Towards Portable High Performance in Python: Transpilation, High-Level IR, Code Transformations and Compiler Directives

(Unrefereed Workshop Manuscript)

Mateusz Bysiek[1,2,a)] Mohamed Wahib[2] Aleksandr Drozd[1,2]
Satoshi Matsuoka[3,1,2]

Abstract:
We present a method for accelerating the execution of Python programs. We rely on just-in-time automatic code translation and compilation with Python itself being used as a high-level intermediate representation. We also employ performance-oriented code transformations and compiler directives to achieve high performance portability while enabling end users to keep their codebase in pure Python. To evaluate our method, we implement an open-source transpilation framework with an easy-to-use interface that achieves performance better than state-of-the-art methods for accelerating Python.

Keywords: abstract syntax tree, gradual typing, high performance computing, transpilation.

1. Introduction

Transpilation, i.e. translation between programming languages at similar levels of abstraction, has been gaining momentum in computing in recent years. Transpilers rely on an intermediate representation (IR) to translate from source to target language in analogy to how compilers rely on an IR to translate source code into machine code.

The level of abstraction of the IR is very important. On one hand it is supposed to hide details about the target platform for execution. On the other hand it is supposed to not be too abstract. All necessary information provided in the source code should be preserved to enable a faithful translation from the source language to target language, irrespective of their specific levels of abstraction.

1.1 IR Abstraction Level

One of the de facto standard intermediate representations used these days in compilers and transpilers is the LLVM Intermediate Representation [15], used by LLVM compiler infrastructure [8]. LLVM IR is a static, strongly typed and relatively low-level language. It preserves all necessary information from the original source code so that the compiler can convert it to machine code correctly.

Due to some very desirable characteristics such as verifiability and quality of the specification, LLVM IR was adopted by many tools, including those meant for more high-level optimizations such as polyhedral loop transformations aimed at enhancing performance [6].

However, LLVM IR lacks high-level constructs which software authors are used to, therefore it is impractical for domain experts to write LLVM IR manually. Additionally, although it's perfectly natural to generate machine code from human-written source code through LLVM IR, it is not practical to generate source code intended for human consumption from LLVM IR.

In this paper, we argue for using a more high-level IR. The main reason is that LLVM IR is too low-level in many cases. In this paper we present and evaluate those cases and elaborate on our approach of using a high-level intermediate representation to address those cases missed by low-level IRs (such as LLVM IR).

Finally, we are working towards creating a high-level IR to enable a new approach for automated code transformations to optimizing to target architectures.

2. Python as a High-level IR

The IR we are proposing in this paper is Python itself. In the next few paragraphs we explain the reasons behind choosing Python as an IR. The standard Python implementation provided by the Python Software Foundation, the CPython, includes its own abstract syntax tree (AST) format. This AST is used throughout Python for many pur-

[1] Tokyo Institute of Technology
[2] National Institute of Advanced Industrial Science and Technology
[3] RIKEN Center for Computational Science
[a)] bysiek.m.aa@m.titech.ac.jp

poses. In the context of transpilation, this AST preserves all information needed by Python's interpreter to correctly interpret Python code. It is also used by the CPython compiler as intermediate representation for compiling Python code into Python bytecode. Next, the bytecode is fed into the CPython interpreter.

One remarkable feature of the CPython AST is that one can access and alter the AST of the executed program at runtime. Additionally, one can create an entirely synthetic AST. Regardless of the way it was created, the CPython AST can also be compiled and executed while the program it represents is running. This makes dynamic instruction generation as well as dynamic software self-alteration at runtime a possibility.

Another remarkable feature of the CPython AST is that it is very high-level. It does not preserve details about the source code formatting, however it preserves all high-level structures such as loops, branching ...etc. This makes it in essence an object-oriented representation of the code. As a result, it is relatively simple to generate the code from the AST, i.e. unparse the AST, and in fact Python code generation is one of the things already available in Python.

2.1 Python and Type Information

CPython's AST is originally designed for a fully dynamically typed language (i.e. Python), therefore it did not contain any way to store type information of variables and other constructs. This characteristic of being dynamic remains at the core of Python today and there are no signs that the Python will transform into a statically typed language.

