Type Freezing: Exploiting Attribute Type Monomorphism in Tracing JIT Compilers

Lin Cheng
Berkin Ilbeyi
lc873@cornell.edu
bi45@cornell.edu
Cornell University
Ithaca, NY, USA

Carl Friedrich Bolz-Tereick
cfbolz@gmx.de
Heinrich-Heine-Universität
Düsseldorf
Germany

Christopher Batten
cbatten@cornell.edu
Cornell University
Ithaca, NY, USA

Abstract

Dynamic programming languages continue to increase in popularity. While just-in-time (JIT) compilation can improve the performance of dynamic programming languages, a significant performance gap remains with respect to ahead-of-time compiled languages. Existing JIT compilers exploit type monomorphism through type specialization, and use runtime checks to ensure correctness. Unfortunately, these checks can introduce non-negligible overhead. In this paper, we present *type freezing*, a novel software solution for exploiting attribute type monomorphism. Type freezing "freezes" type monomorphic attributes of user-defined types, and eliminates the necessity of runtime type checks when performing reads from these attributes. Instead, runtime type checks are done when writing these attributes to validate type monomorphism. We implement type freezing as an extension to PyPy, a state-of-the-art tracing JIT compiler for Python. Our evaluation shows type freezing can improve performance and reduce dynamic instruction count for those applications with a significant number of attribute accesses.

CCS Concepts • **Software and its engineering → Language features**; **Just-in-time compilers**.

Keywords dynamic languages, just-in-time compiler

ACM Reference Format:
Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. 2020. Type Freezing: Exploiting Attribute Type Monomorphism in Tracing JIT Compilers . In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3368826.3377907

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377907

```
1 def foo(pt):
2   x = pt.x
3   y = pt.y
4   return x+y
```
(a)

```
1 assert_type(pt,Point)
2 _x = load_attr(pt,x)
3 _y = load_attr(pt,y)
4 r = add_int(_x,_y)
5 return r
```
(c)

```
1 assert_type(pt,Point)
2 _x = load_attr(pt,x)
3 _y = load_attr(pt,y)
4 assert_type(_x,int)
5 assert_type(_y,int)
6 r = add_int(_x,_y)
7 return r
```
(b)

Figure 1. Example of Type Specialization vs. Type Freezing – (a) dynamic language pseudocode; (b) intermediate representation (IR) nodes after applying type specialization in a traditional JIT compiler; and (c) IR nodes after applying type specialization and type freezing in our proposed JIT compiler.

1 Introduction

Dynamic programming languages have become increasingly popular across the computing spectrum from the Internet of Things (e.g., MicroPython for microcontrollers), to mobile devices (e.g., JavaScript for web browsers), to servers (e.g., Ruby on Rails, Node.js). Among the top-ten most popular programming languages, four of them are dynamic [7]. Dynamic programming languages usually support lightweight syntax, managed memory, garbage collection, and dynamic typing. These features, along with rich and powerful built-in libraries, make dynamic programming languages highly expressive and productive [6, 16, 22, 25].

Type polymorphism, in which data with different types can be associated with a single identifier, is one of the key features of dynamic typing. Because of type polymorphism, many operations (e.g., add) need to use *type dispatching* to determine the concrete operations for specific operands (e.g., add for adding two strings vs. add for adding two integers). Type dispatching and other dynamic features mean dynamic languages are usually interpreted using a virtual machine. For example, each time an interpreter executes method `foo` in Figure 1(a), it needs to determine the correct addition semantics for the types of `x` and `y`.

Type monomorphism, where an identifier is only associated with a single type of data throughout the duration of

the application, is actually not as uncommon as one might expect in dynamic language programs. In fact, an analysis of popular Python-based frameworks and libraries revealed that at least 79% of the identifiers are type monomorphic [30]. Just-in-time (JIT) compilation is a popular way to address the performance gap between dynamic languages and ahead-of-time (AoT) compiled languages. Most recent JIT compilers apply an optimization technique called *type specialization*, which eliminates type dispatching overhead by speculatively replacing generic operations with concrete ones in the JIT compiled code. However, there is no guarantee that any identifier in a dynamic language program will remain type monomorphic, so runtime checks (e.g., `assert_type` at lines 1, 4, and 5 in Figure 1(b)) are needed to ensure correctness.

Attribute type monomorphism is where an attribute in all instances of a certain user-defined type holds only one type of data. For example, in Figure 1, the `x` and `y` attributes of `Point` objects always store integer values. However, a traditional JIT compiler applying type specialization (Figure 1(b)) still needs to insert an `assert_type` on each of these attributes (lines 4–5) to ensure the `Point` type has not been modified elsewhere. Note that the check on the type of these attributes is in *addition* to the preceding check on the type of `pt` (line 1). In Section 4, we present *type freezing*, a new way to exploit attribute type monomorphism that complements type specialization. Type freezing *freezes* type monomorphic attributes of user-defined types, and eliminates the necessity of runtime type checks when performing reads from these attributes. Instead, runtime type checks are done when performing writes to validate type monomorphism. Previous work described a software/hardware hybrid scheme to mine attribute type monomorphism and remove redundant type checks [12]. We propose two pure software mechanisms, *simple type freezing* and *nested type freezing* in the context of a tracing JIT compiler, which achieve similar performance improvements as this prior work without the need for any form of specialized hardware. The JIT compiled code after applying type specialization and our technique is shown in Figure 1(c). We implemented both proposed mechanisms as extensions to PyPy, a widely adopted implementation of Python [20]. Our evaluation on two real machines shows that: (1) for most applications that use user-defined objects heavily, our techniques improve performance by 5% on average and up to 16%, while reducing dynamic instruction count by 8% on average and up to 17%; (2) for applications that rarely use user-defined objects, or do not use them at all, our mechanisms incur minimal overhead.

The contributions of this work are: (1) we quantify type monomorphism in real-world Python workloads; (2) we propose two pure-software mechanisms, *simple type freezing* and *nested type freezing*, to exploit attribute type monomorphism in the context of a tracing JIT; and (3) we show the effectiveness of our techniques with microbenchmarks and full-size benchmarks on PyPy, measured on real machines.

2 Background on JIT Compilation

In this section, we briefly introduce just-in-time compilation, meta-tracing JIT compilers, RPython, quasi-immutables, and the map optimization prevalent in modern JIT compilers.

Just-in-Time Compilation JIT compilation has been increasingly adopted as dynamic languages gain in popularity. Under Rau's categorization of program representations [23], the source code (e.g., C/C++, Fortran, and Python) is called *high-level representation (HLR)*, and a JIT compiler translates a *directly interpretable representation (DIR)* (e.g., Java and Python bytecode) to a *directly executable representation (DER)* (e.g., RISC-V [24] and x86 machine instructions). A method-based JIT compiler (e.g., Google V8 [28], JavaScriptCore [18]) mainly targets frequently executed methods, and preserves the control flow graph in the DER. A tracing JIT compiler (e.g., HotpathVM [14], TraceMonkey [13]) mainly targets frequently executed loops, and compiles the DER from a linear trace through the DIR. While it is simpler to apply certain compiler optimizations when control flow is linear, runtime control flow checks must be added to ensure the execution is still taking exactly the same path. These checks, along with the runtime type checks we mentioned before, are usually called *guards*.

Meta-Tracing JIT and RPython Applying JIT compilation significantly closes the gap between dynamic languages and AoT-compiled languages. However, it is well known that developing performant JIT compilers is very challenging. Meta-JIT compilers (e.g., Truffle framework [29] and RPython framework [2]), which separate language definition from virtual machine and JIT compiler implementations, have been proposed to overcome this difficulty. The *RPython* framework allows language implementers to construct interpreters for the target languages in a high-level language (i.e., a restricted subset of Python). PyPy is a production-ready Python implementation developed using RPython.

With only a few additional hints, the RPython framework is able to automatically generate a JIT compiler for a target language. The RPython framework also provides additional hints that can dramatically improve the performance of the resulting JIT compilers [3]. For example, the hint *promote* marks variables as runtime constants, and allows the JIT compiler to remove subsequent reads to these variables.

