Evaluating the Dynamic Behavior of Python Applications

Austin Cory Bart

CS 6304 – Program Analysis

December 1, 2015

KWII 2225

Overview

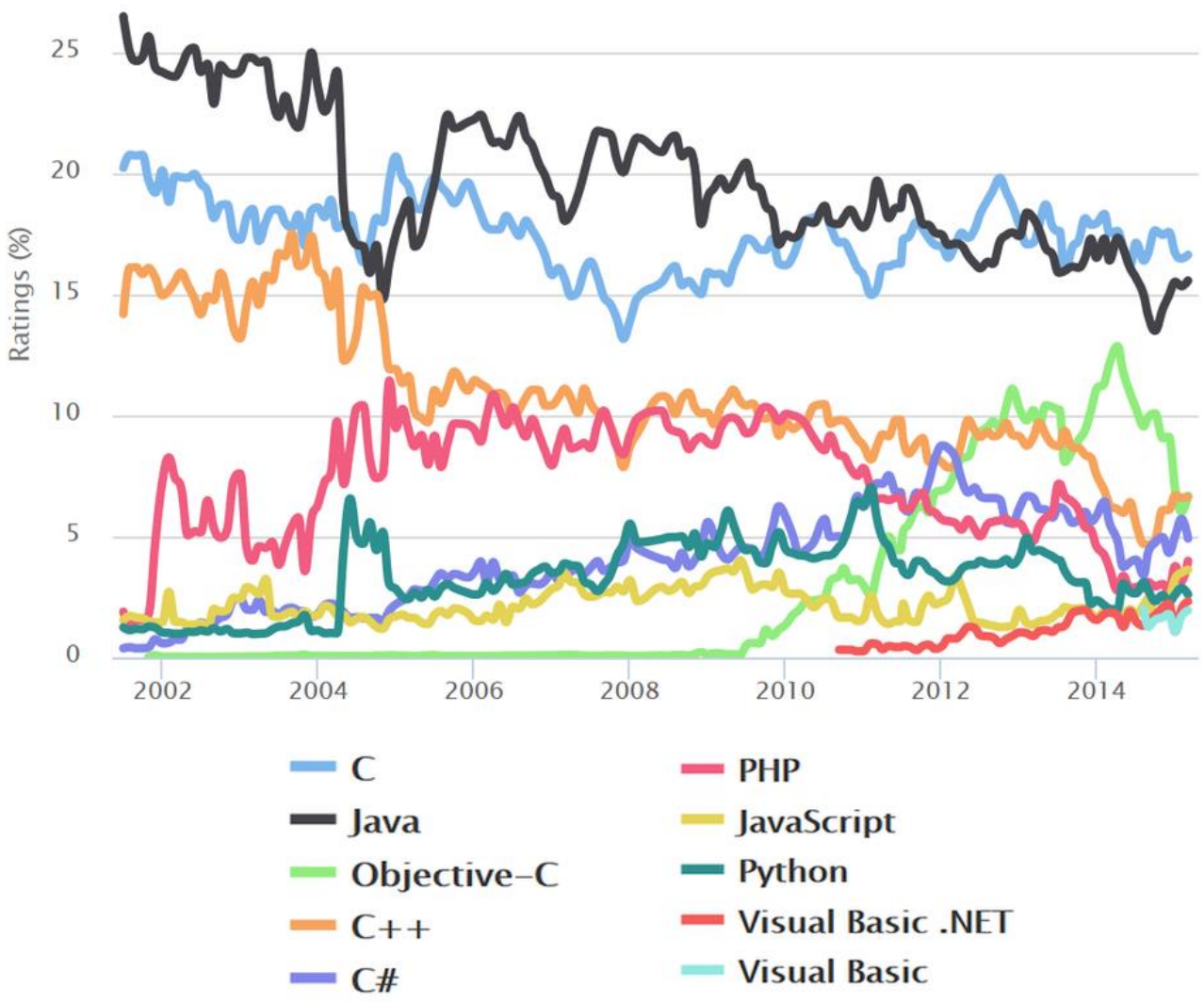
- Background
  - What is Python?
  - What are Dynamic Features?
- Holkner & Harland (2009)
  - Methodology
  - Results
  - Conclusions
- Ackerblom (2014)

Python

- Dynamically-typed, Strongly-typed, static scoping
- Object-oriented, functional, namespaces,
- Built-ins, batteries included, many libraries
- Whitespace significant, readability

Used for…

- Scripting (vs. Perl, Bash)
- Data science (vs. R, MatLab)
- Education (vs. Java, Racket)
- Web servers (vs. PHP, Ruby)



Tiobe Index

Python – A peculiar culture

```
import this
"""The Zen of Python, by Tim Peters. (poster by Joachim Jablon)"""

Beautiful is better than ugly.
Explicit is better than impl..
Simple is better than cσмplεx.
Cσмplεx is better than cOmp1|c@ted.
Flat is better than nested.
S p a r s e  is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
raise PythonicError("Errors should never pass silently.")
# Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
# Although that way may not be obvious at first unless you're Dutch.
Now is better than ...                    never.
Although never is often better thanrightnow.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python vs. JavaScript

- Similarities:
  - Dynamic Typing
  - Control flow: if, foreach, while, functions, etc.
  - Garbage collected, etc.
- Differences
  - Type Coercion
  - Python has more statements
  - setattr/getattr vs. variable[‘access’]
  - Immutability
  - Levels of nothingness
  - Default scoping (var, globals)
  - Whitespace vs. Curlies, Newlines vs. Semicolons

But are they really different?

- Yes.
- JavaScript is terrible.
- Python is wonderful.

- More importantly, JS programmers do more dynamicism than Pythonistas

Four dynamic features

Dynamic Typing

White Magic

