PyPs, a programmable pass manager

Serge Guelton[1], Mehdi Amini[2,3], Ronan Keryell[3], and Béatrice Creusillet[3,4]

[1] Télécom Bretagne, Plouzané, France
[2] Centre de Recherche en Informatique, MINES ParisTech, Fontainebleau, France
[3] HPC Project, Meudon, France
[4] UPMC/LIP6, Paris, France

Abstract. As hardware platforms are growing in complexity, compiler infrastructures need more flexibility: due to the heterogeneity of these platforms, compiler phases must be combined in unusual and dynamic ways, and several tools may need to be combined to handle specific parts of the compilation process efficiently. The need for flexibility also appears in iterative compilation when different phase orderings are explored.
In this context, we need to assemble pieces of software—typically compiler phases—without having to dive into the tool internals. The entity in charge of this phase management is called a "pass manager". While pass managers used to rely on a statically defined schedule, the introduction of plugins in GCC and the current trends in compiler design showcased by LLVM pave the way for dynamic pass schedulers.
The contributions of this paper are a high-level modeling of pass chaining and its implementation in the PIPS source-to-source compiler framework under the name PyPS. As a result, we propose a high level API for building end-to-end specialized compilers with the minimum efforts validated by 5 prototypes detailed in the paper: an iterative compiler, an OpenMP directive generator, a C-to-CUDA translator, a multimedia instruction generator and a C-to-FPGA translator.

Keywords: programmable pass manager, compiler infrastructure, iterative compilation, source-to-source transformation, heterogeneous computing

1 Introduction & Motivation

The continuous search for performance leads to numerous different hardware architectures, as showcased by current trend for heterogeneous computing. To use these architectures efficiently, new languages and paradigms are often introduced, but they typically only target specific architectures. For instance Advanced Vector eXtensions (AVX) intrinsics target vector registers of recent $x86$ processors, OpenMP directives target multi-cores, and most noticeably CUDA targets Nvidia's Graphical Processing Unit (GPU). It is difficult to master all these language-hardware bindings without loosing control on the original code, thus compilers play a key role for "filling the gap" between generic sequential languages and specific parallel languages [2]. Because of the diversity of targeted hardware,

flexibility and retargetability are critical properties for compiler frameworks that must keep up with the steady pace of hardware founders. Also, applications targeting heterogeneous architectures, e.g. General Purpose GPU (GPGPU) with an $x86$ host, raise new problems such as the generation of different codes in different assembly languages, remote memory management, data transfer generations... thus requiring new functionalities that are not available in mainline compilers at that time.

Additionally, iterative compilation [13,18], the process of iteratively transforming, compiling and evaluating a program to maximize a fitness function, is more and more considered as an alternative to standard program optimizations to solve complex problems, but it requires a dynamic reconfiguration of the compilation process. In a compiler infrastructure, the latter is managed by the *pass manager*. Because of the much more complicated compilation schemes, this pass manager must be flexible and provide ways of overtaking the traditional hard-coded pass sequence to allow compiler developers to manipulate the interactions between passes and the compiled program dynamically.

This paper is organized as follows: we argument in favor of source-to-source transformations in Section 2, and introduce a high-level model for pass chaining in Section 3. We then present in Section 4 Pythonic PIPS (PyPS), our implementation in the Paralléliseur Intéractif de Programmes Scientifiques (PIPS) source-to-source compiler framework. It involves a high level Application Programming Interface (API) with a high level abstraction of compilation entities such as analyses, passes, functions, loops, etc. Building up on the fact that an API is only relevant when used extensively, we demonstrate our proposal in Section 5 where the compilation scheme of five distinct compiler prototypes using our model are detailed. Finally we present related works in Section 6.

2 Using a Source-to-source Transformation System

Historically, many transformations have been expressed at the source code level, especially parallelizing transformations, and many successful compilers are source-to-source compilers [5,10,3,22] or based on source-to-source compiler infrastructures [16,32,26,15,7]. Such infrastructures form a flexible transformation system relevant for heterogeneous computing, e.g. parallelism detection algorithm, variable privatization, communication generation etc.