Quasi-Immutable RPython also provides a way to optimize for attributes that rarely change. When defining a VM-level class, one can annotate certain attributes to be quasi-immutable variables. Then hooks, which are very much like write-barriers in the context of garbage collectors, are automatically generated for these attributes' modifiers. During tracing, the values read from these attributes are considered to be constants. When a quasi-immutable variable is written to, all compiled traces that access this variable are invalidated.

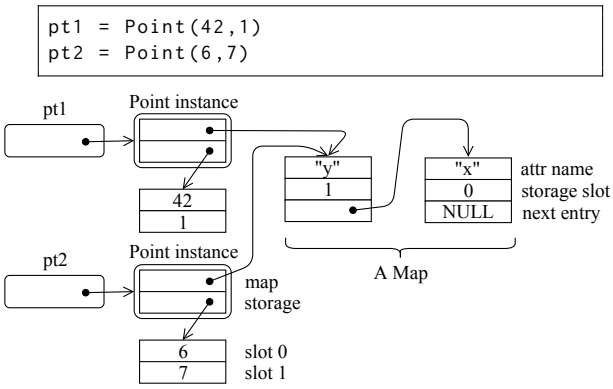

Figure 2. Objects and Maps – A digram of objects and maps after running the code listed at the top. Left side of the figure shows two simplified user-defined objects. Each object has two fields, map and storage. Right side of the figure shows a simplified map, which is composed as a linked list of map entries. Each entry holds meta data for a single attribute.

```
1  def read(self, obj, name):
2    attr = self.find_map_attr(name)
3    if attr is None:
4      self.handle_error()
5    return obj.read_storage(attr.storageindex)
```

Figure 3. PyPy `read` Method – Simplified version of the `read` method in PyPy which uses the map optimization.

Map Optimization Many modern dynamic languages (e.g., Python and JavaScript) allow attribute manipulation outside of object constructors. These languages allow instances of the same class to have different sets of attributes. This can cause significant overhead (e.g., CPython maintains a space consuming attribute dictionary for each object). However, in practice many objects share the same set of attributes. *Maps* (i.e., hidden classes or Shapes in Google V8) are a well-known technique to exploit this similarity, which were first introduced by the SELF project [8]. PyPy employs this technique [5]. A map is an immutable collection of attributes that defines the type of the object. A map is implemented as a linked list of attribute entries (see Figure 2). When an attribute is added to or deleted from an object, the object points to another map which reflects its current attribute set. If the map we want to point to does not already exist, a new one will be created. To read an attribute from an object, we use bytecode `LOAD_ATTR`, which pops the user-defined object off the stack, and then invokes the `read` method of its map. A simplified implementation of `read` is shown in Figure 3. Inside `read`, we first look up where this attribute is stored using the name of this attribute and a linear search over the map. If there is no such attribute, a special routine handles the failure. Otherwise, we read from this object's storage space in a separate data structure using the index found in the map lookup.

3 Attribute Type Monomorphism

Type specialization is a widely adopted JIT compiler technique to exploit type monomorphism in dynamic programming languages by replacing general operations with type-specific ones. Since there is no guarantee that any identifier will remain type monomorphic, runtime checks are needed to verify future data still has the expected type. Previous research has shown that executing these runtime type checks consumes considerable time and energy [10, 11].

Attribute type monomorphism is a special kind of type monomorphism, where an attribute in all instances of a certain user-defined type holds only one type of data. A study on JavaScript applications has revealed the existence of attribute type monomorphism, and the potential benefit of exploiting it [12]. We conduct a similar study on applications from the PyPy Benchmark Suite [21]. We profile and characterize each application using a variety of statistics including the number of attribute reads (AR), the number of attribute writes (AW), number of reads that are to type monomorphic attributes (MAR), and the read-to-write ratio (AR/AW) (see Table 1). We have observed that in all applications except *gcbench*, *hexiom2*, and *raytrace*, there is significant attribute type monomorphism. Reading monomorphic attributes accounts for 74.8% of all reads. Moreover, a considerable amount of accesses (OMAR in Table 1) are to nested objects in certain applications (e.g., *deltablue* and *mako*). Though attribute type polymorphism (i.e., an attribute of a certain user-defined type holds more than one type of data) does exist in almost all applications, the amount of attributes that become polymorphic is small, except in the case of *sympy*. Excluding *sympy*, on average only 18 attributes hold more than one type of data. The existence and abundance of attribute type monomorphism, along with the rareness of attributes that later become polymorphic, motivate us to investigate mechanisms that automatically discover and speculatively remove runtime type checks on monomorphic attributes, while being able to fall back if a monomorphic attribute eventually becomes type polymorphic.

4 Type Freezing

Previous work has found that runtime checks due to dynamic language features constitute 25% of execution time in state-of-the-art virtual machines [10]. Given the performance and energy impacts of runtime checks, and the abundance of type monomorphic attributes, we propose to use a novel software technique, *type freezing*, for exploiting attribute type monomorphism. In this section, we first describe two mechanisms, *simple type freezing* and *nested type freezing*, in detail. We then demonstrate the benefits of applying type freezing with micro-benchmarks, and we discuss the overheads and correctness of type freezing. We extend the example in Figure 1 to a more complex one to better illustrate how our proposed mechanisms work. In Figure 4, we

Table 1. Benchmark Statistics

Benchmark	Attribute Reads										
	Total (AR)	CAR (%)	MAR (%)	OMAR (%)	PAR (%)	AW	AR/AW	MAP	PM	BC	BC/AR
⋆ deltablue	524.10 M	0.0	56.6	41.7	1.7	107.53 M	4.9	81	11	3.67 B	6
⋆ raytrace	5.01 B	0.2	2.7	9.6	87.4	1.23 B	4.1	86	17	34.07 B	6
⋆ raytrace-opt	5.01 B	0.2	90.2	9.6	0.0	1.23 B	4.1	86	10	34.07 B	6
⋆ richards	808.08 M	3.3	64.5	7.5	24.6	297.25 M	2.7	51	11	7.65 B	9
⋆ eparse	20.34 M	0.0	51.6	0.1	48.4	4.12 M	4.9	70	15	168.75 M	8
⋆ telco	376.50 M	0.0	70.9	1.6	27.5	103.42 M	3.6	95	15	3.04 B	8
⋆ float	150.01 M	0.0	100.0	0.0	0.0	90.00 M	1.7	57	10	1.28 B	8
⋆ html5lib	21.17 M	0.0	70.0	5.7	24.4	4.88 M	4.3	232	29	194.09 M	9
⋆ chaos	538.39 M	0.0	86.1	0.0	13.9	110.09 M	4.9	71	11	5.81 B	10
pyflate-fast	53.54 M	0.0	83.5	0.0	16.5	5.88 M	9.1	65	11	777.69 M	14
⋆ pickle	55.93 M	0.0	100.0	0.0	0.0	452.44 K	123.6	85	13	1.12 B	20
icbd	8.26 K	0.0	66.4	0.9	32.7	2.33 K	3.5	61	13	138.62 K	16
hexiom2	426.31 M	0.0	8.3	1.4	90.3	2.38 M	178.9	74	11	9.57 B	22
⋆ scimark	993.98 M	0.0	100.0	0.0	0.0	503.89 K	1972.6	66	11	17.84 B	17
spambayes	11.00 M	0.1	89.2	0.0	10.6	700.78 K	15.7	344	37	287.24 M	26
json-bench	42.81 M	7.5	92.5	0.0	0.0	2.67 K	16034.0	73	10	2.13 B	49
django	32.48 M	0.0	71.5	14.2	14.3	26.67 K	1217.5	202	20	1.32 B	40
mdp	5.62 M	0.0	58.8	0.0	41.2	1.44 M	3.9	101	16	260.66 M	46
⋆ sympy	6.20 M	0.0	87.3	0.0	12.6	1.44 M	4.3	6410	4774	304.72 M	49
⋆ sympy-opt	6.20 M	0.0	86.8	0.0	13.1	1.46 M	4.2	6410	4774	304.77 M	49
⋆ gcbench	37.14 M	0.0	0.0	0.0	100.0	190.47 M	0.2	59	13	1.78 B	47
genshi-text	11.64 M	0.0	98.1	1.7	0.1	12.20 K	954.2	159	18	814.50 M	69
⋆ genshi-xml	5.84 M	0.0	98.0	1.7	0.3	11.71 K	498.7	166	19	792.76 M	135
gzip	1.49 M	0.0	66.3	3.5	30.2	1.06 M	1.4	129	14	225.51 M	151
crypto-pyaes	9.69 M	0.0	92.9	7.1	0.0	3.24 K	2993.5	62	11	6.67 B	688
regex-effbot	11.10 K	0.0	63.1	1.8	35.0	3.22 K	3.4	54	10	9.32 M	839
⋆ chameleon	451.46 K	0.0	84.0	0.4	15.6	488.86 K	0.9	1411	77	506.81 M	1122
mako	260.12 K	0.0	73.7	17.5	8.8	109.34 K	2.4	340	26	399.19 M	1534
spitfire	59.09 K	0.0	73.0	4.1	22.9	20.82 K	2.8	454	60	405.54 M	6862
⋆ meteor-contest	11.52 K	0.0	77.9	0.6	21.5	4.67 K	2.5	54	10	359.55 M	31219
ai	7.88 K	0.0	68.3	0.8	30.9	2.27 K	3.5	54	10	941.71 M	119536
fannkuch	7.88 K	0.0	68.3	0.8	30.9	2.27 K	3.5	54	10	1.09 B	138818
nbody-modified	7.88 K	0.0	68.3	0.8	30.9	2.27 K	3.5	54	10	2.68 B	339759
⋆ fib	7.88 K	0.0	68.3	0.8	30.9	2.27 K	3.5	54	10	6.23 B	791289