However, in recent years, Python's syntax has been changing to accommodate the popular demand to be able to convey type information in source code. Although Python is not without performance issues, the demand was driven by the need to document and debug the code rather than by any performance-related reasons. Even if providing this information is completely optional and unnecessary as far as the Python interpreter execution is concerned, the concept of type hints [13,14] emerged in Python 3.5 and is being refined further in each new version including the most recent Python 3.7 [7].

The AST, as now observed in Python 3, reflects those ongoing changes, and thus in recent Python versions the type information hints in the source code are preserved in the AST.

2.2 Python Type Hints

The type hints are a way to convey static type information in Python. Although they were first introduced in Python 3.5, they are backwards-compatible even with Python 2.

Type hints can be applied in three forms.

(1) type comments: even in Python 2,

```
spam = 0  # type: int
def ham(eggs):
    # type: (float) -> str
    pass
```

(2) type annotations for functions: in any Python 3 code,

```
def ham(eggs: float) -> str:
    pass
```

(3) type annotations for locals: since Python 3.6.

```
spam: int = 0
```

Although type hints can aid in optimization, to our knowledge, they have not yet been adopted by any performance-oriented packages except in our framework (we describe the implementation in Section 4). Type hints are a fundamental concept we base our approach on.

2.3 IR in Multilingual Transpilation

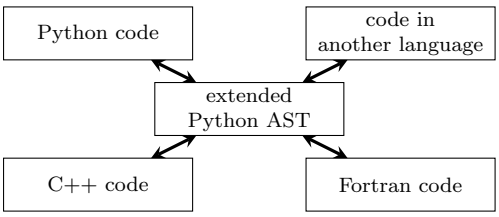

Figure 1 Two-way transformation between the extended Python AST and each of supported languages has to be implemented

In this work, we use CPython AST as the IR for transpilation between more than two programming languages.

The basic requirement for an AST to be practically usable in transpilation is for it to have the capability to convey all necessary information required to faithfully preserving the functionality of the code.

This requirement is already an restrictive, specially if we consider that this requirement has to be satisfied for more than one programming language at the same time. The amount of details needed to be preserved for Python alone may not be so large. However, if we consider the diversity of syntactic structures and idioms in C++, Fortran, CUDA, and possibly other languages we would like to support, it might seem infeasible to try to represent all of the languages using a single IR.

2.4 Transpiler with Human-readable Output

On top of the above requirement we impose on ourselves the requirement of preserving high-level structure of the code, as well as most of the human-required properties of it, such as the naming of variables and comments. With regards to code formatting, we decided to take the approach of generating code that is as readable as possible. However, because what is considered a reasonable indentation and recommended code style changes between programming languages, we did not make an effort to preserve original indentation style. Instead, our approach is to generate code which is objectively readable in a given programming language.

Most notably, Python has very strict rules about indentation. The indentation itself affects control flow, therefore if we would generate code while preserving the original formatting, we might end up with a completely different software.

Conversely, although we assume that the syntax of the input code is correct, we do not assume that its formatting is ideal. Therefore, we are of the opinion that by not preserving the original formatting we avoid some problems while not introducing any new ones, including any possible negative impact on readability.

2.5 Need for Lightweight and Dynamic Tool

The most notable framework for multi-language transpilation is ROSE compiler [11]. The ROSE compiler framework has a very large toolkit [12]. It defines its own abstract syntax tree format. Using this format as the intermediate representation, ROSE provides various tools for inlining and outlining code, as well as tools for generating control flow graphs and call graphs.

Moreover, there are more advanced tools for automatic parallelization using directives [9, 10], as well as loop transformations.

Although comprehensive, ROSE is a complex framework, and has a steep learning curve. In addition, as ROSE is dealing with statically typed compiled languages, its tools are static in nature. ROSE would take the complete application code as input, analyse and transform it, and output the result.

Python, on the other hand, has potential to be very lightweight and still powerful enough to enable multi-language transpilation for relevant subsets of the language. In addition, in Python one can restrict the code analysis even to a single expression, which gives a very fine-grained control over what is analysed and transformed.