In the heterogeneous world, it is common to rely on specific hardware compilers to generate binary code for the part of the application intended to be run on a particular hardware. Such compilers usually take a C dialect as input language to generate assembly code. Thus, it is mandatory for the whole toolbox to be able to generate C code as the result of its processing.

In addition to the intuitive collaboration with hardware compilers, source-to-source compilers can also collaborate with each other to achieve their goal, using source files as a common medium, at the expense of extra switches between the Textual Representation (TR) and the Internal Representation (IR). Figure 1 illustrates this generic behavior and we present in Section 5.6 how we delegate

some loop nest optimizations to the Polyhedral Compiler Collection (PoCC) tool. Moreover, two source-to-source compilers written in the same infrastructure can be combined in that way. For instance, our Single Instruction Multiple Data (SIMD) instruction generator has been used for both the generation of Streaming SIMD Extension (SSE) instructions on INTEL processors and to enhance the code generated by the CUDA/Open Computing Language (OpenCL) generator presented in Section 5.5.

More traditional advantages of source-to-source compilers include their ease of debug: the IR can be dumped as a TR at anytime. For the same reason, they are very pedagogical tools and make it easier to illustrate the behavior of a transformation.

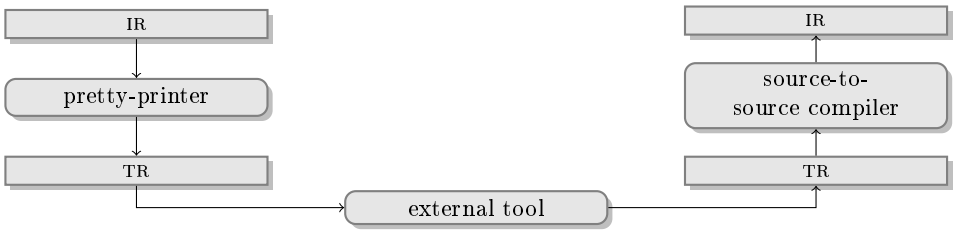

Fig. 1: Source-to-source cooperation with external tools.

3 Model for Code Transformations

The core of a compiler optimizer is the sequencing of code transformations, a.k.a. passes. This section studies from a formal point of view the interaction between passes and elaborates on several transformation composition rules.

Let us give a general definition of what a transformation is. Let $\mathcal{P}$ be the set of *well-formed* programs[5], and $\mathcal{T}$ the set of code transformations. If p is a program, let $\mathcal{V}_{in}(p)$ be the set of its possible input values; and given $v_{in} \in \mathcal{V}_{in}(p)$, let $\mathbf{P}(p, v_{in})$ denotes the results of the evaluation of p.

Definition 1. *A* code transformation *is an application $\mathcal{P} \to \mathcal{P}$ that preserves the semantics of the program, that is to say:*

$$\forall p \in \mathcal{P},\ \forall v_{in} \in \mathcal{V}_{in}(p), \quad \boldsymbol{P}(p, v_{in}) = \boldsymbol{P}(t(p), v_{in})$$

However the former definition of a transformation by-passes an important aspect of code transformations: they can fail. For instance the loop fusion of two loops fails if the second loop carries backward dependencies on the first; a loop-carried backward dependency can prevent loop vectorization and so on, or the pass can even crash because of a lousy implementation. As a consequence, we introduce an *error* state and propose:

[5] We call *well-formed* programs, programs that terminate and whose behavior is not undefined according to the language standards accepted by the compiler.