Total (AR) = attribute reads, excluding read attempts to attributes that do not exist. CAR = constant attribute reads; MAR = monomorphic attribute reads in which data is primitive type; OMAR = monomorphic attribute reads in which data is another user-defined object; PAR = polymorphic attribute reads; AW = attribute writes; AR/WA = read to write ratio; MAP = total number of map entries, in which each holds metadata about a single attribute; PM = map entry that becomes type polymorphic; BC = total bytecode count; BC/AR = bytecodes per attribute read. Applications with a star (⋆) are used for evaluation. Note that *ai, fannkuch, nbody-modified, fib* have similar statistics since the only usage of user-defined objects in these applications involves very similar argument parsing code

define two classes, `Point` and `Line`. Each instance of Point has two attributes `x` and `y` that hold integer values. Each instance of `Line` has two attributes `pt1` and `pt2`, where each of them holds an instance of Point. Method `create_lines` constructs and returns a list of `Line` instances, while method `total_length` takes in a list of `Line` instances, computes, and returns the total lengths of all lines. The simplified JIT trace of the while loop in method `total_length` compiled by PyPy is shown in Figure 5.

In PyPy, each primitive type (e.g., integer and list) is implemented as a separate VM class (i.e., `W_IntObject` and `W_ListObject` respectively), while all user-defined types share a single VM class (i.e., `W_ObjectObject`). Thus, to perform type dispatching and implement type guards, the runtime checks if the VM object is an instance of a certain VM class. For example, `guard_class` at line 19 of Figure 5 checks if `p12` is an integer object by verifying if it is an instance of the VM class `W_IntObject`.

4.1 Simple Type Freezing

Simple type freezing removes guards that check if the VM object is an instance of the expected VM class for type-monomorphic attribute reads. In order to automatically discover type-monomorphic attributes, we need a data structure

```
1  import math
2
3  class Point(object):
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8  class Line(object):
9    def __init__(self, pt1, pt2):
10     self.pt1 = pt1
11     self.pt2 = pt2
12
13 def create_lines(n):
14   lines = []
15   for i in xrange(n):
16     pt1 = Point(i, n-i)
17     pt2 = Point(n-i*2, i)
18     lines.append(Line(pt1, pt2))
19   return lines
20
21 def total_length(n, lines):
22   total_length = 0
23   i = 0
24   while i < n:
25     line = lines[i]
26     pt1 = line.pt1
27     pt2 = line.pt2
28     a_side = (pt1.x - pt2.x) ** 2
29     b_side = (pt1.y - pt2.y) ** 2
30     total_length += math.sqrt(a_side + b_side)
31     i += 1
32   return total_length
```

Figure 4. Running Example – This example creates a list of lines and then calculates their total length.

```
1  i5 = int_lt(i1, i2)                # i < n
2  guard_true(i5)                     #
3
4  p7 = get_array_item(p0, i1)        # line = lines[i]
5  guard_class(p7, W_ObjectObject)    #
6
7  p8 = get(p7, Map)                  # pt1 = line.pt1
8  guard_value(p8, Map of Line)       #
9  guard_not_invalidated()            #
10 p9 = get(p7, slot0)                #
11 guard_class(p9, W_ObjectObject)    #
12
13 p10 = get(p7, slot1)               # pt2 = line.pt2
14 guard_class(p10, W_ObjectObject)   #
15
16 p11 = get(p9, Map)                 # pt1.x
17 guard_value(p11, Map of Point)     #
18 p12 = get(p9, slot0)               #
19 guard_class(p12, W_IntObject)      #
20
21 p13 = get(p10, Map)                # pt2.x
22 guard_value(p13, Map of Point)     #
23 p14 = get(p10, slot0)              #
24 guard_class(p14, W_IntObject)      #
25                 ...
26 p19 = get(p9, slot1)               # pt1.y
27 guard_class(p19, W_IntObject)      #
28
29 p20 = get(p10, slot1)              # pt2.y
30 guard_class(p20, W_IntObject)      #
31                 ...
32 i28 = int_add(i1, 1)               # i += 1
33 jump(p0,i28,i2,f27)                #
```

Figure 5. Baseline PyPy Trace – Simplified trace of the while loop in Figure 4, optimized by the baseline JIT compiler.

to track the types of variables stored for each attribute. One way to implement this is to create a centralized table, where each entry of the table logs whether a specific attribute of a given user-defined type is type monomorphic. However, this table can grow without a bound as new objects and attributes are encountered, and accessing it can have poor data locality, leading to reduced cache performance. A previous study extending the V8 JavaScript JIT compiler for a similar type-monomorphic attribute read optimization concludes that a special hardware cache for this table is necessary since the benefit of exploiting attribute type monomorphism will be diminished using a software monomorphism table [12]. We find it is natural to store attribute type information directly in the maps instead. PyPy already automatically discovers and optimizes for constant attributes using auxiliary variables in a map (see CAR in Table 1). Each attribute entry in a map has a flag, `ever_mutated`. If an attribute is written exactly once, this flag is deemed true and the value stored in this attribute is considered a constant value.

We propose to associate another field, `known_type`, with each attribute entry in a map (see Figure 7). This field can hold either concrete type information, or two special values, `uninitialized` and `mutated`. When an attribute entry is first created, its `known_type` is set to `uninitialized`. During the first store to this particular attribute, `known_type` is initialized to the type of the value to be stored. For subsequent stores, `known_type` is compared with the type of the value to be stored. Upon mismatch, `known_type` is set to another special value, `mutated`, which flags this attribute as type polymorphic. The method we use to conduct bookkeeping is shown in Figure 6(c). When reading an attribute, in addition to finding the storage index by traversing the map and loading the value from the storage (Figure 3), we also read the `known_type` field of this attribute. If its `known_type` holds a concrete type, we convey both the fact that this attribute is type monomorphic and the concrete type information to the JIT compiler through a hint called `record_exact_class`.