It is important to mention that Python can perform the analysis of the code statically, before execution, as well as at any point during the execution. This gives the ability to dynamically adapt the analysis according to the characteristics of the data: floating point precision and value ranges, observed simulation error, or effectiveness of given set of hyper-parameters in deep learning applications.

2.6 Conveying Directives

C++	`#pragma omp parallel for`
Fortran	`!£omp parallel do`
Python	`# omp parallel for`

In most notable HPC programming languages, Fortran and C/C++, the OpenMP directives [2] are conveyed by comments and pragma directives, respectively. Therefore, in our AST, we preserve comments as well as any directives such as macros or include directives without expanding them.

Perform a complete type analysis requires the expansion of the macros and including directives. However, it can be done in such a way as to not expose the expanded code to the user, as long as no transformations occur in the expanded code. Not expanding the macros or include directives is also essential to preserve the clarity of the code.

3. Defining Language Subsets

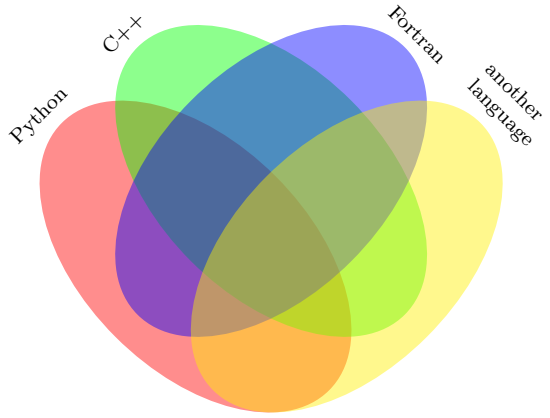

Figure 2 Each distinct area of the above illustration contains some constructs from given programming languages, and presents different challenges when attempting to translate

It is important to note that transpilation is not without its challenges. The most notable challenges, which can sometimes prohibit transpilation almost entirely, is dependency on external libraries which are not open source, as well as extensive use of custom user-defined types.

The dependency on external libraries, however, is only a critical problem when translating between different languages. When performing transformations without translating to a different language, it can diminish the positive effects of optimization, however in principle the code can still be analysed and transformed without issues.

Such dependencies occur mainly outside of kernels in case of numerical applications, which as we show below, is a very favourable condition for the success of our approach.

Custom user-defined types provide a convenient abstraction for users of a given application or library, however they are a challenge for performance-oriented transpilers. That is because each abstraction has different assumptions about how data is handled and processed. When those assumptions lead to sub-optimal operations on data, such as lack of data locality, excessive data movement and such, it becomes prohibitive to improve upon those properties of the application without stripping out the abstraction entirely.

This, of course, might not be an issue for a transpiler that does not aim to generate code intended for human use. As long as no one will read the generated code, all abstractions can be stripped away at will. For those transpilers that seek to generate human-readable code, as in our approach, we make an assumption that only the specifically allowed data types can be used in the areas which are transformed, because only for those types we can know what transformations can and cannot be performed.

We came to the conclusion that such two assumptions: (1) lack of or transparent dependence on external libraries, as well as (2) usage of predictable data types, are correct preconditions for the transpilation of performance-critical parts of scientific applications.

3.1 Mapping Between Fortran and Python

We explored this topic already in previous work [1], in which we presented the idea of two-way transpilation between Python and Fortran.

In that work we define the mapping between selected subsets of Python 3 and Fortran 77/95, in order to improve performance of numerical kernels written in pure Python beyond what is achievable using any state-of-the-art tools. In addition, this provides a means to migrate legacy Fortran applications to Python while preserving their performance.

Encouraged by promising results on a set of small benchmark problems, we have decided to apply our approach in more complex scenarios. We adapted our approach to a wider audience by relying on extensions of tje Python AST instead of our own custom AST. We do this by delegating as much basic operations as possible to open-source external tools and libraries. Finally, the whole framework is released as open-source to the community.

3.2 Mapping Between C++ and Python

To translate between C++ and Python, we need to define the mapping between Python and C++. Very similarly to the discussion in our previous paper, we focus on the following areas: (1) data types, (2) basic syntactic structures, (3) selected common idioms.

3.2.1 Data Types Mapping

First, we introduce the type mapping we adopted in our framework.