Definition 2. *A* code transformation *is an application* $\mathcal{P} \to (\mathcal{P} \cup \{error\})$ *that preserves the semantics of the program or fails.*

And one can revert to the previous state by defining a *failsafe* operator:

Definition 3. *The failsafe operator* $\tilde{\cdot} : \mathcal{T} \to \mathcal{T}$ *is defined by*

$$\forall t \in \mathcal{T},\ \forall p \in \mathcal{P}, \quad \tilde{t}(p) = \begin{cases} t(p) & \text{if } t(p) \neq error \\ p & \text{otherwise} \end{cases}$$

and then a failsafe composition:

Definition 4. *The failsafe composition* $\tilde{\circ} : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ *is defined by*

$$\forall t_0, t_1 \in \mathcal{T} \times \mathcal{T}, \quad t_1 \,\tilde{\circ}\, t_0 = \tilde{t_1} \circ \tilde{t_0}$$

Chaining transformations can be done using the $\tilde{\circ}$ operator and most compilers are using this semantics as their primary way to compose transformations. New passes can be defined as the composition of existing passes, enforcing modularity instead of monolithic passes. For instance a pass that generates OpenMP code can be written as the failsafe combination of *loop fusion*, *reduction detection*, *parallelism extraction* and *directive generation* instead of a single *directive generation*. Lattner advocates [20] for this low granularity in pass design.

Still, the fact that a transformation fails carries some important information. In the example above, if loop vectorization succeeds, vector instruction can be generated, otherwise parallelism extraction may be tried. This kind of behavior is represented by a conditional composition:

Definition 5. *The conditional composition* $\underset{\sim}{\circ} : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ *is defined by*

$$\forall t_0, t_1, t_2 \in \mathcal{T} \times \mathcal{T} \times \mathcal{T},\ \forall p \in \mathcal{P} \quad ((t_1, t_2) \underset{\sim}{\circ} t_0)(p) = \begin{cases} (t_1 \circ t_0)(p) & \text{if } t_0(p) \neq error \\ t_2(p) & \text{otherwise} \end{cases}$$

The $\underset{\sim}{\circ}$ operator is not used in Low Level Virtual Machine (LLVM) or GNU C Compiler (GCC), although it provides interesting perspectives. Let us assume the existence of 3 transformations t_{gpu}, t_{sse} and t_{omp} that convert a sequential C code into a code with CUDA calls, SSE intrinsic calls and OpenMP directives, respectively. Then the following expression:

$$(\text{id}_{\mathcal{T}}, t_{\text{sse}} \,\tilde{\circ}\, t_{\text{omp}}) \underset{\sim}{\circ} t_{\text{gpu}}$$

means try to transform the code into a CUDA code followed by the identity transformation or if it fails try to generate OpenMP directives then SSE intrinsics, whether OpenMP directives were generated or not. It builds a decision tree that allows complex compilation strategies.

If the intended behavior is to stop as soon as an error occurs, and to keep on applying transformations otherwise, as in the sequence

$$\big((t_3, \text{id}_{\mathcal{T}}) \underset{\sim}{\circ} t_2, \text{id}_{\mathcal{T}}\big) \underset{\sim}{\circ} \big((t_1, \text{id}_{\mathcal{T}}) \underset{\sim}{\circ} t_0\big)$$

then writing the default skip transformation is bothersome. Thus we define an error propagation operator:

Definition 6. *The error propagation operator* $\vec{\circ} : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$ *is defined by*

$$\forall t_0, t_1 \in \mathcal{T} \times \mathcal{T}, \quad t_1 \,\vec{\circ}\, t_0 = (t_1, id_{\mathcal{T}}) \wr t_0$$

which makes it possible to rewrite the above example as

$$t_3 \,\vec{\circ}\, t_2 \,\vec{\circ}\, t_1 \,\vec{\circ}\, t_0$$

To give a practical example, let us assume the existence of a transformation $t_{\text{opt\_dma}}$ that optimizes the usage of Direct Memory Access (DMA) functions by trying to remove redundant ones, to merge them, etc. It is not relevant to apply it if no DMA function have been generated by a $t_{\text{gen\_dma}}$ transformation. Guessing that $t_{\text{gen\_dma}}$ returns an error if it failed to generate DMA operations, this kind of interaction can be represented using the expression

$$t_{\text{opt\_dma}} \,\vec{\circ}\, t_{\text{gen\_dma}}$$

This model sets the ground of the PyPS pass manager implemented on top of the PIPS source-to-source compiler infrastructure [1].