PyPy's optimization passes remove duplicated type guards by applying traditional compiler techniques. For example, even though we read from object `line` twice in Figure 4 (lines 26 and 27), only one `guard_class` is inserted for both reads in Figure 5 (line 5). `record_exact_class` is one of

```
1  i5 = int_lt(i1, i2)                 # i < n
2  guard_true(i5)                      #
3
4  p7 = get_array_item(p0, i1)         # line = lines[i]
5  guard_class(p7, W_ObjectObject)     #
6
7  p8 = get(p7, Map)                   # pt1 = line.pt1
8  guard_value(p8, Map of Line)        #
9  guard_not_invalidated()             #
10 p9 = get(p7, slot0)                 #
11
12 p10 = get(p7, slot1)                # pt2 = line.pt2
13
14 p11 = get(p9, Map)                  # pt1.x
15 guard_value(p11, Map of Point)      #
16 p12 = get(p9, slot0)                #
17
18 p13 = get(p10, Map)                 # pt2.x
19 guard_value(p13, Map of Point)      #
20 p14 = get(p10, slot0)               #
21                   ...
22 p19 = get(p9, slot1)                # pt1.y
23
24 p20 = get(p10, slot1)               # pt2.y
25                   ...
26 i28 = int_add(i1, 1)                # i += 1
27 jump(p0,i28,i2,f27)                 #
```

(a) Trace

```
1  def read(self, obj, name):
2    attr = self.find_map_attr(name)
3    if attr is None:
4      self.handle_error()
5    value = obj.read_storage(attr.storageindex)
6    known_type = attr.known_type
7    if known_type is not mutated \
8      and known_type is not uninitialized:
9      record_exact_class(value, known_type)
10   return value
```

(b) Read method implementation

```
1  def record_type_info(self, w_value):
2    if self.known_type is mutated:
3      return
4    if self.known_type is uninitialized:
5      self.known_type = type(w_value)
6    else:
7      if self.known_type is not type(w_value):
8        self.known_type = mutated
9        return
10   return
```

(c) Write path type check implementation

Figure 6. Simple Type Freezing – (a) trace of the while loop in Figure 4 after being JIT compiled by PyPy with simple type freezing; (b) simplified `read` method of PyPy's maps with simple type freezing; (c) type check inserted by type freezing.

the JIT hints provided by RPython framework. A preceding `record_exact_class` marks a certain reference as type frozen, and thus makes subsequent type guards appear to be duplicated. Then the same optimization pass will automatically remove all related type guards on this specific reference. By doing so, simple type freezing eliminates type guards for reads to type-monomorphic attributes. The updated `read` method is shown in Figure 6(b). The same trace of the while loop after incorporating simple type freezing into PyPy is shown in Figure 6(a). Guards at lines 11, 14, 19, 24, 27, and 30 in Figure 5 are removed.

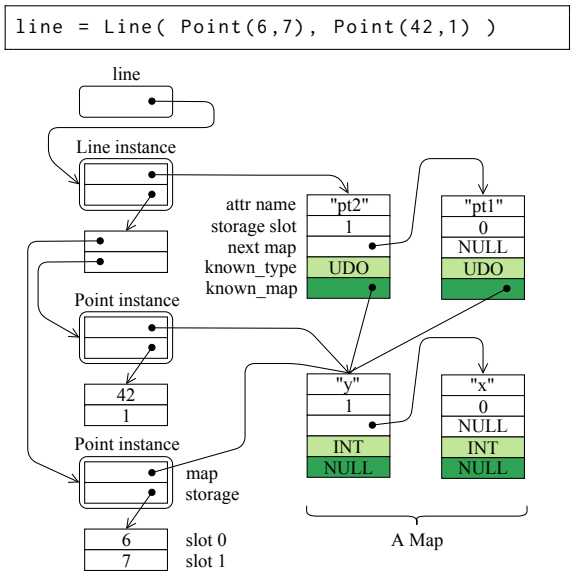

Figure 7. Objects and Maps with Simple and Nested Type Freezing – A digram of objects and maps generated by running the code on the top. Map fields with white background are preexisting. Fields with light green background are added by simple type freezing. Fields with dark green background are added by nested type freezing. UDO = user-defined object; INT = integer.

The optimized trace now assumes that these attributes are monomorphic. However, if a variable of a different type is written to these attributes elsewhere in the code, the trace needs to be invalidated since the monomorphism assumption is no longer correct. To do this, we declare `known_type` to be quasi-immutable. At JIT compilation time, type guards (e.g., `guard_class` in Figure 5) are speculatively removed. The JIT compiler speculates that the `known_type` fields of the monomorphic attributes will not be modified. Thus, if the attribute ever becomes polymorphic, writing into the corresponding `known_type` will automatically invalidate all traces that depend on this specific attribute being type monomorphic (i.e., traces that depend on this particular `known_type` staying immutable). By leveraging this existing mechanism in the RPython framework, we avoid the necessity of adding our own deoptimization mechanism for attributes that turn polymorphic.

4.2 Nested Type Freezing

While simple type freezing can eliminate unnecessary guards for primitive types, it is not sufficient for user-defined types. Unlike primitive types, all user-defined types are represented using the same `W_ObjectObject` VM class, and the JIT uses

the guard_value on the user-defined objects' maps to conduct runtime type checks (e.g., lines 8, 17, and 22 in Figure 5). Among certain applications we study, a considerable number of type monomorphic attribute reads are to user-defined objects (i.e., OMAR in Table 1). We propose *nested type freezing*, which eliminates map reads (e.g., get at lines 16 and 21) and map guards when accessing a type monomorphic attribute that holds user-defined objects.

We associate another quasi-immutable field, known_map, for every attribute entry in maps (see Figure 7). Similar to known_type, it can store either a reference to a concrete map, or two special values, uninitialized and mutated. Note that if a certain monomorphic attribute does not store user-defined objects, the corresponding known_map is meaningless and will never be read. When an attribute entry is first created, its known_map is set to uninitialized. During the first store to this particular attribute, if the value to be stored is a user-defined object, its map is stored in known_map for the corresponding attribute entry in the outer object's new map. For subsequent stores, simple type freezing checks if the value to be stored is still a user-defined object. If so, we further read and compare the map of the object to be stored with known_map. Upon mismatch, the special value mutated will be stored into known_map. An example implementation of the new write semantics is shown in Figure 8(c). If an attribute is observed to have multiple user-defined types stored to it, instead of marking this attribute type polymorphic, we mark it as map-level type polymorphic. *Map-level type polymorphic* attributes exclusively store user-defined objects (i.e., different VM instances of the W_ObjectObject VM class), but may store different *types* of user-defined objects (e.g., a certain attribute can store both instances of Point and instances of Line). Distinguishing map-level type polymorphism from general type polymorphism allows us to preserve the benefit of simple type freezing for these cases (i.e., guard_class can still be removed).

To implement nested type freezing, we extended the vanilla RPython framework with a new hint, record_exact_value. All other parts of our proposed techniques are implemented at the language implementation level. During tracing and compiling, record_exact_value makes a particular value (e.g., pointer to the map of a certain object) appear to be a known constant. Then during JIT optimization, subsequent reads (e.g., get on this map) and guards (e.g., guard_value on this map) are removed by constant folding [3]. The trace of the while loop shown in Figure 4 after being JIT compiled by PyPy with nested type freezing is shown in Figure 8(a), while the read method implementation that supports both simple and nested type freezing is shown in Figure 8(b).

As described so far, the nested type freezing optimization is not safe. It is not sufficient to detect a map-level type-polymorphic attribute by solely observing what values are stored into this specific attribute. This is because the referenced inner user-defined object is itself mutable, and its

```
1  i5 = int_lt(i1, i2)                  # i < n
2  guard_true(i5)                       #
3
4  p7 = get_array_item(p0, i1)          # line = lines[i]
5  guard_class(p7, W_ObjectObject)      #
6
7  p8 = get(p7, Map)                    # pt1 = line.pt1
8  guard_value(p8, Map of Line)         #
9  guard_not_invalidated()              #
10 p9 = get(p7, slot0)                  #
11
12 p10 = get(p7, slot1)                 # pt2 = line.pt2
13
14 p12 = get(p9, slot0)                 # pt1.x
15
16 p14 = get(p10, slot0)                # pt2.x
17                   ...
18 p19 = get(p9, slot1)                 # pt1.y
19
20 p20 = get(p10, slot1)                # pt2.y
21                   ...
22 i28 = int_add(i1, 1)                 # i += 1
23 jump(p0,i28,i2,f27)                  #
```

(a) Trace

```
def read(self, obj, name):
  attr = self.find_map_attr(name)
  if attr is None:
    self.handle_error()
  value = obj.read_storage(attr.storageindex)
  known_type = attr.known_type
  if known_type is not mutated \
    and known_type is not uninitialized:
    record_exact_class(value, known_type)
    if isinstance(value, W_ObjectObject):
      known_map = attr.known_map
      if known_map is not mutated \
        and known_map is not uninitialized \
        and known_map.is_terminal:
        record_exact_value(value._map, inner_map)
  return value
```

(b) Read method implementation

```
def record_type_info(self, w_value):
  if self.known_type is mutated:
    return
  if self.known_type is uninitialized:
    self.known_type = type(w_value)
  else:
    if self.known_type is not type(w_value):
      self.known_type = mutated
      return
  if self.known_map is mutated:
    return
  if isinstance(w_value, W_ObjectObject):
    if self.known_map is uninitialized:
      self.known_map = w_value._map
    else:
      if self.known_map is not w_value._map:
        self.known_map = mutated
  return
```

(c) Write path type check implementation

Figure 8. Simple and Nested Type Freezing – (a) trace of the while loop in Figure 4 after being JIT compiled by PyPy with simple and nested type freezing; (b) simplified read method of PyPy's maps with simple and nested type freezing; (c) type check inserted by type freezing.

