Dynjit: A General-purpose Just-In-Time Compiler for Python Programming Language

システム情報工学研究科　情報理工学位プログラム
202020696 Wanghongxuan Zhao

コンピュータサイエンス専攻
指導教員 亀山幸義

***Abstract*—Python has been a popular programming language, and widely used in various fields such as scientific computing, web applications, data analysis and machine learning. However, its popularity worsens the performance issues of computation-intensive Python programs. To address this, many attempts for Python Just-In-Time(JIT) compilers have been made during the last 30 years. These attempts are either domain-specific, CPython-incompatible, or never finished. After taking lessons from previous attempts, inspired by recent JIT successes for dynamic programming languages, and observing the progress of Python code analysis tools and Python-To-C code generators, we designed and prototyped Dynjit, a general-purpose JIT compiler for Python programming language. Dynjit is also CPython-compatible, hence the compatibility for CPython C extensions are kept. Dynjit is lightweight and extensible, as a consequence, it's feasible to introduce Dynjit of the early stage into existing codebase, and gradually producing better performance as the features being extended. Several benchmarks are performed. It was observed that the performance usually increases to 1.5 times to 10 times, depending on the scenarios. Dynjit can make many kinds of common abstractions zero-cost.**

***Index Terms*—Just-In-Time(JIT) compiler, partial evaluation, type specialization**

I. Introduction

Recent years, Python has become one of the most popular programming languages. As a consequence, many industries or research institutes started writing computation-intensive programs in Python, for fast prototyping, rich ecosystem and better maintainability. Besides, a range of other programming tasks involving heavy computations, including (Web) Backend, Graphic User Interfaces(GUI), and Games, are also proliferating, and hence suffering from Python performance issues.

To address this, the direct and existing approach is writing C extensions for Python. There are many famous C extensions, such as Numba and PyTorch. However, writing extensions will cause the "two-language problem" [1], which is likely to damage performance, maintainability, readability and raise the difficulty of coding.

Another approach is using compilers to optimize Python itself. Static compilers used by static programming languages can produce higher performance code, but Python's dynamic nature rejects static compilation. As a result, we have to apply runtime compilation, a.k.a. Just-In-Time(JIT) compilation.

Attempts to developing a practical Python JIT compiler have been made by many commercial companies, open source communities or individual developers/researches. The major attempts are Pyjion [2], Pyston [3], PyPy [4], Numba [5] and HOPE [6].

However, all these attempts are not sufficient for the real world use. Numba and HOPE are restricted to numeric computing, and have issues for being mixed with pure Python features. Pyjion and Pyston have stopped active development for many years, as it turned out that it's too difficult for them to collect human resources and funds to complish their huge implementations. PyPy has survived the longest and become the most successful alternative Python implementation, however, PyPy's issues of supporting existing C extensions prevent it from being widespread, because there are already many widely used C extensions in the de facto Python implementation, CPython.

The performance overhead of Python remains a long-term problem, but this situation has essentially changed in the recent years.

The new situation comes from 2 parts. Firstly, a few Python frameworks that could be used as components of building a JIT compiler, have become mature. For instance, a JIT needs a compiler in runtime, and Cython [7], the most widely used Python-to-C compiler, can be leveraged to avoid generating C/C++ code, managing C compiler environment, and also simplify the works of properly calling Python C APIs. Secondly, in recent years there are several successful attempts to adding JIT to other dynamically typed programming languages, especially for Julia, which is dynamically typed but can even produce code as efficient as C. These successes are inspiring to us.

In the new situation, we reviewed the previous Python JIT attempts, and found out 4 key points.

1) *Approaches that don't require patching CPython are probably preferable*. This becomes generally agreed and explicitly stated by Pyjion and Pyston.
2) A general-purpose JIT is expected. Domain-specific JITs such as Numba and HOPE were rarely used, and helpful to limited kinds of Python performance bottlenecks.
3) A CPython-compatible JIT is expected. PyPy's incompatibility against CPython causes issues for C extensions.
4) A JIT implementation that is lightweight but extensible is expected. Many Python JIT projects tried making a large codebase initially, and when they died, even with a large codebase, they still cannot provide some basic uses.