4 PyPS

Instead of introducing yet another new domain-specific language to express these operators, we chose to leverage existing tools and languages, taking advantage of the similarity with existing control flow operators: indeed the transformation composition is similar to a function definition, the failsafe operator can be implemented using exception handling and the conditional composition performs a branching operation. This leads to the idea of using a general purpose language coupled with an existing compiler infrastructure, while clearly separating the concerns.

4.1 Benefiting from Python: *on the shoulders of giants*

Using a programming language to manage pass interactions offers all the flexibility needed to drive complex compilation processes, without the need of much insight on the actual IR. Conceptually, a scripting language is not required. However, it speeds up the development process without being a burden in terms of performance as all the time is spent in the transformations themselves.

Some approaches introduced dedicated language [33] for the pass management, but we rather follow the well known Bernard of Chartres' motto: "on the shoulders of giants" and thus chose to use Python as our base language. This choice proved to be better than expected by not only providing high-level constructions in the language but also by opening access to a rich ecosystem which widens the set of possibilities.

4.2 Program abstractions

In the model presented in Section 3, transformations process the program as a whole. However, transformations can proceed at lower granularity: *compilation unit level*[6], *function level*[7] or *loop level*:

- at compilation unit level, decisions based upon the target can be made following the rule of thumb "one compilation unit per target". This helps to drives the compilation process by applying different transformations to different compilation units;
- most functions that consider stack-allocated variables work at the function level: *common sub-expression elimination*, *forward substitution* or *partial evaluation* are good examples;
- a lot of optimization focus on loop nests, without taking care of the surrounding statements. This is the case for polyhedral transformations.

Interprocedural transformations, like building the *callgraph*, require knowledge of the whole program, or can be improved by such knowledge (e.g. *constant propagation*), thus the program granularity is still relevant. The class diagram in Figure 2 shows the relations between all these abstractions. These ones—and only these ones—are exposed to the pass manager. The `Maker` in charge of the compilation process is introduced in Section 4.4.

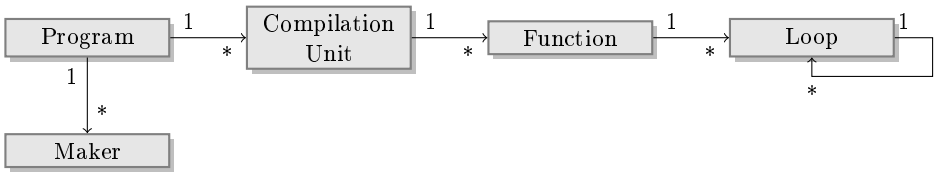

Fig. 2: PyPS class hierarchy.

4.3 Control structures

We now present the main control structures involved in implementing the operators presented in Section 3.

Conditionals Conditionals are used when transformation scheduling depends on user input or on current compilation status. Listing 1.1, extracted from our SIMD Architecture Compiler (SAC) compiler, illustrates the use of conditionals to implement the `-fno-pass-name/-fpass-name` switch as in GCC.

[6] A source file in C.
[7] Also referred as "module level".

Listing 1.1: Conditionals in PYPS.

```
if conditions.get('if_conversion',False):
  module.if_conversion()
```

Listing 1.2: Iteration over selected modules.

```
for kernel in terapix_kernels:
  kernel.microcode_normalize()
```

Listing 1.3: Iteration over inner loops.

```
for l in module.inner_loops(): l.unroll(4)
```

Listing 1.4: Multiple instruction selection.

```
for pattern in ["min","max","adds"]:
  module.pattern_recognition(pattern)
```

Fig. 3: For loop is a control structure commonly involved in PYPS

For Loops For loops are used to perform repetitive tasks, such as in the listings from Figure 3:

1. applying a transformation to each function or loop of a set;
2. applying a transformation iteratively with varying parameters.