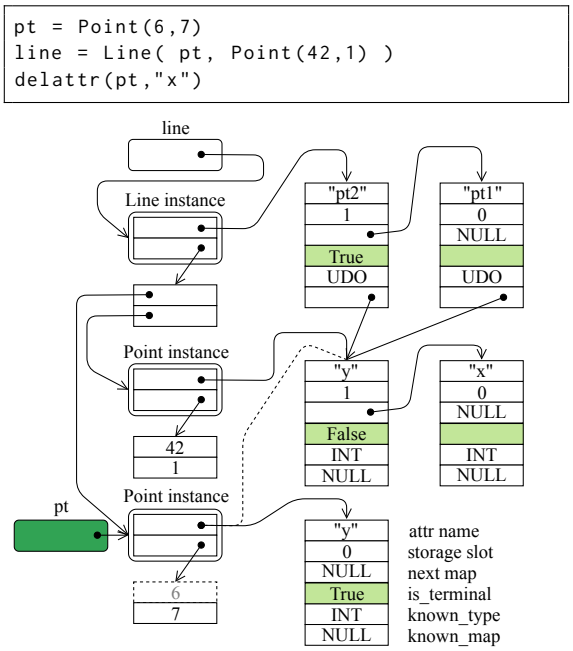

Figure 9. Terminal Maps – Structure of a user-defined object can be mutated through another reference (i.e., remove attribute x through pt in this example). UDO = user-defined object; INT = integer.

map can be changed at a later point through a separate reference. For example, consider the code shown in Figure 9. Assume that initially, both attributes of `Line` are deemed type monomorphic and store `Point` instances that have attributes `x` and `y`. We then alter the structure of a `Point` instance being stored in `line` by removing attribute `x` through an external reference (i.e., `pt`). By definition, attribute `pt1` of `Line` becomes map-level type polymorphic. We can no longer safely assume that all instances stored in attribute `pt1` have an `x` attribute. However, this mutation cannot be noticed by solely observing writes to `line.pt1`. To address this issue, we propose the concept of *terminal maps*. A map is said to be terminal if none of the references that point to this map have added or removed any attributes. We add one additional quasi-immutable boolean field, `is_terminal`, in each attribute entry to track this property (see light green fields in Figure 9). If any reference modifies the type by adding or removing an attribute, it sets the `is_terminal` field to false. For example, when attribute `x` is removed from `pt` in Figure 9, the map `pt` originally points to (the one linked to `pt` with a dash line in Figure 9) has its `is_terminal` field set to be false. As we have seen in Section 2, maps are linked lists composed by attribute entries. The `is_terminal` field in the head entry of a certain linked list indicates that this particular map is considered terminal or not. When reading a type-monomorphic attribute that holds a user-defined object, nested type freezing checks if the corresponding `known_map` stores a reference to a concrete map. If so, it further tests if

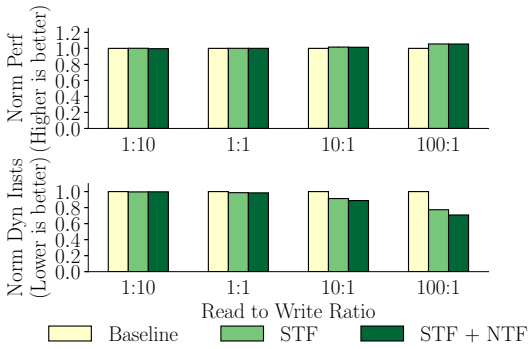

Figure 10. Micro-Benchmark Performance and Dynamic Instructions – STF = simple type freezing; NTF = nested type freezing. Showing 95% confidence interval over 60 runs.

this map is terminal. Only after verifying the map is still considered terminal, nested type freezing eliminates subsequent reads and guards on the map of the object to be read.

The deoptimization mechanism for nested type freezing is the same as simple type freezing. At JIT compilation time, map reads and guards (e.g., `get` and `guard_value` for reading from pt1 and pt2 in Figure 5) are removed by speculating that the corresponding `known_map` and `is_terminal` are immutable. Upon misspeculation, an attribute becomes map-level type polymorphic, and writing into its `known_map` and/or `is_terminal` triggers invalidation of all related traces.

4.3 Type Freezing Benefits

We use the code in Figure 4 with different read-to-write ratios as micro-benchmarks (using 200K `Point` and 100K `Line` instances) to show the benefit of applying simple and nested type freezing. The results are shown in Figure 10. As the read-to-write ratio increases, the benefit of type freezing increases. While simple and nested type freezing provide significant dynamic instruction count benefits (1.40× better instruction count at 100 read-to-write ratio), this does not translate to the same level of overall performance improvement (1.06×). This difference is mostly due to microarchitectural cache effects in modern processors. Without type freezing, when an attribute is read, its type has to be loaded and checked. This will cause the type and the neighboring addresses on the cache line to be brought into the L1 cache. When the attribute value is read from the storage, this data would likely already be in the cache since it is located near the type information in address space, amortizing some of the attribute load cost. With type freezing, we skip loading the attribute's type, so we do not benefit from the spatial locality effect. In the case where there are more writes than reads, type freezing does not incur any significant overhead. Also, we observe that the dynamic instruction count is slightly reduced when the read-to-write ratio is 1. The reason is that, in certain cases, type checks in the write path can be optimized away. For

example, even though theoretically we need to read and verify the types of `pt1` and `pt2` before storing them into a new `Line` instance in method `create_lines` of Figure 4, no `get` or `guard_value` will be generated. This is because we create both `pt1` and `pt2` in the same trace, and thus their type information is already known. This effect is quite common in practice. In another scenario, the value to be stored is read from a type-monomorphic attribute, and this can also eliminate the necessity of performing type checks in the write path.

4.4 Type Freezing Overheads

In the optimized read path, we conduct either one (i.e., applying simple type freezing only) or three (i.e., applying both simple and nested type freezing) field reads and comparisons. We also need to execute either one or two JIT hints (i.e., `record_exact_class` for simple type freezing, and `record_exact_value` for nested type freezing). These additional operations indeed hurt the interpreter performance. However, in practice the interpreter executes for a fraction of the overall execution time for many benchmarks [17]. Thus, the overhead induced by the extra logic we added could be considered as not significant[1]. Once the code is JIT compiled, reading from `known_type`, `known_map`, or `is_terminal` yields zero instructions, since all three are treated as runtime constants by the compiler. Additionally, `record_exact_class` and `record_exact_value` will also be eliminated from the JIT-compiled trace. Once an attribute becomes type polymorphic, subsequent reads will take a path that is the same as the one in the baseline implementation, and thus incur no performance overhead.

Unlike the read path overheads, the extra computation (e.g., type checks) in the write path is a true overhead for both interpreted code and JIT-compiled code. However, this overhead can be amortized over a number of reads in most cases, and will not diminish the benefit of type freezing. As shown in Table 1, most of the applications we studied have substantially more reads than writes. As in the read path, once a particular attribute becomes type polymorphic, subsequent writes will take the original write path which does not induce any additional overhead. Also, as we mentioned in Section 4.3, not every write to a type monomorphic attribute yields runtime type checks. This phenomenon further reduces the type checking overhead induced by type freezing.

