Efficient Hosted Interpreter for Dynamic Languages

PhD Defense by **Wei Zhang**

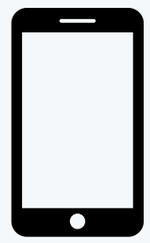
Committee:
Prof. Michael Franz
Prof. Kwei-Jay Lin
Prof. Guoqing Xu

UNIVERSITY OF CALIFORNIA

UCI

A Modern Web Service

client side



JS, Coffee…

server side

Python, PHP, Ruby…

UNIVERSITY OF CALIFORNIA

image courtesy of flaticon.com

UCI

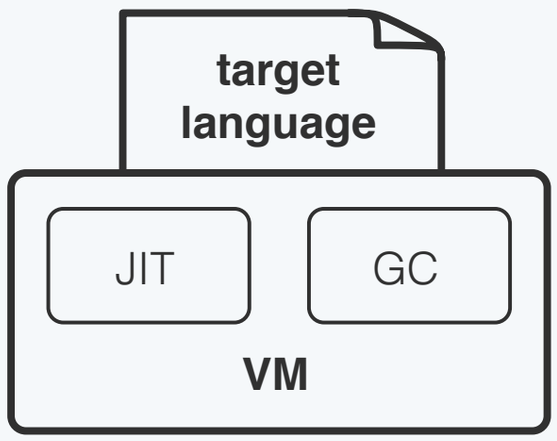
Dynamic languages are no longer — “scripting languages”

- No longer simply used to accomplish small tasks
- Ubiquitous in multiple domains
- Appealing to programmers; offer higher “productivity”
- Suffer from suboptimal performance

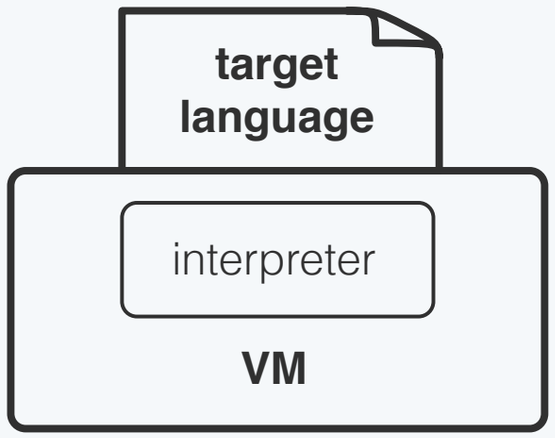
Brief history of dynamic language VMs

1. 1980s: early academic work on Smalltalk & SELF as full-custom VMs
2. Early 90s: interpreters written in C (Python, Ruby)
3. Late 90s: more powerful and popular VM for statically typed OO languages like JVM and CLR (Java & C#)
4. Early 00s: hosted dynamic language VMs (Rhino, Jython, JRuby)
5. Late 00s: second coming of full-custom VMs for dynamic languages (V8)

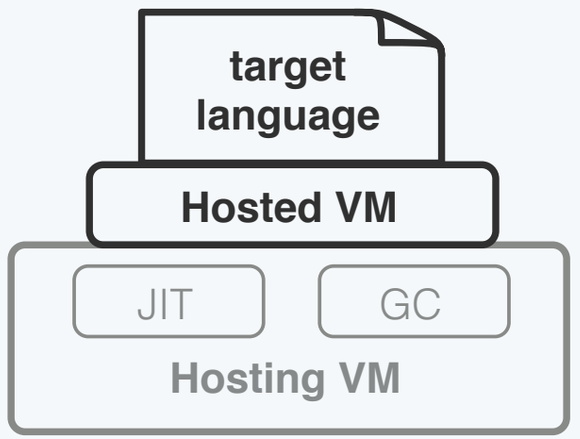
Architectural choice of dynamic language VMs



full-custom



interpreter-based



hosted

Hosted VM / interpreter for dynamic languages

- Full-custom VMs are costly to build and maintain
- Existing VMs offer mature and powerful components (JIT, GC)
- Interpreters are more cost-effective
- **Existing hosted VMs do not offer competitive performance**

UNIVERSITY OF CALIFORNIA

UCI

ZipPy is a hosted interpreter for Python3

- Built atop Truffle framework
- Supports the common feature of the language
- Open sourced at *https://bitbucket.org/ssllab/zippy*