Listing 1.2 illustrates a basic iteration over selected modules in our compiler for TERAPIX. Listing 1.3 from our SAC compiler shows how to unroll all inner loops by a factor of four. Finally in our SAC compiler, we also perform early pattern recognition. Listing 1.4 demonstrates the use of loops to apply this pass for various patterns.

Exceptions A PyPS exception can be raised in three situations:

- a user input is invalid;
- a pass fails;
- an internal error (bug) happens.

Each of them has a different meaning and can be caught or propagated depending on pass manager developer objectives. A sample usage is given in Listing 1.5. Listing 1.6 illustrates the error propagation operator presented in Def. 6 on a sequence of two passes in a loop: *dead_code_elimination* is applied if *redundant_load_store_elimination* succeeds.

Listing 1.5: Using exceptions to adapt the compilation process.

```
# Try most efficient parallelization pass first:
try: module.coarse_grain_parallelization()
except:
# or downgrade to loop_distribution
  try: module.loop_distribution()
  except: pass # or do nothing
```

While Loops While loops are useful to achieve a goal within an unknown number of iterations. In our SAC compiler, while loops are combined with exception handling to reach a fix point. Listing 1.6 shows this pattern on a data transfer optimizer.

Listing 1.6: Searching fix point.

```
try:
  while True:
    module.redundant_load_store_elimination()
    module.dead_code_elimination()
except:pass # an exception is raised when nothing is cleaned up
```

4.4 Maker

As code for heterogeneous targets is generated in the form of source code, a final compilation step is needed to get the final binary (or binaries). *Stricto sensu*, this is not the role of the pass manager. However, as it is in charge of the source generation, it has the information concerning which code is to be compiled for which architecture using which specific compiler. Moreover, complex transformations for hybrid or distributed architectures might require the addition of runtime calls. They are frequently shipped as shared library and thus the final compilation step is dependent from the transformations that were done on the code. For instance if some FFT operations using *FFTW3* are automatically transformed in the equivalent GPU operations with *CuFFT*, and the the linking process has to be adjusted. The pass manager can keep track of this. As a consequence, it is useful to provide primitives to drive the final compilation process.

Our proposal is to associate an automatically generated Makefile to the generated sources. This makefile is a template that holds generic rules to compile CUDA/OpenCL/AVX/OpenMP code, and it is filled with the proper dependencies to run the compilation with appropriate compiler/linker flags. This approach offers the advantage of delegating most of the difficulties to third-party tools, in a source-to-source manner.

5 PyPS Applications

To assert our proposal, we exhibit several real-world applications for which the flexibility of the proposed pass manager has proved to be a valuable property. For each scheme, the Source Lines Of Code (SLOC) of its PyPS version is given.

5.1 OpenMP

The OpenMP code generator written in PyPS uses a very classic compilation scheme summarized in Listing 1.7, but still illustrates the basic functionality of the pass manager. Two parallelization algorithms are tried successively and whatever the method, directives are generated.

Listing 1.7: OpenMP compilation scheme.

```
def openmp(module, distribute=True):
  module.reduction_detection() #find reductions
  module.privatize() #find private variables
  try:module.parallelize() #find parallelism or raise exception
  except:if distribute: module.distribute() #loop distribution
  finally:module.generate_omp_pragma() #directive generation
```

The whole compilation scheme takes 38 SLOC.

5.2 TERAPIX

The TERAPIX architecture [4] is a Field Programmable Gate Array (FPGA) based accelerator for image processing developed by THALES. It achieves high throughput and low energy consumption thanks to a highly specialized architecture and a limited Instruction Set Architecture (ISA). In particular it uses Very Long Instruction Word (VLIW) instructions written in a dedicated assembly. The compilation scheme summarized in Listing 1.8 shows how the name of generated functions takes part to the process. An external tool is used to compact the generated code, enforcing tool reuse. The generated makefile calls a C compiler for the host code and a specific assembler for the accelerator.

The whole compilation scheme takes 411 SLOC.