Python 3	C++
`str`	`std::string`
`int`	`int`
`np.int32`	`int32_t`
`np.int64`	`int64_t`
`float`	`float`
`np.double`	`double`

Table 1 Excerpt of mapping of types between Python and C++

There are many more types we have mapped, such as different precisions of floating point numbers as well as various container types (lists, sets, dictionaries) and arrays types. We are however still experimenting with many of the details of the mapping.

3.2.2 Basic Syntactic Structures

Second, we describe the basic syntax that we consider for transpilation. The list below, as the list of types above, is not exhaustive.

Function declarations

Python
```
def add(a: int, b: int) -> int:
    ...
```

C++
```
int add(int a, int b);
```

In Python, it is syntactically valid to use an ellipsis (three dots) as an expression. It is also valid to make a statement out of any expression.

Therefore, the ellipsis symbol, when used as a function body, can very well denote that the body of the function is missing, and such do-nothing function becomes the declaration.

Function definitions

Python
```
def add(a: int, b: int) -> int:
    return a + b
```

C++
```
int add(int a, int b) {
    return a + b;
}
```

The function definitions in basic form are trivial to translate.

Loops

Python
```
while True:
    continue
```

C++
```
while (true)
    continue;
```

The while loops can be translated as-is.

Python
```
for i in range(0, 10):  # type: int
    pass
```

C++
```
for (int i=0; i < 10; ++i) { }
```

On the other hand, the `for` loops have certain limitations. In C++, the for loop condition can be arbitrary and so can be the expression evaluated at the end of the iteration, as well as the loop initialization. Therefore, when looping over numbers we restrict the allowed form of the loop to the basic form.

Python
```
for number in list_of_floats:  # type: float
    pass
```

C++
```
for (float my_number : list_of_floats) { }
```

In the case of iterating over lists or other containers, the translation is more straightforward.

3.2.3 Selected idioms

Below, we briefly introduce some of the idioms in Python and C++.

Printing

Python
```
print('error:', message, file=sys.stderr)
```

C++
```
std::cerr << "error: " << message << std::endl;
```

Pointer arithmetic

As pointers do not exist in Python, the pointer arithmetic can be preserved in extended CPython AST but unparsing such AST into Python is not feasible. Therefore using bare pointers in C++ prevents the transpiler from operating. However, C++ managed pointers are allowed as these behave similarly to Python references in many contexts.

Python
```
x = MyType(5)
print(x)
```

C++
```
std::shared_ptr<MyType> x =
    std::make_shared<MyType>(5);
std::cout << *x << std::endl;
```

4. Implementation

We implement a framework which demonstrates all the concepts described above. It's called *transpyle*. The implementation is fully open source, and is available on GitHub[*1]. The basic design of *transpyle* is that the framework is responsible for transforming the Python AST. It is also orchestrating the work of parsing source code from various languages into Python AST, as well as unparsing the Python AST into various languages. Each language other than Python has a language-specific AST format, which needs to be converted into the Python AST. However the work required to parse/unparse languages is delegated to libraries that perform the actual work. We have selected modules that offer access to ASTs very close to parse trees, where all details, even comments, are preserved.

All the dependencies necessary to run the framework are also free and open source.

4.1 Dependencies

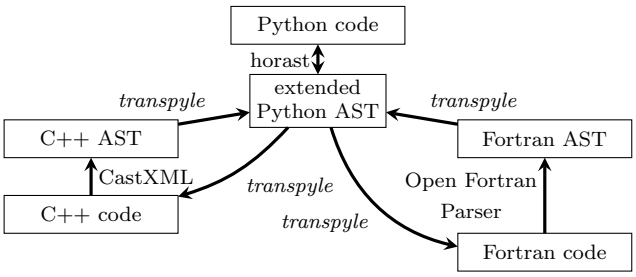

Figure 3 Diagram showing responsibilities of various modules used in our transpilation framework

To enable Fortran support, we rely on Open Fortran Parser for parsing Fortran into an XML representation and we use f2py for creating Python interface for compiled Fortran code. To enable C++ support we use CastXML for parsing C++ into an XML parse tree, and we utilize SWIG for making an interface between compiled C++ code and Python. We rely on GNU Compiler Collection to compile both C++ and Fortran, however in principle any modern compiler of those languages can be used. Additionally, we use several supporting Python modules, some of which we implemented ourselves: typed-ast, horast, typed-astunparse, static-typing, pcpp, pycparser, and others. Due to significant amount of dependencies we provide a Docker image where all the dependencies are preconfigured.