```
my_variable = Dog()
my_variable = "Klaus"
my_variable = House()
```

Reflection

Grey Magic

```
my_dog = Dog()
setattr(my_dog, "name", "Klaus")
name = getattr(my_dog, "name")
```

Dynamic Objects

Grey Magic

```
my_dog = Dog()
Dog.bark = lambda : "woof"
my_dog.__class__ = Cat
del Dog.bark
```

Dynamic Code

Black Magic

```
exec('truth= "exec is evil"')
computed = eval('truth')
__import__('os')
```

Dynamic languages

- "Giving people a dynamically-typed language does not mean that they write dynamically-typed programs."

Static occurrence of dynamic constructs

Program	exec	eval	execfile	_import_	LOC	% of LOC
Idle	1	0	1	1	4449	0.07
Gadfly	0	2	0	1	10200	0.03
Grail	4	2	0	0	6419	0.09
HTMLgen	0	4	1	2	4794	0.2
J--	0	0	0	0	1498	0
Lib	11	23	2	12	23754	0.2
Pystone	0	0	0	0	186	0

Dynamic occurrence of dynamic constructs

exec	eval	execfile	_import_	Instructions	% of Instructions
1	0	0	12	346617	0.004
0	47	0	0	7957055	0.0005
214	6	0	0	4676698	0.005
0	831	10	0	422496	0.2
0	0	0	0	8096543	0
0	0	0	0	6702077	0

"He found that up to 7% of variable stores caused a change of type, and that these stores were localised in up to 5% of the total number of variables"

Aycock, J. (2000), 'Aggressive Type Inference', Proceedings of the 8th International Python Conference pp. 11–20.

Methodology

- Analyze large corpus of Python programs
- Tracing to measure use of dynamic features
- Analyze how they are used against hypotheses

A. Holkner and J. Harland, "Evaluating the dynamic behavior of Python Applications" Proceedings of the 32nd Australian Conference on Compuer Science Volumn 91, 2009.

RPython/PyPy

- “Restricted Python”
- Dynamic until `__main__`

- A host of changes
  - No mutating class/method attributes
  - Variables have types (non-primitives unioned with None)
  - Lists -> arrays
  - …

2 Hypotheses

1. That programs written in Python generally do not make use of dynamic features, or that if they do, they can be trivially rewritten in a more static style
2. That while programs written in Python use dynamic features, they do so mostly during program startup, and afterwards behave like a statically-compiled program

2 Hypotheses

1. That programs written in Python generally do not make use of dynamic features, or that if they do, they can be trivially rewritten in a more static style
2. That while programs written in Python use dynamic features, they do so mostly during program startup, and afterwards behave like a statically-compiled program

Result: Partially true

Program Selection

4,500 packages on PyPi (**70,339 today)**

1,000 stable (11,320 today)

50 for linux end-users (106 today)

24 standalone

Do you believe these will be representative?

6x Games	**12x Interactive**	**6x Interactive**

Python Bytecode

```
>>> def hello_world(phrase): print("Printing", phrase, 5)
...
>>> import dis
>>> dis.dis(hello_world)

  1           0 LOAD_CONST               1 ('Printing')
              3 LOAD_FAST                0 (phrase)
              6 LOAD_CONST               2 (5)
              9 BUILD_TUPLE              3
             12 PRINT_ITEM
             13 PRINT_NEWLINE
             14 LOAD_CONST               0 (None)
             17 RETURN_VALUE
```

Four dynamic features

Dynamic Typing

White Magic

```
dog.neighbor = Dog()
dog.neighbor = Animal()
dog.neighbor = [a, b]
dog.neighbor = None
```

attr_mutate_generalize

attr_mutate_type

attr_mutate_none

Reflection

Grey Magic

```
setattr(my_dog, "name", "Klaus")
name = getattr(my_dog, "name")
delattr(my_dog, "name")
globals()['my_dog'] = 3
```

call_locals

call_setattr

call_getattr

call_delattr

call_globals

Dynamic Objects

Grey Magic

```
my_dog = Dog()
Dog.bark = lambda : "woof"
my_dog.__class__ = Cat
del Dog.bark
```

attr_add

attr_del

Dynamic Code

Black Magic