5.3 SAC

SAC is an automatic generator of multimedia instructions that can output either AVX, SSE or NEON intrinsics. It combines polyhedral transformations at the loop level and pattern matching at the function level to generate vector instructions. Listing 1.9 shows an extract of its compilation scheme in charge of tiling as many loops as possible.

The whole compilation schemes takes 220 SLOC.

Listing 1.8: Terapix compilation scheme.

```
def terapix(module):
  counter,kernels=0,list()
  for loop in module.loops():
    try:
      # Generate a new name from parent and a numeric suffix
      kernel=module.name+str(counter)
      loop.parallelize()
      module.outline(label=loop.label,name=kernel)
      # Add the kernel object from its name to our kernel list:
      kernels.append(workspace[kernel])
    except:pass
  for kernel in kernels:
    kernel.terapix_assembly()
    kernel.pipe("vliw_code_compactor")
```

Listing 1.9: SAC compilation scheme extract.

```
def autotiler(loop):
  if loop.loops():
    try: loop.tile(...)
    except: map(autotiler,loop.loops())
  else: loop.unroll(...)
```

5.4 An Iterative Compiler

PyPS makes it easier to build an iterative compiler. Cloning the workspace, iterating over a parameter domain for transformations such as *loop unrolling* is straight-forward. We have also implemented a genetic algorithm that takes advantage of PyPS abstraction to test different transformation scheduling, benchmark the resulting applications and so forth. Additionally, each iteration is embedded in a remote process, *à la* Erlang, which gives a free performance boost on distributed multi-core machines. The whole compiler takes 600 SLOC.

5.5 Par4All

Par4All is an open-source initiative to federate efforts around compilers to allow automatic parallelization of applications to hybrid architectures. It relies heavily on PyPS capabilities. The compilation process is driven by user switches that select different back-ends:

- the OpenMP backend is based on the method showed earlier;
- CUDA and OpenCL back-ends rely on the parallelization algorithms used for OpenMP and add complex handling in the pass management process to address cases like C99 idioms unsupported by the NVIDIA compiler or Fortran

to CUDA interface. The subtle differences between the two back-ends are mostly handle in a runtime. Some others few differences are addressed in the process of pass selection using branches wherever required.
- the SCMP back-end [6] generates task based applications for an heterogeneous MP-SoC architecture designed by CEA for the execution of dynamic or streaming applications [29]. To generate inter-task communications, the compiler relies on a PIPS phase also used by the CUDA back-end. The resulting code is then post-processed by a phase directly coded in Python for further adaptation to a run-time specific to the SCMP architecture.

PAR4ALL is already a big project which ensures the portability of the original source code to many targets with many different options. It's a perfect case study for validating the concepts that we propose within PyPS, and its implementation is actually made easier by the reuse and the easy combination of the different constructs available in PyPS.

PAR4ALL represents around 10K source lines of code, including many other stuff than the pass management. For the pass management part, the core system takes only 71 SLOC, the OpenMP part 16, the SCMP part 38, and the CUDA/OpenCL part 117. This one is bigger than expected because the transformation of Fortran or C99 to CUDA requires many special handling.

5.6 When compilers interact with each other

The heterogeneity of target platforms encourage compilers developers to specialize their project. As a consequence, an ambitious compilation infrastructure such a PAR4ALL cannot rely on a single framework to provide retargetability. PAR4ALL core is based on PIPS framework which provides most of the state-of-the-art transformations. But as it targets a wide range of applications, it misses some automated analyses to drive the transformations in specific cases. For this purpose PAR4ALL aims at collaborating with different source-to-source tools at different parts of the compilation process.

One of these tools is PoCC [25], a polyhedral compiler that includes a sophisticated automated decision process. However it also imposes several limitations on the input code, and the user must insert directives to delimit the parts to process. To handle this automatically and transparently, we have integrated in PyPS a high level method "poccify()" which acts at module or loop level and successively triggers several passes:

1. the static control parts of the code are detected, taking into account most of PoCC restriction [17];
2. directives are generated, surrounding static control part and informing PoCC on the liveness of variables;
3. static control parts are outlined in new functions and new compilation units;
4. the compilation units are passed to PoCC, which optimizes the code for the target architecture;
5. the code resulting from PoCC processing is then held out back to PIPS to be cleaned and inlined at its original place.