4.2 Gradual Performance

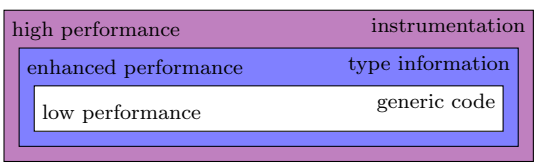

Figure 4 Simple diagram showing performance levels achievable while gradually building upon the original generic code.

[*1] https://github.com/mbdevpl/transpyle

We introduce the concept of gradual performance, as a way of non-intrusive performance improvement of code. Gradual performance is a kind of step-by-step improvement of the performance of the code. In this kind of improvement, one can instrument the existing code with performance-enabling instructions in a way that can be parametrized depending on any data available statically or dynamically. Static data can be the current execution architecture, and dynamic data can be the array size in array operations.

We illustrate the concept of gradual performance using a motivating example. Let us consider a naive pure Python implementation of a element-wise vector summation.

```
def add(arr1, arr2):
    assert len(arr1) == len(arr2)
    arr3 = np.ndarray((arr1.size,), dtype=float)
    for i in range(0, arr1.size):
        arr3[i] = arr1[i] + arr2[i]
    return arr3
```

How can we improve the performance of the code without rewriting it?

4.2.1 Providing Static Type Information

First, we annotate types in the above code. This type information, although parsed by the parser of the Python interpreter, will be ignored by the interpreter at execution time.

```
def add(arr1: st.ndarray[1, float],
        arr2: st.ndarray[1, float]
        ) -> st.ndarray[1, float]:
    assert len(arr1) == len(arr2)
    arr3 = np.ndarray((arr1.size,), dtype=float
                      )  # type: st.ndarray[1, float]
    for i in range(0, arr1.size):  # type: int
        arr3[i] = arr1[i] + arr2[i]
    return arr3
```

4.2.2 Transforming and Transpiling

Second, we add the decorator `@transpyle.vectorize('i', 4)` to our function which will vectorize the loop indexed with 'i' at the granularity of four, an additional decorator `@transpyle.transpile('Fortran')` which will translate the vectorized code of the function into Fortran 95, compile it, and substitute the Python version with the compiled Fortran version.

Finally, we also add a comment `# omp parallel for`. It is a basic OpenMP directive which lets the loop workload to be automatically distributed across all processor cores. This directive will be dormant unless the code is transpiled, therefore the top-most decorator is necessary for the directive to have any effect.

The final code will be as follows:

```
@transpyle.transpile('Fortran')
@transpyle.vectorize('i', 4)
def add(arr1: st.ndarray[1, float],
```

```
        arr2: st.ndarray[1, float]
        ) -> st.ndarray[1, float]:
    assert len(arr1) == len(arr2)
    arr3 = np.ndarray((arr1.size,), dtype=float
                ) # type: st.ndarray[1, float]
    # opm parallel for
    for i in range(0, arr1.size):  # type: int
        arr3[i] = arr1[i] + arr2[i]
    return arr3
```

4.3 Transformations

We present details about the transformations available in our framework.

4.3.1 Inlining

Inlining is a basic optimization. Below example is an illustration of inlining with basic addition operation in a element-wise vector addition function.

```
def add(a, b):
    return a + b

def elementwise_add(arr1, arr2):
    assert len(arr1) == len(arr2)
    arr3 = np.array()
    for i in range(len(arr1)):
        arr3[i] = add(arr1[i], arr2[i])
    return arr3

elementwise_add_inlined = \
    trasnpyle.inline(elementwise_add, add)
```

As a result of inlining of `add`, the following function is obtained:

```
def elementwise_add_inlined(arr1, arr2):
    assert len(arr1) == len(arr2)
    arr3 = np.array()
    for i in range(len(arr1)):
        arr3[i] = arr1[i] + arr2[i]
    return arr3
```

However, the true benefits of this operation show themselves only when we consider additional transformation. Instead of a naive addition operation, we could, at runtime inline a call to architecture-specific and precision-specific implementation. Such specialized implementations are available on most modern platforms for a multitude of specific data types and provide highest performance.