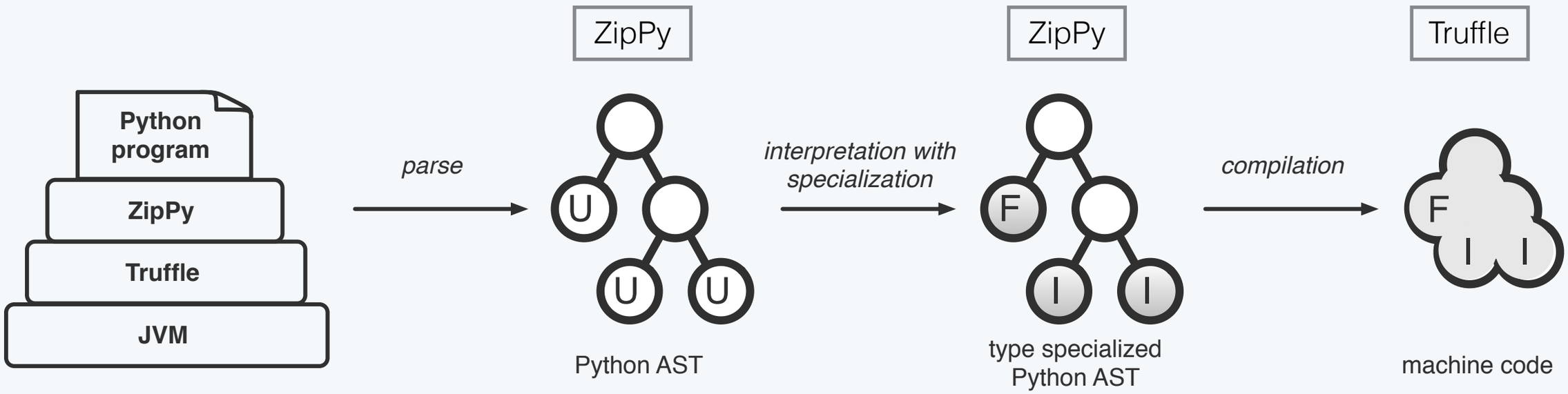
Truffle is a multi-language framework

- Facilitates AST interpreter construction
- Streamlines type specialization via AST node rewriting
- Bridges the guest interpreter with the underlying JIT compiler

UNIVERSITY OF CALIFORNIA

UCI

ZipPy on Truffle



U: Uninitialized **F**: Float **I**: Integer

Agenda

- *Trufflization*
  - ★ Generators optimizations
  - ★ Efficient object model for Python
    - ★ *our contributions*

UNIVERSITY OF CALIFORNIA

UCI

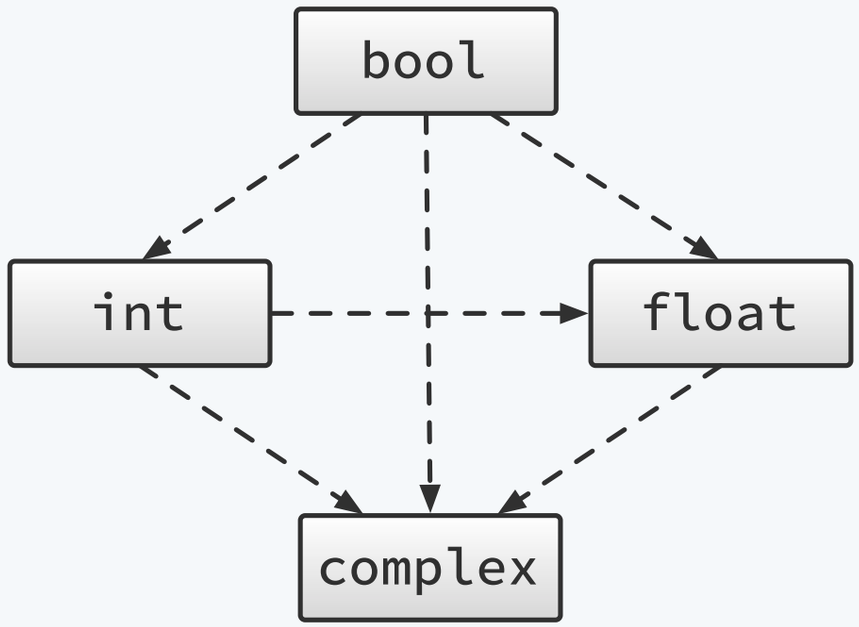
A for range loop example in Python

```
def sum(n):
  ttl = 0
  for i in range(n):
    ttl += i
  return ttl

print(sum(1000))
```