We store two additional references (i.e., `known_type` and `known_map`) and a boolean field (i.e., `is_terminal`), in each attribute entry in the maps. A previous study on JavaScript applications has pointed out that the average number of maps ever created in an application is usually small [12]. Table 1 confirms this for our own applications. Most applications create less than 300 attribute entries, and thus associating three additional fields in each attribute entry will not induce a significant storage overhead.

Once a type-monomorphic attribute becomes type polymorphic, all related traces need to be invalidated. If the same code segment continues to be executed frequently, it needs to be recompiled, and this time the attribute is treated as type polymorphic. This may lead to more trace compilation in a JIT compiler with type freezing than in the baseline. To prevent frequent recompilation, we add a heuristic to disable type freezing after seeing a certain number of failures.

4.5 Type Freezing Correctness

Here we informally discuss the correctness of type freezing. Runtime type guards are necessary in the context of tracing JIT compilers because certain operations after a type guard are speculatively specialized for a specific type of data (e.g., integer add vs. float add). For a specific guard on the type of the value being read from an attribute, it can be removed safely as long as the value's type is the same (i.e., monomorphic) for all the instances that have their attribute read at the point of the guard. Type freezing assumes a stronger requirement, in which *all* instances with this attribute must only store values of a single type (i.e., attribute type monomorphism), and only eliminates type guards for attributes if this stronger requirement is met. Thus, type freezing does not affect the correctness of program execution. If at a later point the invariant is invalidated because a different type is stored into the attribute we invalidate all the traces that had a guard removed using PyPy's existing deoptimization mechanism. This ensures that no trace that was optimized under the now wrong assumption will be executed any more.

5 Evaluation

We created three variants of PyPy: PyPy-none is the same as upstream PyPy; PyPy-simple-freezing is upstream PyPy extended with simple type freezing; PyPy-nested-freezing is upstream PyPy with both simple and nested type freezing. Moreover, PyPy-nested-freezing is translated on our modified RPython framework, where we incorporated our new RPython hint, `record_exact_class`. Type freezing is a pure software technique that is easy to implement. In total, we added less than 200 lines of code to the PyPy and RPython framework code base to realize PyPy-nested-freezing.

We focus our evaluation on applications that frequently utilize user-defined objects, but we also include applications that infrequently, or rarely use user-defined objects. We use bytecodes per attribute read (see BC/AR in Table 1) as an approximate measure of how often applications utilize user-defined objects. We include all applications that execute less than 10 bytecodes per attribute read, and we select a representation set from the remaining applications. We also include *gcbench* to quantify the overhead of type freezing when there is almost no attribute type monomorphism, and

[1] There are also some mitigating techniques in place in PyPy to reduce the overhead of maps in the interpreter, such as a software cache.

Table 2. Evaluation Environment Setup

	Platform A	Platform B
Processor	Xeon E5620	Xeon Gold 5218
Base Frequency	2.40GHz	2.30GHz
Turbo Frequency	2.66GHz	3.90GHz
Memory	48GB	96GB
GC Nursery	6MB	11MB
OS	CentOS 7	
Kernel Version	3.10.0-957.21.2.el7.x86_64	
Baseline PyPy	PyPy 7.2.0	

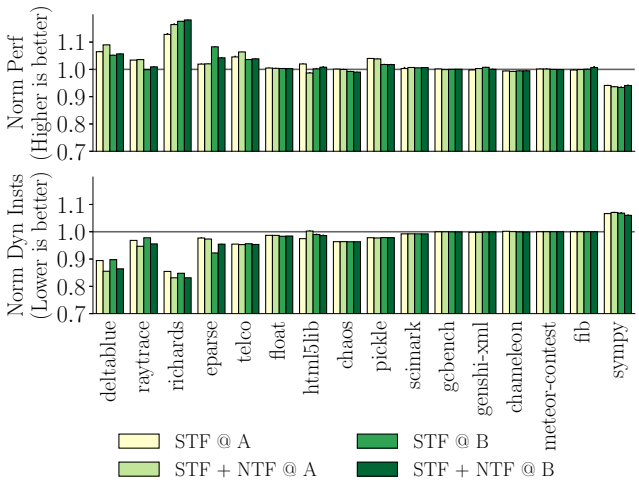

Figure 11. Performance and Dynamic Instructions – STF = simple type freezing; NTF = nested type freezing; A = evaluation platform A; B = evaluation platform B. Showing 95% confidence interval over 60 runs. Note that y-axis starts from 0.7 instead of 0.

sympy for the case which there are many polymorphic attributes. Applications selected are indicated with a star (★) in Table 1. We performed our evaluation on two different server platforms with different microarchitectures (see Table 2). We measure end-to-end performance by taking statistics over 60 runs.

5.1 Results

Normalized performance and dynamic instructions of each application are shown in Figure 11. Note that the y-axis starts from 0.7. As a pure-software technique, simple type freezing achieves a speedup of 13% and 6% on *richards* and *deltablue* respectively on platform A, while improving the performance of *telco*, *raytrace* and *pickle* by around 4%. In the case of applications that access nested user-defined objects (*richards* and *deltablue*) nested type freezing further boosts the performance by up to 14%. Simple and nested type freezing combined reduces dynamic instructions of *deltablue* and *richards* by more than 10%. As we have seen in Section 4.3, type freezing improves dynamic instruction count more significantly than the overall performance. This can correspond to even higher performance gains on mobile and IoT platforms with simple in-order cores, and can potentially also lower energy consumption on all systems. The results on platform B are similar, though a few applications show different behaviors on different platforms. Less performance improvement is achieved for *pickle* and *raytrace*, while *eparse* performs better on platform B.

Type freezing generally does not incur significant overheads for the benchmarks that do not benefit from its optimizations. For applications that rarely use user-defined objects (*chameleon*, *meteor-contest*, and *fib*), and applications that have almost no attribute monomorphism (*gcbench*), type freezing does not hurt the performance. Though being classified as frequently using user-defined objects, *float* and *html5lib* see no performance benefit from type freezing. Further inspection revealed that in both applications, attribute access only contributes to a small percentage of the total dynamic instructions executed. A previous study by Ilbeyi et al. shows that *float* spends most of its time performing garbage collection, while *html5lib* spends most of its time executing AoT compiled functions, instead of JIT compiled code [17]. Type freezing targets JIT compiled regions, and thus cannot help these two applications. For *scimark*, a large number of attribute reads are loop invariant, which can be optimized away by baseline PyPy. Thus in this case, type freezing only slightly improves performance. Overall, there are cases where our technique leads to slightly worse performance and more dynamic instructions compared to baseline, but the amount of overhead incurred by type freezing is not significant for all but one benchmark (i.e., *sympy*).

While it would be interesting to understand the behavior of each individual application, there is still no automatic way to compare the hundreds of traces compiled for each of them. It is possible that type freezing interferes with other heuristics and mechanisms in PyPy. We suspect this interference may lead to the case of *chaos*, in which type freezing does significantly reduce the number of dynamic instructions, but without any performance improvement.