We hereby present Dynjit.

1) Type specialization and static dispatch
2) Control flow optimization
3) Split for union types and boolean values

II. Dynjit Architecture

A. Core CPY IR

To optimize Python programs, the first job is the retrieval of code semantics. At least before 3.10, there is no reliable way in Python to retrieve the Python source code or Abstract Syntax Trees. The severe reality forced us to figure out an approach to retrieve code semantics from runtime objects. This turns out to be valid, as a Python user-defined functions holds a code object. This code object contains Python bytecode instructions, which decides the behaviors of CPython stack virtual machine while calling the function.

intermediate representation called Core CPY is developed, to make following code analysis easier. There're only 11 instructions in Core CPY, while Python bytecode has more than 119 instructions after Python 3.8. The reason for avoiding large instruction set is, the following code analysis needs one pattern matching case for each instruction.

The semantics for Core CPY is similar to Python bytecode, which is operated by a stack virtual machine.

$$
\begin{aligned}
\langle \text{instr} \rangle \quad ::= \quad & \textit{CONST}\ \textbf{constant} \\
\mid\ & \textit{LOAD}\ \textbf{symbol} \\
\mid\ & \textit{STORE}\ \textbf{symbol} \\
\mid\ & \textit{JUMP}\ \textbf{int} \\
\mid\ & \textit{CALL}\ \textbf{int} \\
\mid\ & \textit{ROT}\ \textbf{int} \\
\mid\ & \textit{PEEK}\ \textbf{int} \\
\mid\ & \textit{POP} \\
\mid\ & \textit{RETURN} \\
\mid\ & \textit{JUMP\_IF}\ \textbf{bool bool int}
\end{aligned}
$$

Part of the semantics of Core CPY are given below. Other instructions are omitted to avoid the introduction of heavy prerequisites.

$$\frac{\_, \overbrace{\cdots}^{S'} = S}{S \vdash_{instr} \textbf{Pop} \Rightarrow S'}$$

$$\frac{a_1, \cdots, a_i, a_{peek}, \cdots = S}{S \vdash_{instr} \textbf{Peek}\ i \Rightarrow a_{peek}, \overbrace{\cdots}^{S}}$$

Core CPY is the input of the partial evaluation.

B. Partial Evaluation and Configurations

Partial evaluation is the center of our research. During partial evaluation, variables or literals are expressed in the form of a pair **(repr, shape)**, and this pair is called abstract value. An abstract value actually corresponds to a real object at runtime.

A **repr** can be static or dynamic. When it's static, it holds the real Python runtime objects, and use of the static repr may result in constant propagation; if it's dynamic, it refers to a named or temporary variable.

```
type repr =
| S of python_object
| D of var_slot
```

A **shape** corresponds to a unique Python type, and describes the common properties of the abstract value, such as static methods and instance methods of the corresponding python object.

```
type shape =
| TopS
| TupleS    of shape list
| UnionS    of shape list
| ClosureS  of shape * python_function_object
| NominalS  of typename * static_methods_dict * ...
| ...
```

A user-defined type will correspond to a **NominalS shape**.

Partial evaluation will be performed to every Python function to JIT. This starts from the beginning of the function body, and the code is in Core CPY form, and output the $\textbf{Dynjit}^{tree+goto/label}$ IR. We also need an abstract stack to hold all abstract values, and evaluate Core CPY following its original semantics.

The offset of a Core CPY instruction is called program point, and a pair of program point and the current abstract stack(holding all abstract values in the environment) is called **configuration**.

The **configuration** is crucial, because partial evaluation can be regarded as the transitions from one configuration to another, and any configuration will decide the following configuration sequence. Besides, one configuration will produce a "basic block" of $\textbf{Dynjit}^{tree+goto/label}$ IR.