```
exec('truth= "exec is evil"')
computed = eval('truth')
__import__('os')
```

call_execfile

exec_statement

call_reload

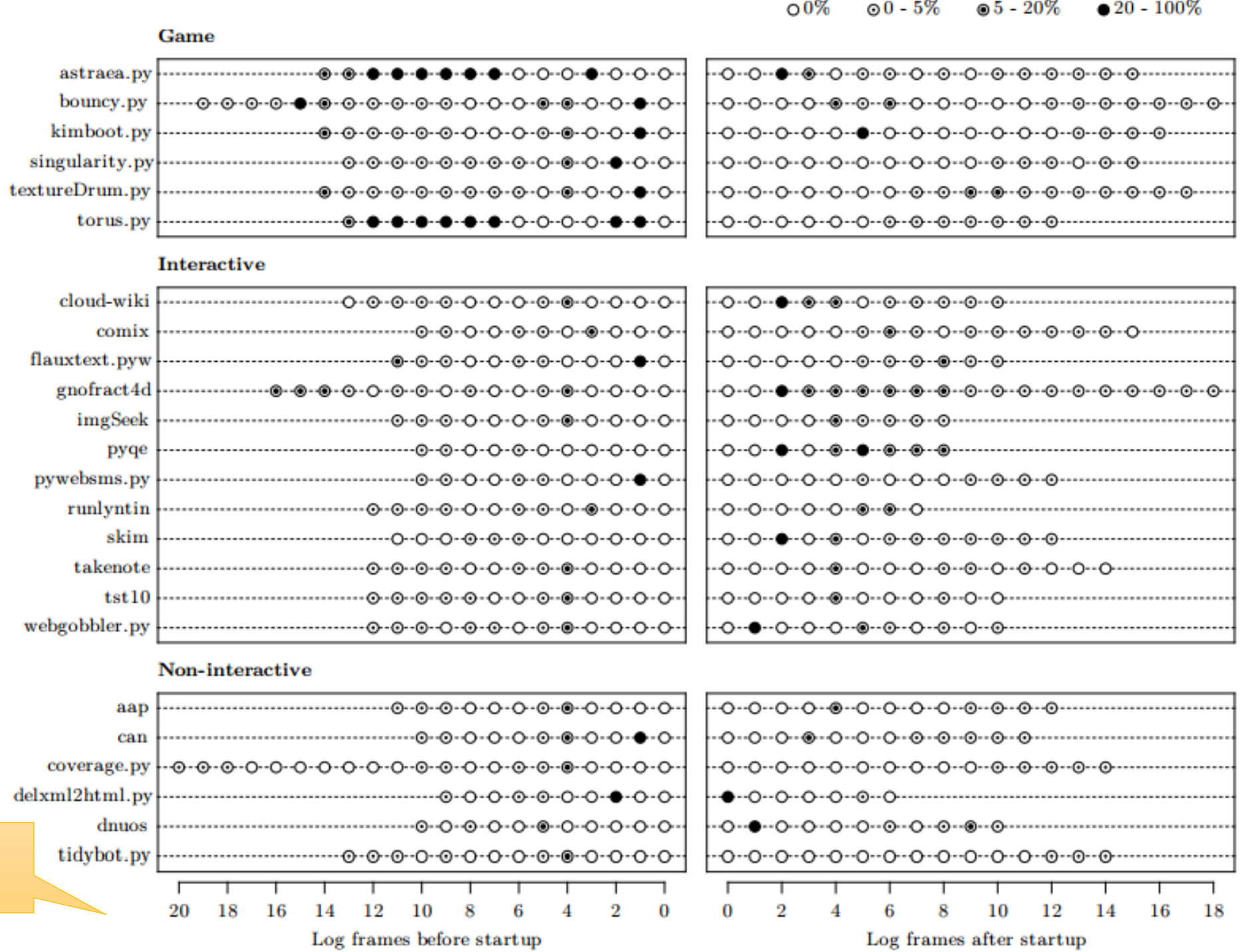
call_eval

Dynamic instrumentation

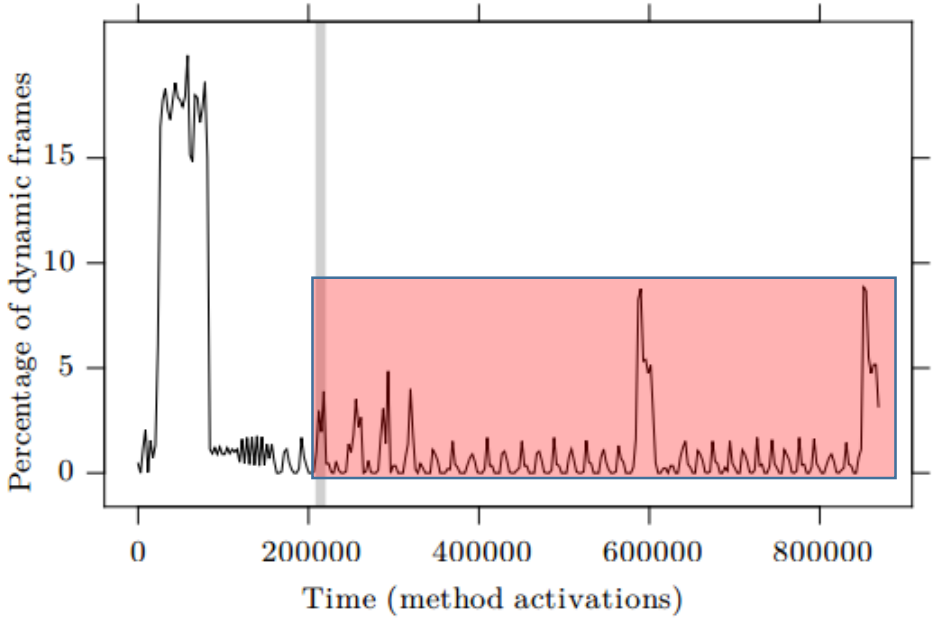
- Bytecode tracing function
- Uses
  - Current frame
  - VM stack
  - Opcode
  - In main?
- -> Is dynamic?

Manually added

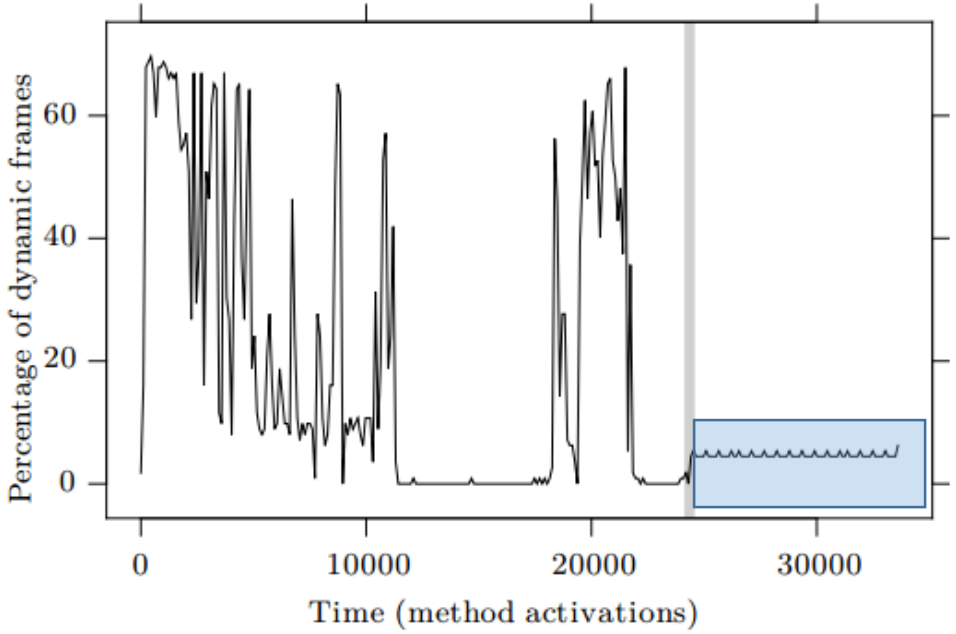
Speed impact!
Hilarious interactions with game clocks!



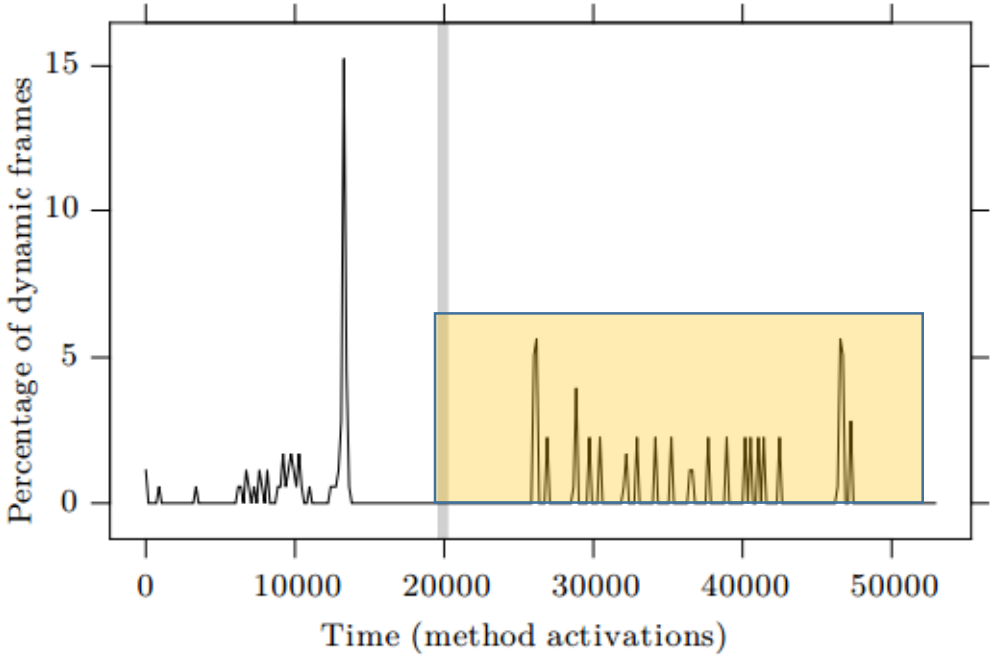
70% higher before startup



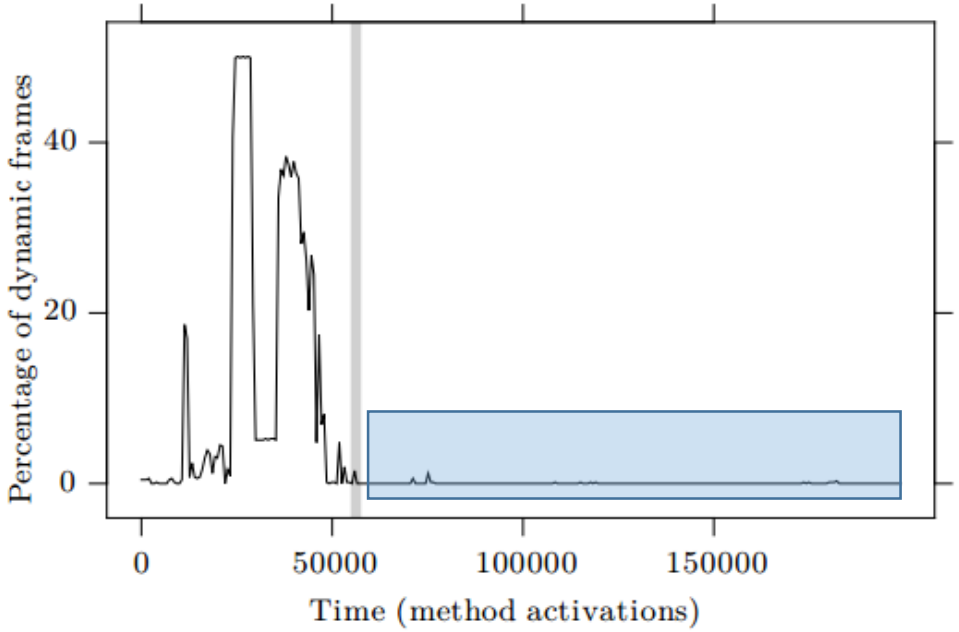
(a) gnofract4d (Interactive)



(b) torus.py (Game)



(d) tidybot (Non-interactive)



(c) kimboot (Game)

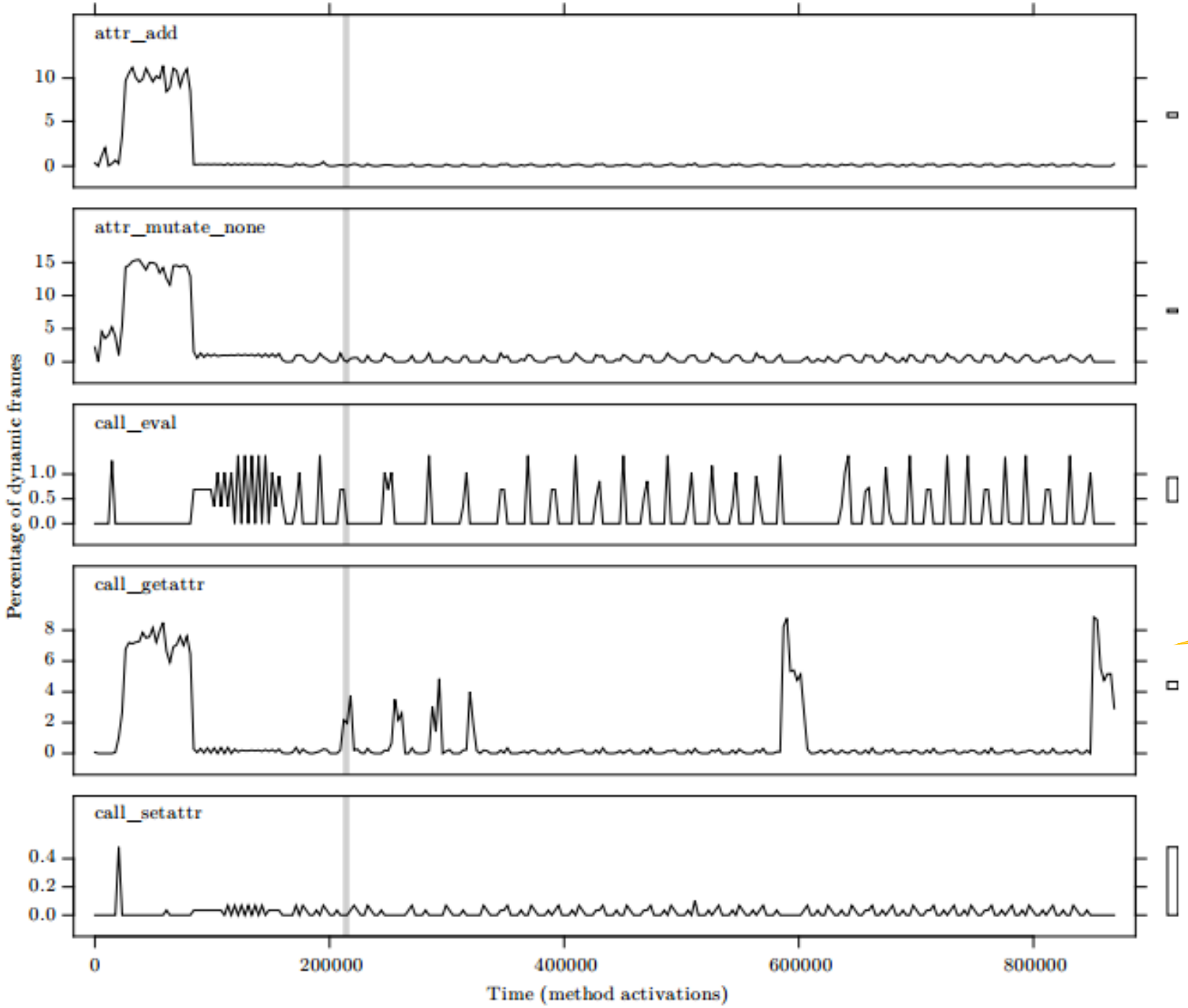


Figure 3: Detailed dynamic behavior for gnofract4d

Generating C code