4.3.2 Loop unrolling

Loop unrolling reduces the overhead of iteration and can be beneficial even for pure Python codes.

Let us consider again the code for element-wise vector addition.

```
@transpyle.unroll('i', 4)
def elementwise_add(arr1, arr2):
    assert len(arr1) == len(arr2)
    arr3 = np.array((arr1.size,), dtype=float)
    for i in range(0, len(arr1)):
        arr3[i] = arr1[i] + arr2[i]
    return arr3
```

This result of unrolling the loop at granularity four is shown below.

```
def elementwise_add_unrolled(arr1, arr2):
    assert len(arr1) == len(arr2)
    arr3 = np.array((arr1.size,), dtype=float)
    for i in range(0, len(arr1), 4):
        arr3[i] = arr1[i] + arr2[i]
        arr3[i + 1] = arr1[i + 1] + arr2[i + 1]
        arr3[i + 2] = arr1[i + 2] + arr2[i + 2]
        arr3[i + 3] = arr1[i + 3] + arr2[i + 3]
    return arr3
```

The unrolled loop body can be also further refined. We can substitute it with a call to a routine that takes advantage of vector extensions to the instruction sets of modern processors. We can do it provided that the aim is to transpile the function to C++ or Fortran.

5. Use Cases

In this section, we summarize some of the use cases we identified for our approach.

5.1 Use Case 1: High-performing HPC prototypes

The readability of code achieved in *transpyle* framework is useful in its own right. Because the code generated by our framework is readable and preserves the original structure, naming and comments, it can be further hand-tuned with relative ease.

In certain cases, relying on automatic optimizations performed by the compiler does not yield satisfying performance results. In such cases, having a low-level implementation is an advantage, because hand-tuning can be only performed if the final C, C++ or Fortran code to be compiled is available.

As our framework generates such low-level code, and the code can be inspected and modified at any time, the altered low-level implementation can used instead of the one generated by our transpiler.

5.2 Use Case 2: Legacy Fortran CFD Optimization

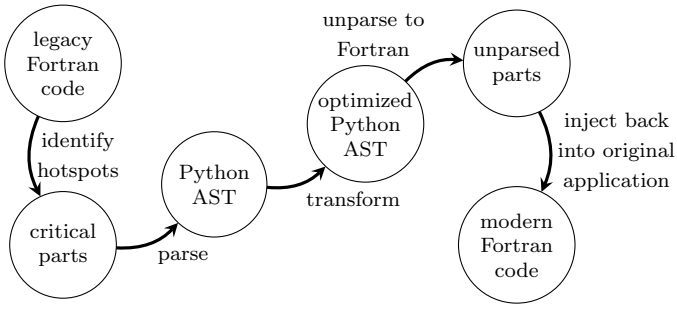

Figure 5 High-level overview of optimization process of the FLASH code.

A scientific simulation framework FLASH [5] is a mature scientific library being developed at The Flash Center for Computational Science established in 1997 at the University of Chicago. It is implemented in Fortran and suffers from what is a common condition of legacy code: outdated code that does not work well on modern architectures.

FLASH is a computational fluid dynamics framework designed for simulating thermonuclear flashes and is used by scientists in fields related to cosmology around the globe. Several efforts has been made already to port the FLASH code partially to CUDA, or add OpenMP support in certain modules.

Although designed to be modular and massively parallel [4], currently the FLASH framework still relies entirely on Message Passing Interface (MPI) for its parallelism, which becomes a bottleneck on many-core architectures.

Our transpilation framework enables legacy codebases like FLASH framework to be automatically adapted to new multi- and many-core architectures.