Having several compilers cooperating using a classical pass manager would have been difficult. But we managed to handle that in an easy way thanks to the benefit of Python capabilities for handling our operators simply, e.g. with the exception mechanism. The tricky part was about joining the two compilers, but the choice of a general purpose language like Python with embedded subprocess and file management made this integration straightforward.

6 Related work

In traditional compilers such as GCC, a compilation scheme is simply a sequence of transformations chained one after the other and applied iteratively to each function of the original source code [31]. This rigid behavior led to the development of a plugin mechanism motivated by the difficulty to provide additional features to the existing infrastructure. Samples of successful plugins include "dragonegg" and "graphite". GCC's shortcoming led to the development of LLVM [21] that addresses the lack of flexibility by providing an advanced pass manager that can change the transformation chaining at run-time.

The source-to-source ROSE compiler [26,28] does not include any pass manager but provides a full access to all analyses, passes and IR through a clean C++ API. An Haskell wrapper provides scripting facilities. Although this approach offers good flexibility (any kind of pass composition is possible), it offers neither clean concept separation nor any abstraction. This leads to complicated interactions with the compiler. The addition of the scripting wrapper does not solve the problem because it is a raw one-to-one binding of the full API. The POET [33] language, designed to build compilers, suffers from the same drawbacks, as there is no clean separation between compiler internals and pass management: everything is bundled all together.

Compiler directives offer a non-intrusive way to drive the compilation process, much as a pass manager does. They are extensively used in several projects [19,8,9] to generate efficient code targeting multi-cores, GPGPU or multimedia instruction set. They can specify a sequential ordering of passes, but they do not provide extra control flow, and may have complex composition semantics.

Transformation recipes, *i.e.* specialized pass manager implementations, are presented in [14]. The paper emphasizes the need of a common API to manipulate compiler passes. It leads to a LUA based approach proposed in [27] to optimize the generation of CUDA codes. It provides bindings in this scripting language for a limited number of parametric polyhedral loop transformations. The addition of scripting facilities makes it possible to benefit from complex control flow, but the interface is limited to polyhedral transformations. The approach is procedural and the API allows access to analysis results such as control flow graph, dependence graph, etc.

The needs for iterative compilation led to the development of an extension of the GCC pass manager [12]. It provides a GCC plugin that can monitor or replace the GCC pass manager. It is used to apply passes to specific function but the ordering is still sequential. The underlying middleware, ICI, is also used

in [11] where an interactive compiler interface is presented. The authors enforce the need for clear encapsulation and concept separation, and propose an XML based interface to manage the transformation sequencing.

Our approach basically turns a compiler into a program transformation system, which is an active research area. FERMAT [30] focuses on program refinement and composition, but is limited to transformation pipelining. STRATEGO [24] software transformation framework does not allows much more than basic chaining of transformations using pipelines. CIL [23] only provides a flag-based composition system, that is activation or deactivation of passes in a predefined sequence without taking into account any feedback from the processing.

7 Conclusion

In this paper, we introduce PyPS, a pass manager API for the PIPS source-to-source compilation infrastructure. This compilation framework is reusable, flexible and maintainable, which are the three properties required to develop new compilers at a low cost while meeting time-to-market constraints. This is due to a clear separation of concepts between the internal representation, the passes, the consistency manager and the pass manager. This clear separation is not implemented in GCC and is not fully exploited either in research compilers, although pass management is becoming a research topic.

Five specific compilers and other auxiliary tools are implemented using this API. OpenMP, SSE, SCMP and CUDA/OpenCL generators, and an optimizing iterative compiler are used to illustrate how the API helps to meet very different goals.