for range loop

addition

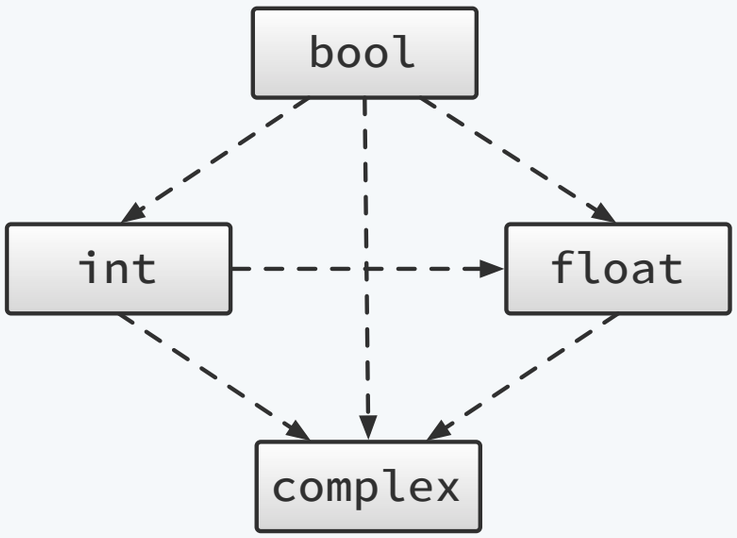
Numeric types in Python



int has arbitrary precision

type coercion

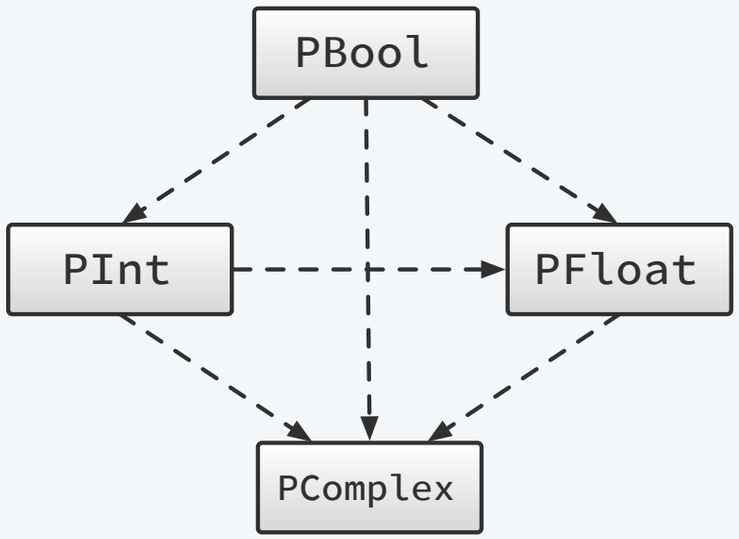
Numeric types in ZipPy



int has arbitrary precision

type coercion

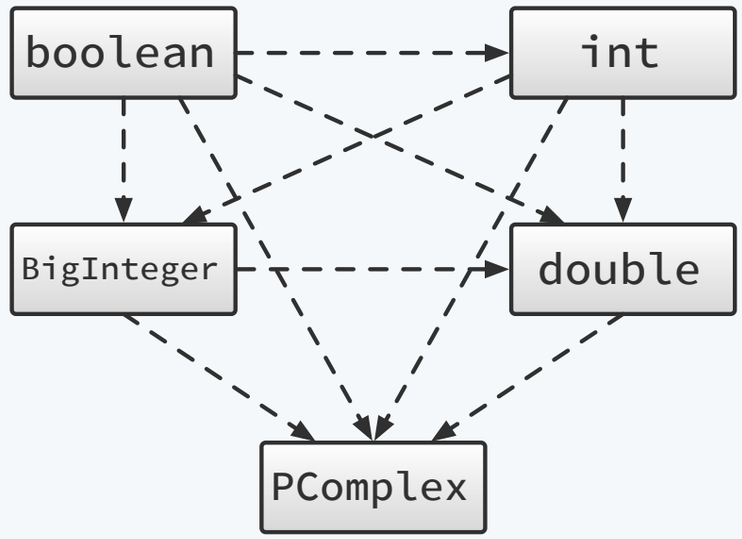
numeric types



PInt has arbitrary precision

type coercion

boxed representation



type coercion

unboxed representation

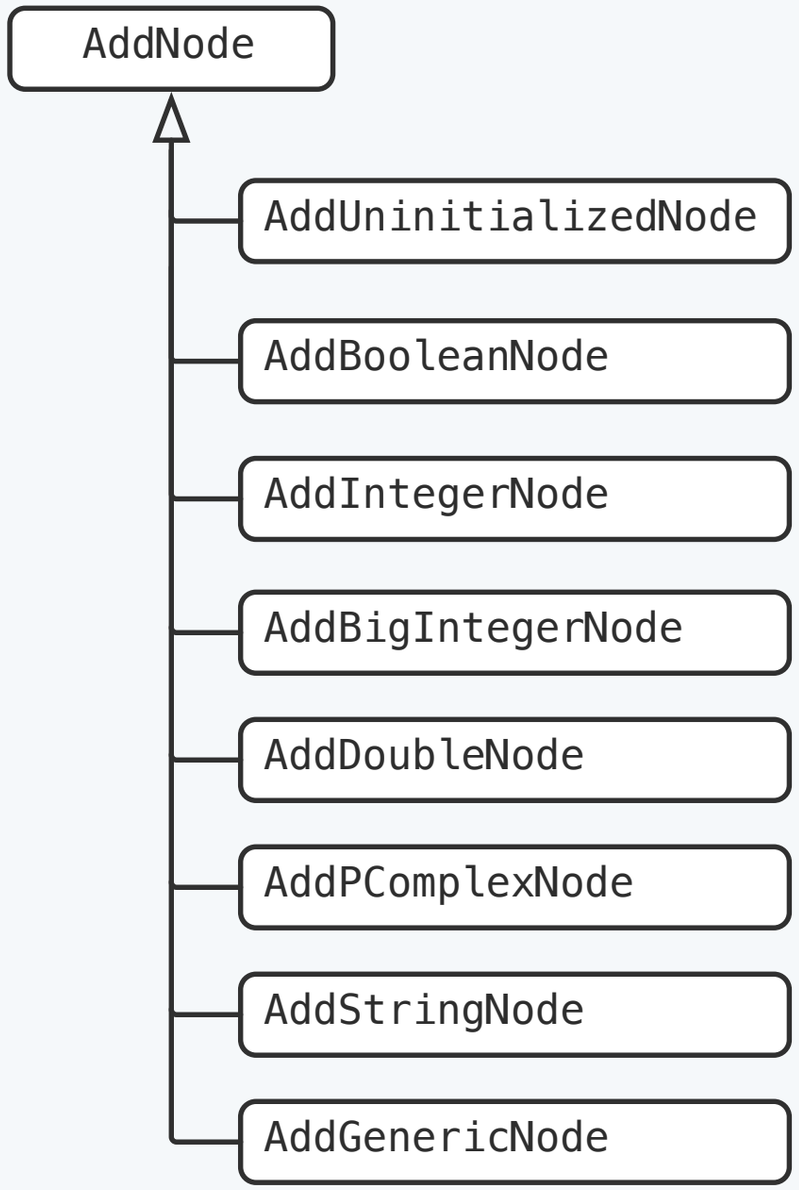
Type specialization for addition

```
abstract class AddNode extends BinaryArithmeticNode {

  @Specialization
  int doBoolean(boolean left, boolean right) {
    final int leftInt = left ? 1 : 0;
    final int rightInt = right ? 1 : 0;
    return leftInt + rightInt;
  }

  @Specialization(rewriteOn = ArithmeticException.class)
  int doInteger(int left, int right) {
    return ExactMath.addExact(left, right);
  }

  @Specialization
  BigInteger doBigInteger(BigInteger left, BigInteger right) {
    return left.add(right);
  }

  @Specialization
  double doDouble(double left, double right) {
    return left + right;
  }

  @Specialization
  PComplex doComplex(PComplex left, PComplex right) {
    return left.add(right);
  }

  @Specialization
  String doString(String left, String right) {
    return left + right;
  }
  //...
}
```

