effects. Functions, variables, commons... are all entities. Hence, the internal representation [11] is language-independent.

The Newgen [32] data description language was adopted to implement PIPS internal representation. Newgen uses dynamic typing to provide automatically compact data management features, thanks to a layer of macros and functions, including higher-order generic traversal operators.

Multiple implementation languages were available for fast prototyping, CommonLisp, and efficiency, C. Thanks to Newgen, data structures could be shared by the two languages.

A unique symbol table was designed to ease interprocedural analyses.

Persistence was necessary for interprocedural analyses because of the memory size constraint and to allow interactive
uses. A make-like system was implemented to ensure the PIPS database consistency and some pass ordering automatically [11]. It also made PIPS modular and evolutive, with an easy declarative way to add new passes [5].

A sparse implementation was chosen for the polyhedral library because few variables are involved in each constraint. Finally, an interactive window-based interface was added for demonstrations and for pedagogical reasons.

2.4 Future Work of 1991

We anticipated the need for path transformers [18], profitability analyses [36], programming rules, assertions about key parameters and data transformations such as array expansion or privatization [12, 13], as well as improvement in convex array region translation, and feared complexity due to affine transformers.

We assumed that non-convex array regions would be useful in signal processing codes and expected a lack of target machines and parallel languages.

3. LOOKING BACKWARD

PIPS later competitors such as Parafrase-2, Polaris, SUIF, SUIF-2 are gone, but PIPS is still with us and has not been rewritten in Java or C++. Why?

3.1 Strong Points

Polyhedral techniques are very flexible [2] and often not too complex [35]. They support loop parallelization, with neither control nor call restrictions [13], and automatic distribution [9, 23].

Interprocedural techniques, which PIPS pioneered, are now key to compilers for heterogeneous targets [14, 1]. A language-neutral high-level source-to-source internal representation is useful for the user who can recognize her source code, for debugging the compiler since the internal representation can be compiled and executed at any step, and for supporting new input languages.

Automatic consistency is important to manage interprocedural issues and to add new passes. Pass dependencies are managed by PIPS, not by the pass programmer.

The data description language NewGen was useful to update the internal representation without modifying PIPS code. This proved key when adding C to Fortran as a source language.

However, interactivity and multiple implementation languages turned out to be of little use.

3.2 Extensions

To process industrial code, we had to extend the initial Fortran subset to cope with entries, stack allocation, dependent types, while loops and the HPF/OpenMP directives. And then we had to support C99 with pointers and dynamic allocation.

We combined static and dynamic analyses to obtain both safety and efficiency: array overflows, aliasing detection and proper variable initialization [27, 26, 28]. We also had to analyze non-integer variables and non-affine expressions because they control the behaviors of large applications.

To process C, we had to extend and/or to implement usual code analyses and transformations such as use-def chains, points-to analysis, dead-code elimination, control simplification, induction variable substitution, scalarization and loop fusion [5, 1, 22].

We also had to go beyond a simple interprocedural approach and to add procedure cloning, inlining and outlining: procedure boundaries must be moved to fit the target machine [14].

Finally all kinds of parallelism must be detected. Commutative and transitive expression optimization was added to improve ILP [38]. Reduction detection [17], small vector parallelism, SSE or AVX [15], and GPU code generation with data distribution [1, 3] were introduced later. Finally task parallelism must be exploited with multicores [19, 33].

On the implementation side, list-based algorithms were replaced by hash-tables to scale up when large applications are analyzed, modules written in CommonLisp were rewritten in C, and exception management had to be introduced to cope with magnitude overflows in polyhedral operators. A Python embedding, pyxs, was introduced to provide the flexibility required with in- and out-lining [14]. The 1.x Par4All initiative [33] also uses Python to increase PIPS robustness and to simplify its use (see par4all.org).

4. CONCLUSIONS

PIPS has proved over the years to be a fertile ground for the polyhedral model [21], data transformations [13], communication synthesis [4, 7], compilation for distributed memory machines [9, 23], ILP [37], code maintenance [29, 6], program verification [25], scratchpad management [8], offload compilers [14, 1, 10], and task parallelism [18, 33].

These advances were made possible by PIPS modular and evolutive structure, by its language-neutral internal representation and by new analyzes and a better understanding of their domains [36, 34, 13, 24, 22], but also by many contributors to the PIPS infrastructure, to its classical pass portfolio, to its tutorials and website. And PIPS grew from 50 KLOC in 1991 to about 600 KLOC in 2014.

New challenges are now addressed with PIPS: manycores, heterogeneous systems, complex memory hierarchies, code modelization, tool combinations and parallel languages. On the infrastructure side, we intend to combine robustness for industrial use and openness for research.

5. ACKNOWLEDGMENTS

We owe many thanks to the PIPS contributors, who are too numerous to be all mentioned here, but whose names are listed on its website, pips4u.org. Their work was funded by many French and European programs, institutions and companies: ANR, CEA, CNRS, DRET, EDF, ESPRIT, HPC Project/SILKAN, Télécom SudParis and THALES.

We are also thankful to Claude Girault, who helped fund and set up the so-called C3 group, and to the members of this group, especially Corinne Ancourt, Philippe Cluzeau, Alain Darde, Paul Feautrier, Catherine Montgenet, Guy-René Perrin, Patrice Quinton and Yves Robert.

6. ADDITIONAL REFERENCES