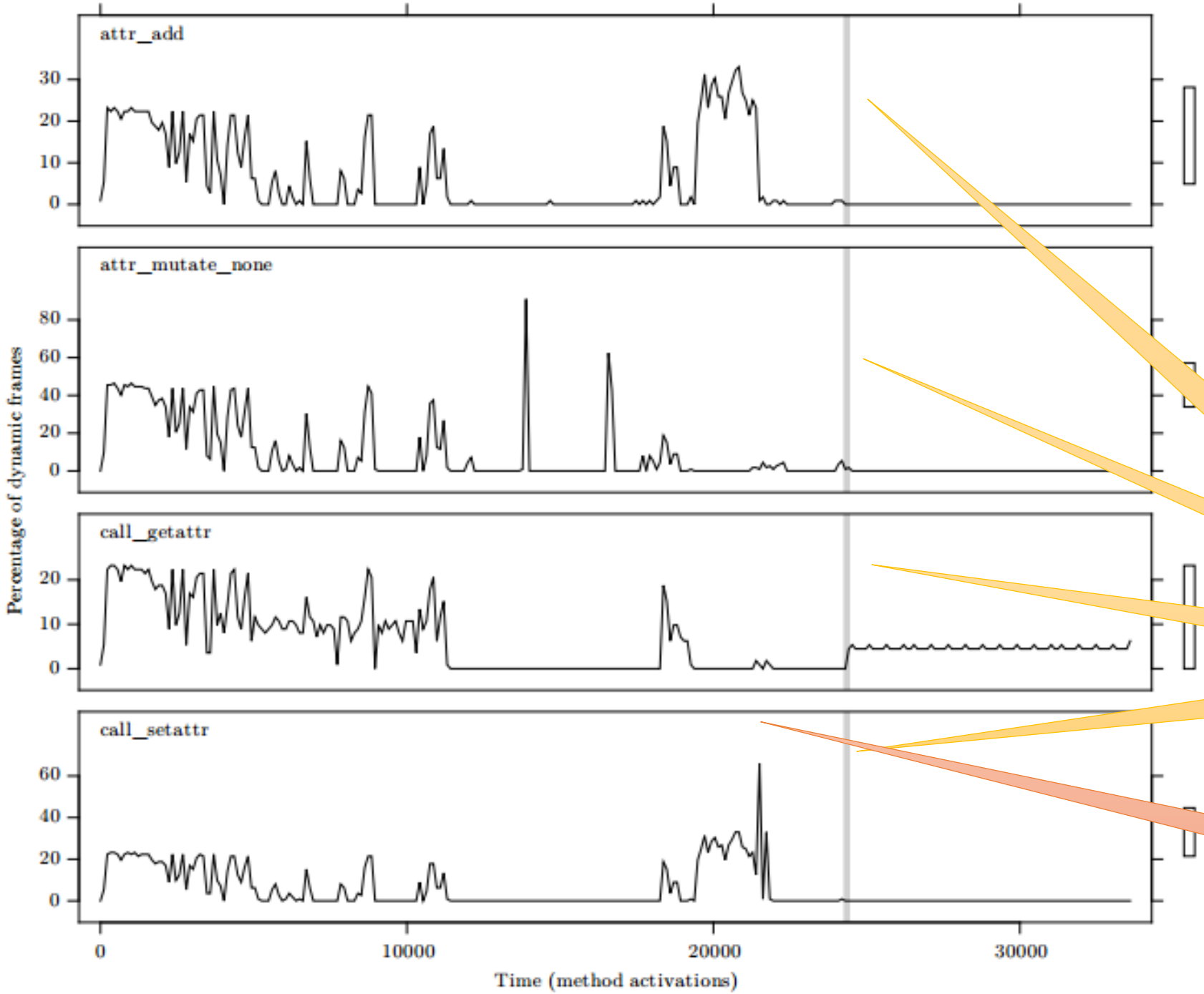


Figure 4: Detailed dynamic behaviour for torus.py

Lazy-loaded module system

Potential correlation between setattr and getattr?

Their analysis

"In both of these cases it's clear that the programs use dynamic features in ways that cannot be easily translated into non-dynamic code. This indicates that **RPython is not a suitable language for these particular programs**."

Blatantly not true for torus.py

Usage of Dynamic Features After Startup

Number of programs using each measured dynamic feature at least once after program startup.

Feature	Programs	Percent
call_execfile	0	0%
call_reload	0	0%
call_delattr	0	0%
attr_del	3	13%
attr_mutate_generalize	4	17%
call_locals	4	17%
exec_stmt	4	17%
call_eval	5	21%
call_globals	6	25%
call_setattr	13	54%
attr_mutate_type	15	63%