5.3 Use Case 3: Deep Learning Auto-tuning

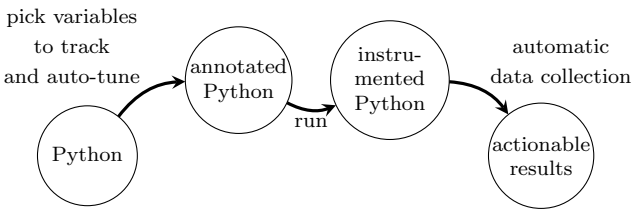

Figure 6 High-level overview of how data collection in protoNN works

The framework protoNN [3] is being currently implemented. It will enable auto-tuning of Python-based deep learning models, as well as transparent and elastic scheduling of deep neural network (DNN) training jobs on modern HPC systems.

One of the features enabling this auto-tuning is ability to dynamically alter the running code originally implemented for the *transpyle* framework. protoNN uses Python type hints to specify which parameters need to be tracked or modified, and our transpiler framework picks up those hints and dynamically instruments the code as necessary.

6. Conclusion

We introduced the concept of using a abstract syntax tree (AST) format provided in Python as high-level intermediate representation (IR) for multi-language dynamic transpilation enabling code transformations. We presented a detailed account of our approach, and contrasted it with current approaches exhibited in LLVM IR and ROSE compiler.

We also described selected details of implementation of the framework *transpyle*, which is a demonstration of the concepts we introduced and is an ongoing open-source project that can be found on GitHub.

Finally, we discussed the benefits of having high-level IR with presented characteristics in multitude of scenarios, including writing HPC code prototypes which are simpler to hand-tune, optimization of legacy Fortran computational fluid dynamics codes and last but not least auto-tuning of deep learning codes.

References

[1] Bysiek, M., Drozd, A. and Matsuoka, S.: Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance Through Transpilation and Type Hints, *Proceedings of 6th Workshop on Python for High-Performance and Scientific Computing*, Piscataway, NJ, USA, IEEE Press, pp. 9–18 (online), DOI: 10.1109/PyHPC.2016.12 (2016).

[2] Dagum, L. and Menon, R.: OpenMP: an industry standard API for shared-memory programming, *IEEE computational science and engineering*, Vol. 5, No. 1, pp. 46–55 (1998).

[3] Drozd, A., Wahib, M., Bysiek, M. and Shpakovich, M.: protoNN (2018).

[4] Dubey, A., Antypas, K., Ganapathy, M. K., Reid, L. B., Riley, K., Sheeler, D., Siegel, A. and Weide, K.: Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code, *Parallel Computing*, Vol. 35, No. 10-11, pp. 512–522 (2009).

[5] Fryxell, B., Olson, K., Ricker, P., Timmes, F., Zingale, M., Lamb, D., MacNeice, P., Rosner, R., Truran, J. and Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, *The Astrophysical Journal Supplement Series*, Vol. 131, No. 1, p. 273 (2000).

[6] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A. and Pouchet, L.-N.: Polly-Polyhedral optimization in LLVM, *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011, p. 1 (2011).

[7] Langa, .: PEP 563 – Postponed Evaluation of Annotations (2017).

[8] Lattner, C. and Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, p. 75 (2004).

[9] Liao, C., Quinlan, D. J., Willcock, J. J. and Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore, *International Workshop on OpenMP*, Springer, pp. 28–41 (2009).

[10] Liao, C., Quinlan, D. J., Willcock, J. J. and Panas, T.: Semantic-aware automatic parallelization of modern applications using high-level abstractions, *International Journal of Parallel Programming*, Vol. 38, No. 5-6, pp. 361–378 (2010).

[11] Quinlan, D.: ROSE: Compiler support for object-oriented frameworks, *Parallel Processing Letters*, Vol. 10, No. 02n03, pp. 215–226 (2000).

[12] Quinlan, D., Liao, C., Too, J., Matzke, R. P., Schordan, M. and Lin, P.: ROSE compiler infrastructure (2012).

[13] van Rossum, G., Lehtosalo, J. and Langa, .: PEP 484 – Type Hints (2014).

[14] van Rossum, G. and Levkivskyi, I.: PEP 483 – The Theory of Type Hints (2014).

[15] Zhao, J., Nagarakatte, S., Martin, M. M. and Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations, *Acm sigplan notices*, Vol. 47, No. 1, ACM, pp. 427–440 (2012).