UNIVERSITY OF CALIFORNIA

UCI

AddNode derivatives

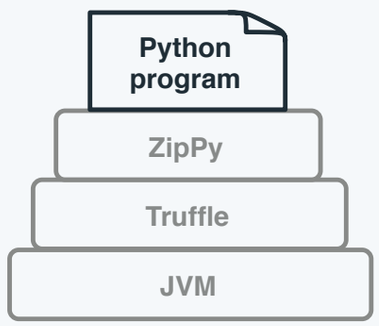


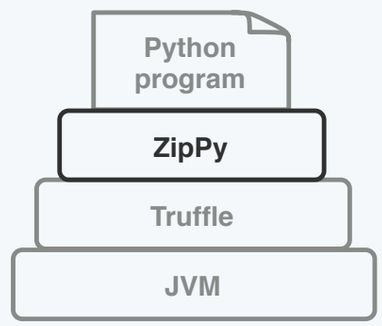
for-range loop in Python

```
def sum(n):
  ttl = 0
  for i in range(n):
    ttl += i
  return ttl
```

ForNode specialization for range iterator

```
class ForNode extends LoopNode {
  @Specialization
  public Object doPRange(VirtualFrame frame,
                         PRangeIterator range) {
    int start = range.getStart();
    int stop = range.getStop();
    int step = range.getStep();

    for (int i = start; i < stop; i += step) {
      ((WriteNode) target).executeWrite(frame, i);
      body.executeVoid(frame);
    }

    return PNone.NONE;
  }
}
```



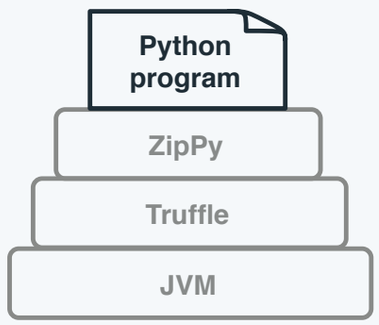


UNIVERSITY OF CALIFORNIA

UCI

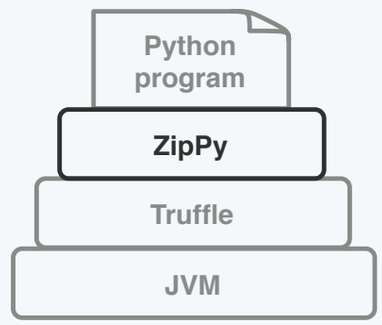
for-range loop in Python

```
def sum(n):
  ttl = 0
  for i in range(n):
    ttl += i
  return ttl
```



optimized for-range loop

```
public int sum(int n) {
  int ttl = 0;

  for (int i = 0; i < n; i++) {
    ttl += i;
  }

  return ttl;
}
```

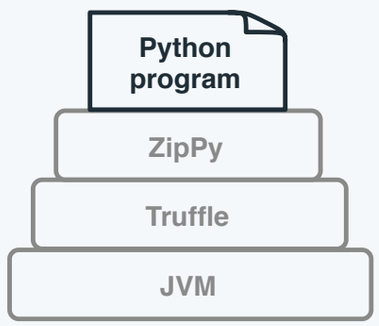


UNIVERSITY OF CALIFORNIA

UCI

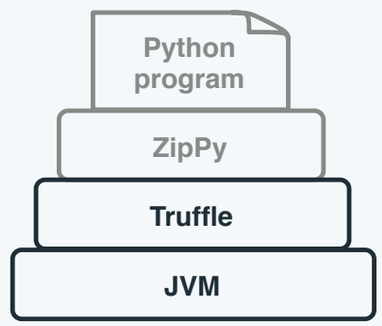
for-range loop in Python

```
def sum(n):
  ttl = 0
  for i in range(n):
    ttl += i
  return ttl
```