We show how it is possible to address retargetability by combining different transformation tools in an end-to-end parallelizing compiler for heterogenous architectures. Moreover it is done in an elegant and efficient way relying on the clear separation of concepts that we identified and on facilities offered by Python ecosystem.

Acknowledgments

This work has been partly funded by the French ANR. The authors would like to thank Frédéric PERRIN, Grégoire PAYEN de la GARANDERIE, Adrien GUINET and Sébastien MARTINEZ for their contribution to PyPS, and the HPC Project startup for taking interest into it.

References

1. Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS is not (just) polyhedral software. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, April 2011.

2. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
3. Eduard Ayguade, MarcGonzalez, Marc Gonzalez, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In *In First European Workshop on OpenMP*, 1999.
4. Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gérard Gaillat, Olivier Ruch, and Pascal Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *Design Automation and Test in Europe (DATE'2008)*, pages 610–615. IEEE, 2008.
5. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, 1994.
6. Béatrice Creusillet. Automatic Task Generation on the SCMP architecture for data flow applications. `http://www.par4all.org/documentation/publications`, 2011.
7. Steven Derrien, Daniel Ménard, Kevin Martin, Antoine Floch, Antoine Morvan, Adeel Pasha, Patrice Quinton, Amit Kumar, and Loïc Cloatre. GeCoS: Generic Compiler Suite. `http://gecos.gforge.inria.fr`.
8. Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Garzarán, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 136–151. Springer Berlin / Heidelberg, 2006.
9. CAPS Entreprise. HMPP Workbench. `http://www.caps-entreprise.com/`.
10. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *In Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, PLDI, 1998.
11. Grigori Fursin and Albert Cohen. Building a practical iterative interactive compiler. In *In 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07)*, 2007.
12. Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, 2008.
13. David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
14. Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Khan. Loop transformation recipes for code generation and auto-tuning. In *Languages and Compilers for Parallel Computing*. 2010.
15. Sang ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *16th International Workshop on Languages and Compilers for Parallel Computing*, LCPC, 2003.
16. François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing (ICS)*, pages 244–251, 1991.

17. Dounia Khaldi, Corinne Ancourt, and Francois Irigoin. Automatic C Programs Optimization and Parallelization using the PIPS-PoCC Integration. Technical report, A/448/CRI, MINES Paris-Tech, 2011.
18. Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Conference on Language, Compiler, and Tool for Embedded Systems*, pages 12–23. ACM Press, 2003.
19. Kazuhiro Kusano and Mitsuhisa Sato. A comparison of automatic parallelizing compiler and improvements by compiler directives. In *International Symposium on High Performance Computing*, ISHPC '99, 1999.
20. Chris Lattner. *LLVM*, chapter 11. 2011. `http://www.aosabook.org/`.
21. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, CGO'04, Palo Alto, California, 2004.
22. H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramirez, D. Rodenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunovic, and A. Zaks. Acotes project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, 2010.
23. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, volume 2304 of *Lecture Notes in Computer Science*, April 2002.
24. Karina Olmos, Karina Olmos, Eelco Visser, and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *International Conference on Compiler Construction*, 2005.
25. Louis-Noël Pouchet, Cédric Bastoul, and Uday Bondhugula. PoCC: the Polyhedral Compiler Collection, 2010. `http://pocc.sf.net`.
26. Daniel J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
27. Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Cha Jacqueline. A programming language interface to describe transformations and code generation. In *International conference on Languages and compilers for parallel computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.
28. Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.
29. Nicolas Ventroux and Raphaël David. SCMP architecture: an asymmetric multiprocessor system-on-chip for dynamic applications. In *International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, 2010.
30. Martin P. Ward. Assembler to C Migration Using the FermaT Transformation System. In *ICSM*, pages 67–76, 1999.
31. Wikibooks, editor. *GNU C Compiler Internals*. `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`, 2006-2009.
32. Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 1994.
33. Qing Yi. Automated Programmable Control and Parameterization of Compiler Optimizations. In ACM, editor, *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Chamonix, France, April 2011.