A use of **configuration** is to prove the termination of the partial evaluation, because once configurations are finite, the partial evaluation must stop. The finiteness of configurations is achieved by restricting the maximum depth of the recursive **shape**s, such as **TupleS** and **ClosureS**. Nested **shape**s exceeding the maximum depth will become **TopS**, to which we can performance no optimizations.

The partial evaluation stops when all reachable configurations are reached, where the shapes/types of abstract values in each configuration are successfully inferred. This means we've successfully typed the programs, though part of the abstract values take **TopS shape**s.

C. Type Specialization

Notice that nested partial evaluations can be performed for inner function calls inside the current partial evaluation. This helps us infer arguments types, or accurately, argument **shape**s of an inner function call. By knowing the argument **shape**s,

specialized methods can be chosen, this is our adaption to Type Specialization.

Specifically, if the inner called function is a builtin Python function, special rules can be written for them to generate method specializations. For instance, if we know the left and right operands of **+** are both **integer**(represented in form of **NomS integer ...**), then we can replace the generic operator **+** with **int_int_add**, if **int_add_add** has been implemented.

D. $\mathbf{Dynjit}^{tree+goto/label}$ *IR and* $\mathbf{Dynjit}^{flatten}$

```
type expr =
| A of abstract_value
| E of expr * expr list


type tree = (* Dynjit^{tree+goto/label} *)
| Assign of var_slot option * expr
| Goto   of label
| Label  of label
| Return of expr
| If     of expr * tree list
                 * tree list
| TyChk  of expr * shape
                 * tree list
                 * tree list


type flat = (* Dynjit^{flatten} *)
| Assign of var_slot option * expr
| Goto   of label
| Label  of label
| Return of expr
| If     of expr * label
| TyChk  of expr * shape * label
```

$\mathbf{Dynjit}^{tree+goto/label}$ is the result of partial evaluation, and can be simply converted to $\mathbf{Dynjit}^{flatten}$ later. $\mathbf{Dynjit}^{flatten}$ is used for generating C/C++/Cython code.

E. Union Type Split

During partial evaluation, calling another function might return an abstract value which holds union **shape**s(**UnionS**).

To optimize code in this case, we learn from the Julia programming language.

The idea can be demonstrated with the following code.

Original Python Code

```
x = function_returns_int_or_float(...)
y = x + x
...
```

Union-split Python Code

```
x = function_returns_int_or_float(...)
if isinstance(x, int):
    y = int_int_add(x, x)
    ...
elif isinstance(x, float):
    y = float_float_add(x, x)
    ...
```

F. Boolean Value Split

Boolean values will be specially handled. Specifically, if an abstract value is inferred to have boolean shape, and the **repr** of the abstract value is dynamic, we will split the configuration to 2 new ones, one's **repr** of the original abstract value is a static **True**, and the other's is a static **False**.

Boolean value split is for optimizing the control flows. **JUMP_IF** instruction in Core CPY can be optimized the condition of jump can be decided statically.

The idea can be demonstrated with the following code.

Original Python Code

```
x = f() # x is inferred to be a boolean
y = g() # y is inferred to be a boolean

if x and y:
    s1
elif x or y:
    s2
else:
    s3
```

Boolean-split Python Code

```
x = f() # x is inferred to be a boolean
if x is True:
    y = g() # y is inferred to be a boolean
    if y is True:
        s1
    else:
        s2
else:
```

```
y = g()
if y is True:
    s2
else:
    s3
```

ACKNOWLEDGMENT

REFERENCES

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[2] Brett Cannon and Dino Viehland. *Pyjion - A JIT for Python based upon CoreCLR*. https://github.com/microsoft/Pyjion, 2018.

[3] Kevin Modzelewski et al. *An open-source Python implementation using JIT techniques*. https://github.com/pyston/pyston, 2017.

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[5] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[6] Joël Akeret, Lukas Gamper, Adam Amara, and Alexandre Refregier. Hope: A python just-in-time compiler for astrophysical computations. *Astronomy and Computing*, 10:1–8, 2015.

[7] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.