call_getattr	21	88%
attr_add	22	92%
attr_mutate_none	23	96%

- Metaprogramming
- Coupled

- Used as `=None`
- As part of a reckless copy

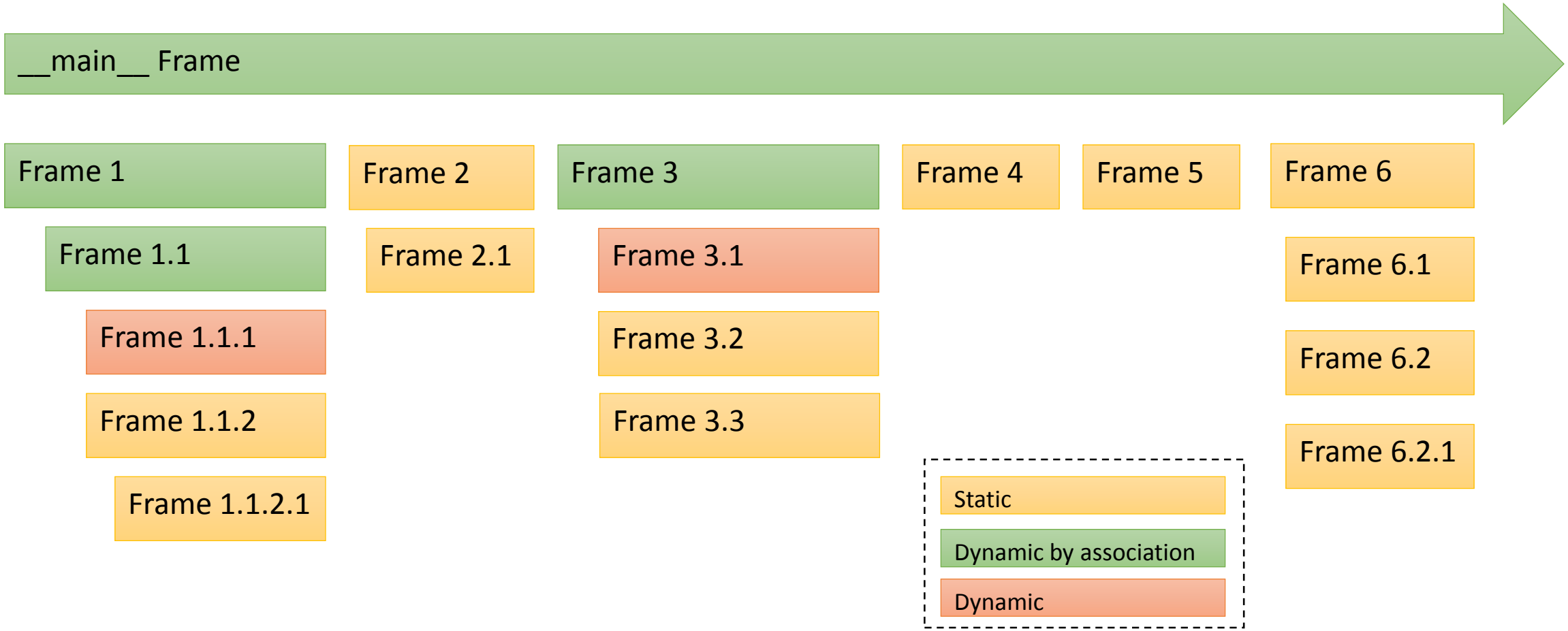
Feh!

- `Int -> float`
- `Str -> Unicode`
- Programmer Error

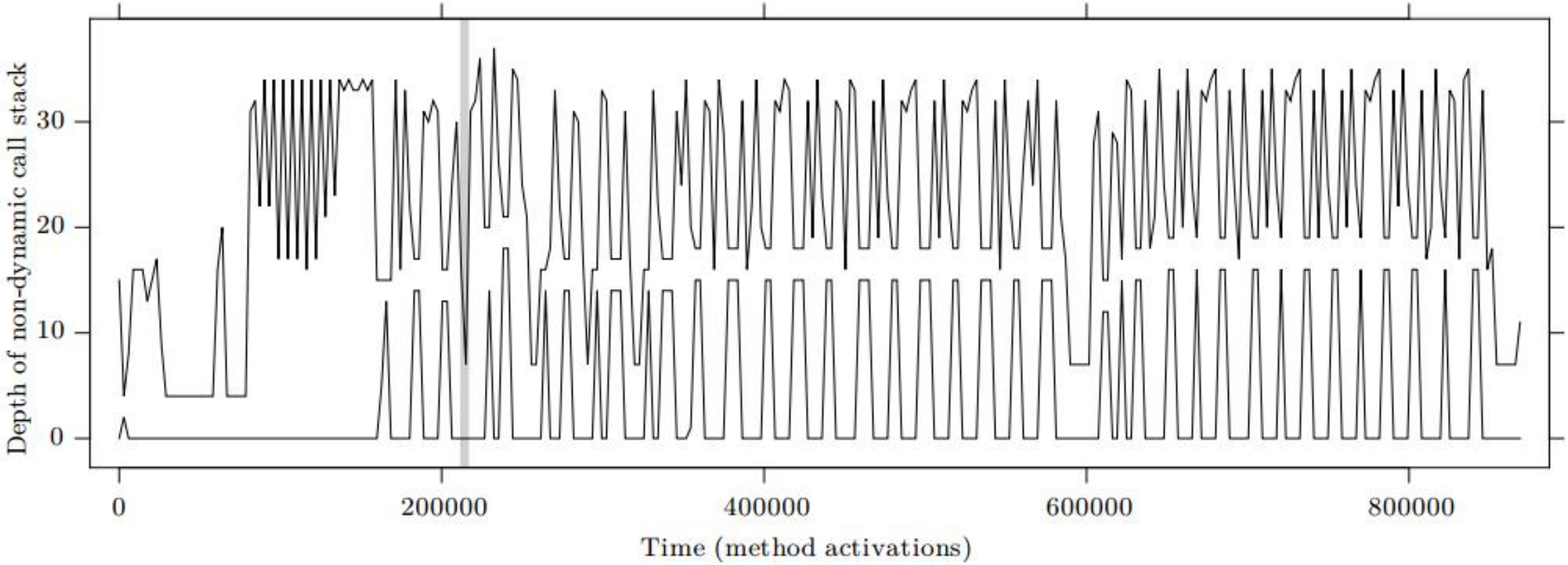
- Used in standard library
- Delayed startup
- But also legit used

Depth of Dynamic Frames

- Can we find isolatable chunks of code that we treat statically?

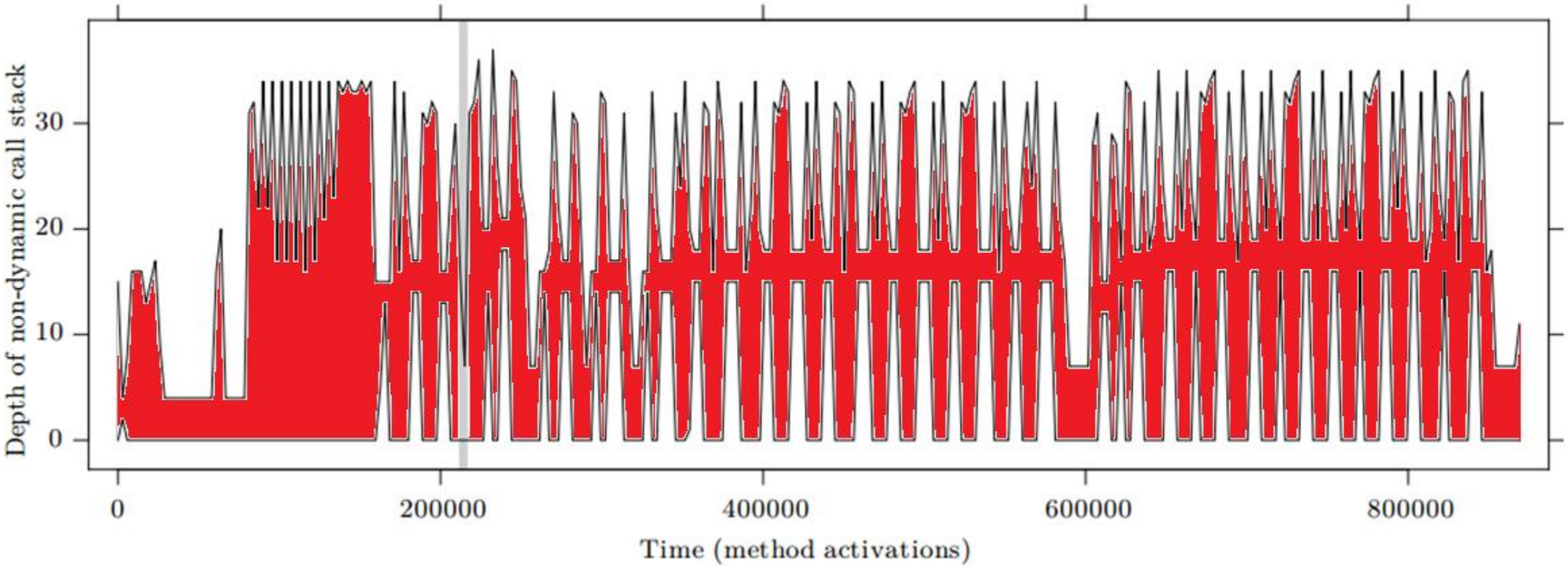


Depth of non-dynamic frames over time (gnofract4d)



High values indicate the current call stack has non-dynamic frames for a large number of parent frames. A zero value indicates the frame is dynamic. The vertical reference line shows the division between startup time and run time.

Depth of non-dynamic frames over time (gnofract4d)