5.2 Application-Level Optimizations

The results presented in the previous section require absolutely no changes to the application. In this section, we study *sympy* and `raytrace` and demonstrate how small modifications to the application (i.e., simply changing how attributes are initialized) can significantly improve performance.

sympy is the only application that performs consistently worse when type freezing is enabled. After a detailed analysis, we found that a simple modification can eliminate almost all of this overhead. *sympy* is a library for symbolic mathematics and it has a core class, `Rational`, which is used to store rational numbers. By definition, a rational number is a

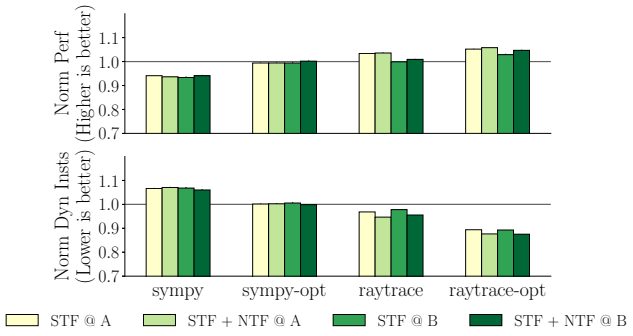

Figure 12. Case Study Performance and Dynamic Instructions – STF = simple type freezing; NTF = nested type freezing; A = evaluation platform A; B = evaluation platform B. Showing 95% confidence interval over 60 runs. Note that y-axis starts from 0.7 instead of 0.

number that can be expressed as a fraction $\frac{p}{q}$, where p and q are integers. Attributes `p` and `q` are usually small integers than can fit into machine registers. However, there are a few occurrence where they overflow to long integers. When it happens, these two attributes become type polymorphic. As a core component of *sympy*, `Rational` instances are accessed in a significant number of traces which are all invalided by type freezing's deoptimization mechanism. Invaliding and recompiling these traces significantly hurts performance. We made a minimal change to *sympy*'s code base by initializing `p` and `q` to `None` before assigning their actual values. Doing so makes these two attributes type polymorphic and our implementation will not even attempt to apply type freezing. The result is shown as *sympy-opt* in Figure 12.

raytrace is an another example where a small modification can enable an application to perform much better with type freezing. Holkner et al. found that many cases of attribute type mutation are due to straight forward type casting (e.g., switching between integers and floats) [16]. After closely inspecting our applications, we found that the same phenomenon caused a relatively high polymorphic attribute read rate in *raytrace*. While attributes in user-defined objects of *raytrace* usually hold float values, they are initialized to integer zeroes. We made minor changes to *raytrace* so that attributes are initiated to float zeroes. The resulting application is *raytrace-opt*. This simple modification decreases the number of polymorphic attribute reads from nearly 90% to 0% (see Table 1). From Figure 12 we can observe that comparing to vanilla *raytrace*, *raytrace-opt* achieves better performance and further reduces dynamic instructions.

6 Related Work

Dot et al. propose a software/hardware hybrid mechanism, which requires modifying the processor, to exploit attribute type monomorphism in the context of JavaScript [12]. They use a standalone data structure, *Class List*, to profile maps and discover type monomorphic attributes. When a method accessing such attributes is JIT compiled, related guards are speculatively removed, just like in type freezing. Upon miss-speculation (i.e., a monomorphic attribute becomes polymorphic), they rely on hardware exceptions to invalidate speculatively compiled code. A special hardware cache is introduced to speedup accesses to frequently used *Class List* entries. Type freezing is a pure software technique, and achieves similar performance improvement. Doh et al.'s work also exploits the case where another user-defined object is stored in a type-monomorphic attribute, but the concept of terminal representation is not mentioned in their work. As we have shown in Section 4.2, solely observing the type of user-defined object to be stored is insufficient to capture all attribute type polymorphism.

Holkner et al. profile production-stage open-source Python programs, and find that even though dynamic features exist throughout the entire life cycle of programs, they mostly happen during startup. They also find attribute type mutation is largely due to straightforward type casting [16]. Xia et al. conduct static analysis on a large Python code base and reveal that 79.7% of the identifiers are type monomorphic [30].

PyPy's storage strategy optimization is a way to exploit type monomorphism in collection data structures. It enables more efficient memory usage and removes type guards in JIT-compiled code when members of certain data structures (e.g., Python lists and dictionaries) are type monomorphic [4]. Dot et al. revealed that guards related to dynamic features contribute significantly to the total execution time when running on the start-of-the-art Google V8 JIT compiler [10]. Checked Load uses hardware to perform dynamic type checking [1]. ShortCut targets V8's unoptimized baseline compiler and uses special hardware to eliminate type dispatching [9]. Other software/hardware hybrid or hardware only schemes have been proposed in the literature as well [11, 19, 26, 27]. Type freezing is a pure software mechanism and does not require any hardware modification. Hackett and Guo [15] propose to combine unsound static type inference with dynamic checks to speculatively emit more specialized code. Type freezing does not perform any static analysis.

7 Conclusions

In this paper, we propose type freezing, a novel pure software scheme for exploiting attribute type monomorphism in dynamic programming languages. Our evaluation on two real machines with applications from the PyPy benchmark suite shows that for applications that can benefit from it, type freezing improves performance by 5% on average and up to 16% and reduces dynamic instruction count by 8% on average and up to 17%. We suspect type freezing can have a impact in other JIT compilers, and we hope our work inspires others to explore the potential for this technique in other tracing JIT compilers or even method JIT compilers.

A Artifact Appendix

A.1 Abstract

This guide describes how to setup PyPy with *Type Freezing* and run both the micro-benchmark and PyPy benchmark experiments we did in this paper. This guide provides instructions to:

- build PyPy from source
- import prebuilt Docker image
- run the micro-benchmark experiments in Figure 10
- run PyPy benchmark experiments in Figure 11 & 12

We have built and tested PyPy and our experiments on a `x86_64` machine with Intel processor. Building PyPy requires approximately 1GB of disk and 6GB of memory. We provide a script to setup dependencies and build PyPy from source as well as a prebuilt Docker image. We also provide scripts to run both experiments, which automatically generate figures similar to Figure 10 and Figure 11 & 12. We have shown in this paper that behavior and performance of each application is non-trivially affected by the machine it runs on, and thus the generated plots may not exactly match these two figures.

A.2 Artifact Checklist

- **Program:** Our customized PyPy, along with both micro-benchmarks in Figure 10 and PyPy benchmarks in Figure 11 & 12, are included in both the source code tarball and the Docker image.
- **Compilation:** We have included a script for building PyPy, which is our main software artifact, from source. A prebuilt PyPy is included in the Docker image. Building PyPy from source takes around 1 hour, depending on the machine it runs on. Importing Docker image only takes a few minutes.
- **Data Set:** All necessary data sets are included.
- **Environment:** We have tested our customized PyPy and evaluation scripts on Ubuntu 18.04. Our experiments rely on `perf` to collect data from CPU performance counters. `perf` is available through mainstream packages. Root privilege is required to modify `perf` configuration file, run Docker, and install necessary dependencies for building PyPy from source.
- **Hardware:** We have tested our customized PyPy on `x86_64` machines with Intel processors. It is recommended to run our experiments on Intel processors with Westmere or later microarchitectures.
- **Execution:** We provide scripts to run both experiments as described in this paper. A more detailed description of how to use them is included in `README`. We also provides a script to run functional unit tests, which is recommended before running either experiments. Running on an idle machine as the sole user is preferred. Running the unit tests and both experiments takes around 4 hours, but it can vary between machines.
- **Output:** Running the scripts as instructed in `README` yields two plots, which should be similar to Figure 10 and Figure 11 & 12 respectively.
- **Publicly Available?:** Source code and a prebuilt Docker image are publicly available at `https://doi.org/10.5281/zenodo.3542289`
- **Code Licenses:** MIT License; Creative Commons Attribution 4.0 International

A.3 Description

A.3.1 How Delivered

Both the source code and a prebuilt Docker image are publicly available at
`https://doi.org/10.5281/zenodo.3542289`

A.3.2 Hardware Dependencies

We have tested our customized PyPy on `x86_64` machines with Intel processors. It is recommended to run our experiments on Intel processors with Westmere or later microarchitectures.

A.3.3 Software Dependencies

We have tested our customized PyPy and evaluation scripts on Ubuntu 18.04. Our experiments rely on `perf` to collect data from CPU performance counters. `perf` is available through mainstream packages. Root privilege is required to modify `perf` configuration file, run Docker, and install necessary dependencies for building PyPy from source. A detailed description of how to install `perf` is included in `README`.

