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*—Since the last decade, Python has been a most popular programming language, and widely used in various fields. However, its popularity worsens the performance issues of computation-intensive Python programs. To address this, many attempts about 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 have designed and prototyped Dynjit, a general-purpose JIT compiler for Python programming language. Dynjit is also CPython-compatible, hence the compatibility to CPython C extensions are kept. Dynjit is lightweight and extensible, which allows introducing Dynjit of the early stage into those existing codebases, and gradually producing better performance as the features being extended. Several benchmarks have been performed. It is observed that the performance usually increases to > 1.5 times, depending on the scenarios. Further, 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 have 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 in Python, 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 and individual developers/researchers. The major attempts are Pyjion [2], Pyston [3], PyPy [4], Numba [5] and HOPE [6].

However, all these attempts are not sufficient for 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 their active developments 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 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 at runtime, and Cython [7], the most widely used Python-to-C compiler, can be leveraged to avoid directly 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 review the previous Python JIT attempts, and find 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 only 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 heavy implementation in the very beginning.

We learn from the existing attempts, and hereby present Dynjit.

II. Python JIT Issues

It has been nearly 30 years for the world to try optimizing Python's performance. Due to the dynamic nature of Python, the existing attempts are restricted and cannot work out a good trade-off for general-purpose programming in Python.

Firstly, Python programs are overly generic. A Python function accepts arbitrary arguments at runtime, and checks the validity later, even though the validity can be statically checked. For instance, *isinstance* is heavily used by users in Python to check the types at runtime, and the addition of float objects tries to check if the arguments are floats, otherwise a conversion will be attempted.

Various kinds of code specialization are introduced in Dynjit to address this issue.

Secondly, Python is object-oriented, hence attribute accessing is frequent. However, attribute accessing in Python involves the operation of the attribute name string, during which hash lookups happen. Attribute accessing in Python can be categorized into four modes

1) accessing fields maintained by dictionary
2) accessing "__slots__" fields
3) accessing static/class methods
4) accessing methods

It is observed by our research that the last 3 modes can be greatly optimized in different approaches, when the type information is provided.

Besides, there are still a few other language features caused by the dynamic nature of Python. These features are much more difficult to optimize.

For instance, Python global variables can be modified outside a module, at any stage of the program. This makes it hard to decide the references of a Python global symbol.

Another considerably specific instance is *union type*. *Union type* is classified as a difficult feature to optimize among dynamic programming languages. *Union type* denotes the cases that the return type of a function depends on the runtime value of arguments or the global states.

The Julia programming language [1] gives an approach to handle union types. With a minor performance loss to split programs involving union types into several specialized codes without union types, it has finally succeeded in optimizing a large range of programs suffering from union types.

```
x :: Union{Int, String} = ..
if x isa Int
    # now x is treated as an integer
    # in the following programs S1
    S1
else
    # now x is treated as a string
    # in the following programs S2
    S2
end
```

Listing 1. Union Split in Julia

To address the same issue, their ideas have been stolen to strength the capability of Dynjit.

III. 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.

Th intermediate representation called Core CPY is developed, to make the 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.

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

Formula 1. Syntax of Core CPY

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

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

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

Formula 2. Semantics for the instructions POP and PEEK

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 a real Python runtime object, and use of the static repr may contribute to constant propagation; if it's dynamic, it refers to a named or temporary variable.

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

Listing 2. Dynjit repr

A **shape** usually 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_func_object
| NominalS  of typename * method_dict * ...
| ...
```

Listing 3. Dynjit Shapes

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 $\mathbf{Dynjit}^{tree+goto/label}$ 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 $\mathbf{Dynjit}^{tree+goto/label}$ 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 argument 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_int_add** has been provided.

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

```
type expr =
| Val of abstract_value
| Call 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
```

Listing 4. Dynjit IRs

$\mathbf{Dynjit}^{tree+goto/label}$ 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.

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

Listing 5. Union Split: Original 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)
    ...
```

Listing 6. Union Split: Union-split Code

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 when the condition of jump can be decided statically.

The idea can be demonstrated with the following 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
```

Listing 7. Boolean Split: Original 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
```

Listing 8. Boolean Split: Boolean-split Python Code

PROGRESS

Dynjit has been now prototyped and shown a large factor of performance improvement over several kinds of Python programs, such as attribute accessing, loops, union types, reading global variables, closure constructions and dynamic dispatch. The program that can be optimized is restricted, for instance, attribute accessing can have a **1.5x** speed up when the attribute is a "__slots__" field, and the speed up for type specialized programs depends on the complexity of programs, and can usually produce a factor of $>$ **3.5x**.

However, the speed up is now not that dramatic due to the following reasons.

1) Dynjit now does no object unboxing
2) It's hard to measure the effect of several optimizations when they collaborate together.

The code of Dynjit prototype is available in the same-name GitHub repository for reproduction purpose.

FUTURE OUTLOOK

Paper works are now the focus of our research.

Dynjit widely adopts ideas from many existing attempts and some other programming languages, and has made progress. It's crucial for us to present paper works and accept peer reviews, which contributes to the solution to Python performance issues, and helps to the stabilization of the research production.

On the opposite direction, to address some existing restrictions to current Dynjit, radical evolutions might be applied. For instance, it's hard for us to optimize recursive functions, and the runtime generated code can be overly large.

Dynjit now can optimize a limited set of Python language features, we are to work on the optimizers for full-featured CPython, in order to be practical, and general-purpose.

Besides of the aspect of generality, it is also observed that Dynjit is much slower than Numba in numeric computing tasks. So far this is thought to be a consequence of lacking of object unboxing. Object unboxing and other optimizations are now being considered to speed up domain-specific tasks.

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.