High values indicate the current call stack has non-dynamic frames for a large number of parent frames. A zero value indicates the frame is dynamic. The vertical reference line shows the division between startup time and run time.

Authors' Conclusions

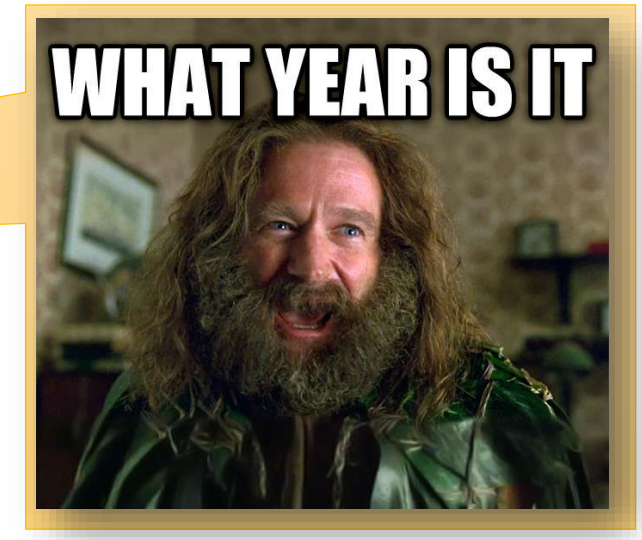
- "Firstly, the hypothesis that programs written in Python generally do not use dynamic features is clearly invalidated." Ehhh…
- "… among the programs tested, 70% have less dynamic activity after startup [and are suitable for Rpython]." Sure!
- "Nearly all programs used reflective features after startup, and over 20% of the programs executed dynamic code." Believable
- "We attributed some dynamic object modification to delayed initialization…" Legit

Lingering Questions

- What about Dynamic Typing?