A.4 Installation

- **Option 1. Install Locally on Ubuntu**
  - Download and extract `type-freezing-source.tar.gz`
  - Extract files

```
$ tar -xzvf type-freezing-source.tar.gz
$ cd type-freezing-source
$ unzip pyxcel-artifact-master.zip
$ cd pyxcel-artifact-master
```

  - Create build directory

```
$ mkdir -p build
```

  - Run installation script

```
$ chmod +x ./setup/setup-ubuntu.sh
$ ./setup/setup-ubuntu.sh -d build
```

    Note that if you are running as root, you need to edit `./setup/setup-ubuntu.sh` and remove two occurrences of `sudo`.
- **Option 2. Import Docker Image**
  - Download `type-freezing-docker.tar.gz`
  - Load image into Docker

```
$ sudo docker load --input \
> type-freezing-docker.tar.gz
```

  - Start a new container

```
$ sudo docker run -it --cap-add SYS_ADMIN \
> type-freezing-docker /bin/bash
```

  - Prebuilt artifact is located at `/artifact_top/`, this is equivalent to `build` in option 1.

This information is also available in `README`. Before start the experiment workflow, please refer to `README` for instructions on preparing `perf`.

A.5 Experiment Workflow

Our workflow has three parts

- functional tests
- micro-benchmarks
- PyPy benchmarks

A detailed description of how to run each of them is also included in `README`. Note that if Docker image is being used, there is no need to perform the following command since `$PYXCEL_TOP` is already set.

```
$ source setup-env.sh
```

Note that if Docker image is being used, `/artifact_top/` is the `build` directory.

If the following error occurred, please refer to `README` about how to install `perf`.

```
UnboundLocalError: local variable 'fp'
referenced before assignment
```

A.5.1 Functional Tests

- Go to `build` directory (`/artifact_top/` if using Docker)
- Setup environment variable (skip this if using Docker)

```
$ source setup-env.sh
```

- Go to functional test directory

```
$ cd $PYXCEL_TOP/pyxcel-artifact/functional
```

- Invoke unit tests

```
$ pytest ./ -v
```

All tests should pass.

A.5.2 Micro-Benchmarks

- Go to `build` directory (`/artifact_top/` if using Docker)
- Setup environment variable (skip this if using Docker)

```
$ source setup-env.sh
```

- Go to micro-benchmarks directory

```
$ cd $PYXCEL_TOP/pyxcel-artifact/performance/mbmark
```

- Create a new directory to host profiling files

```
$ mkdir -p run; cd run; rm ./*
```

The following error can be ignored

```
rm: cannot remove './*': No such file or directory
```

- Run micro-benchmarks script

```
$ python \
> $PYXCEL_TOP/pyxcel-artifact/performance/mbmark/\
> run.py
```

- Plot Figure 10

```
$ python plot_mbmark.py
```

Final output is `mbmark.pdf`.

A.5.3 PyPy Benchmarks

- Go to `build` directory (`/artifact_top/` if using Docker)
- Setup environment variable (skip this if using Docker)

```
$ source setup-env.sh
```

- Go to PyPy benchmarks directory

```
$ cd $PYXCEL_TOP/pyxcel-artifact/performance/\
> benchmarks
```

- Create a new directory to host profiling files

```
$ mkdir -p run; cd run; rm ./*
```

The following error can be ignored

```
rm: cannot remove './*': No such file or directory
```

- Run PyPy benchmarks script

```
$ python \
> $PYXCEL_TOP/pyxcel-artifact/performance/\
> benchmarks/run.py
```

- Plot Figure 11 & 12

```
$ python plot_benchmark.py
```

Final output is `cycles.pdf`.

A.6 Evaluation and Expected Results

- functional tests - All tests should pass
- micro-benchmarks - Compare generated `mbmark.pdf` with Figure 10
- PyPy benchmarks - Compare generated `cycles.pdf` with Figure 11 & 12

Acknowledgments

This work was supported in part by NSF SHF Award #1527065, NSF CRI Award #1512937, and equipment donations from Intel. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

References

[1] Owen Anderson, Emily Fortuna, Luis Ceze, and Susan Eggers. 2011. Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)* (Feb 2011).

[2] Carl Friedrich Bolz. 2012. *Meta-Tracing Just-In-Time Compilation for RPython.* Ph.D. Dissertation. Mathematisch-Naturwissenschaftliche Fakultät, Heinrich-Heine-Universität Düsseldorf.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)* (Jul 2011).

[4] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. *ACM SIGPLAN conf. on Systems, Programming, Languages, and Applications (OOPSLA)* (Oct 2013).

[5] Carl Friedrich Bolz and Laurence Tratt. 2015. The Impact of Meta-Tracing on VM Design & Implementation. *Science of Computer Prog.* 98 (Aug 2015), 408–421.

[6] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and Why) Developers Use The Dynamic Features of Programming Languages: The Case of Smalltalk. *Empirical Software Engineering* 18, 6 (Dec 2013), 1156–1194.

[7] Stephen Cass. 2018. The 2018 Top Programming Languages. *IEEE Spectrum* (Jul 2018).

[8] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF A Dynamically-Typed Object-Oriented Language Based on Prototypes. *ACM SIGPLAN conf. on Systems, Programming, Languages, and Applications (OOPSLA)* (Oct 1989).

[9] Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. 2017. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2017).

[10] Gem Dot, Alejandro Martínez, and Antonio González. 2015. Analysis and Optimization of Engines for Dynamically Typed Languages. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)* (Oct 2015).

[11] Gem Dot, Alejandro Martínez, and Antonio González. 2016. Erico: Effective Removal of Inline Caching Overhead in Dynamic Typed Languages. *Int'l Conf. on High-Performance Computing (HIPC)* (Dec 2016).

[12] Gem Dot, Alejandro Martínez, and Antonio González. 2017. Removing Checks in Dynamically Typed Languages Through Efficient Profiling. *Int'l Symp. on Code Generation and Optimization (CGO)* (Feb 2017).

[13] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Jun 2009).

[14] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. *ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments (VEE)* (Jun 2006).

[15] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. *SIGPLAN Not.* 47, 6 (Jun 2012), 239–250.

[16] Alex Holkner and James Harland. 2009. Evaluating The Dynamic Behaviour of Python Applications. *Australasian Conference on Computer Science* (Jan 2009).

[17] Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. 2017. Cross-Layer Workload Characterization of Meta-Tracing JIT VMs. *Int'l Symp. on Workload Characterization (IISWC)* (Oct 2017).

[18] JavaScriptCore 2019. JavaScriptCore. Online Webpage. https://trac.webkit.org/wiki/JavaScriptCore.

[19] Channoh Kim, Jaehyeok Kim, Sungmin Kim, Dooyoung Kim, Namho Kim, Gitae Na, Young H Oh, Hyeon Gyu Cho, and Jae W Lee. 2017. Typed Architectures: Architectural Support for Lightweight Scripting. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Apr 2017).

[20] pypy 2014 (accessed Sep 26, 2014). PyPy. Online Webpage. http://www.pypy.org.

[21] pypybmarks 2014. PyPy Benchmark Suite. Online Webpage. https://bitbucket.org/pypy/benchmarks.

[22] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing Dynamic Features in Python Programs. *Working Conf. on Mining Software Repositories* (May 2014).

[23] B. Ramakrishna Rau. 1978. Levels of Representation of Programs and the Architecture of Universal Host Machines. *SIGMICRO Newsl.* 9, 4 (Nov 1978), 67–79.

[24] riscv 2019. RISC-V. Online Webpage. https://riscv.org.

[25] Elder Rodrigues Jr and Ricardo Terra. 2018. How Do Developers Use Dynamic Features? The Case of Ruby. *Computer Languages, Systems & Structures* 53 (Sep 2018), 73–89.

[26] Thomas Shull, Jiho Choi, Maria J Garzaran, and Josep Torrellas. 2019. NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory. *Int'l Symp. on High-Performance Computer Architecture (HPCA)* (Feb 2019).

[27] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking The Memory Hierarchy for Modern Languages. *Int'l Symp. on Microarchitecture (MICRO)* (Oct 2018).

[28] v8 2019. V8 JavaScript Engine. Online Webpage. https://code.google.com/p/v8.

[29] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. *Symp. on Dynamic Languages* (Oct 2012).

[30] Xinmeng Xia, Xincheng He, Yanyan Yan, Lei Xu, and Baowen Xu. 2018. An Empirical Study of Dynamic Types for Python Projects. *Int'l Conf. on Software Analysis, Testing, and Evolution* (Nov 2018).