- Are these representative programs?
  - End-user vs. libraries
- Why care about the entire codebase?

“How do Python Programmers Use Python?”

- Beatrice Åkerblom
- 2014 presentation at PyCon
- Python 2.6



Median, Average, Minimum and Maximum for All Features

Per program dynamic feature usage

	Median	Avg	Min	Max
Entire programs	5.8 K	390 K	214	6.7 M
Library start-up	674	4.5 K	81	56 K
Library run-time	883	350 K	0	6.6 M
Program-specific start-up	508	3.2 K	0	33 K
Program-specific run-time	154	33 K	0	610 K

Type Hinting: Future of Python?

```python
class Greeter:
    def __init__(self, fmt: str):
        if fmt:
            self.fmt = fmt
        else:
            self.fmt = 'Hi there, {}'

    def greet(self, name: str) -> str:
        return self.fmt.format(name)


greeting = Greeter(None)
print(greeting.greet('jane'))
```

Expected type 'str', got 'None' instead more... (⌘F1)

```python
from typing import List


def greeting(names: List[str]) -> str:
    return 'Hello, {}'.format(', '.join(names))


greeting([1, 2, 3])
```

Expected type 'List[str]', got 'List[int]' instead more... (⌘F1)

```python
from typing import Union, List, Dict

GreetingType = Union[List[str], Dict[int, List[str]]]

def greeting(names: GreetingType) -> GreetingType:
    fmt = 'Hello, {}'
    if isinstance(names, dict):
        return [(k, fmt.format(', '.join(v))) for k, v in
                names.items()]
    else:
        return fmt.format(', '.join(names))

print(greeting(['jane', 'john', 'judy']))
print(greeting(
    {10: ['jane', 'judy'],
     11: ['john'],
     12: ['judy', 'john']
     }
))
```