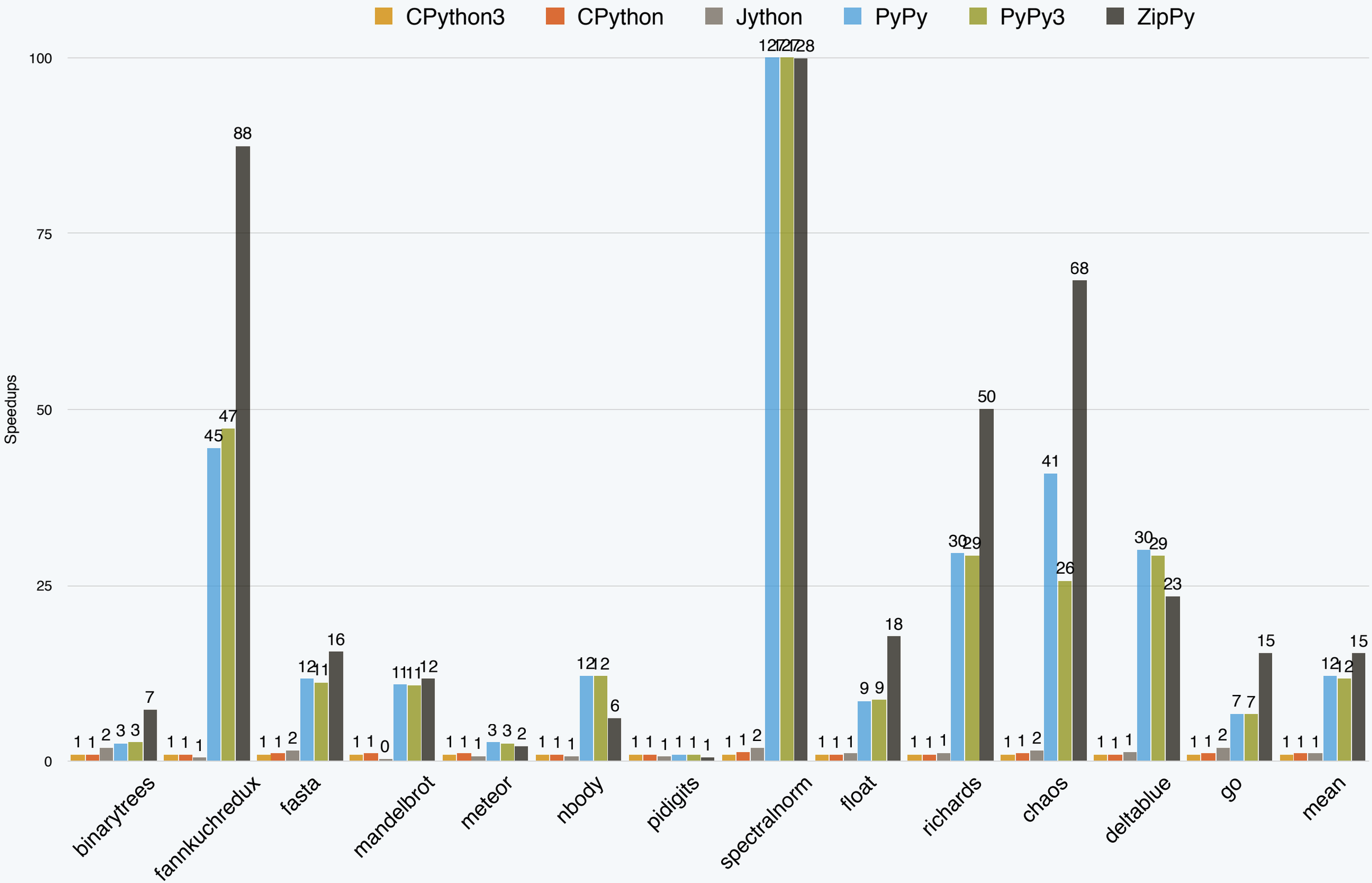
JIT compiled for range loop

```
        jmp L7

L6:     mov       ecx, edx
        add       ecx, ebp
        jo        L8
        mov       edx, ebp
        incl      edx
        mov       esi, ebp
        mov       ebp, edx
        mov       edx, ecx
L7:     cmp       eax, ebp
        jle       L9
        jmp       L6
L8:     call      deoptimize()

L9:
```



UNIVERSITY OF CALIFORNIA

UCI


CPython3
CPython
Jython
PyPy
PyPy3
ZipPy
Speedups
100
75
50
25
0
binarytrees
fannkuchredux
fasta
mandelbrot
meteor
nbody
pidigits
spectralnorm
float
richards
chaos
deltablue
go
mean

benchmmark	CPython3	CPython	Jython	PyPy	PyPy3	ZipPy
binarytrees	1.00	0.94	1.99	2.60	2.70	7.31
fannkuchredux	1.00	0.97	0.51	44.53	47.29	87.50
fasta	1.00	1.04	1.55	11.73	11.24	15.57
mandelbrot	1.00	1.08	0.34	10.91	10.82	11.69
meteor	1.00	1.02	0.77	2.64	2.62	2.13
nbody	1.00	0.97	0.73	12.13	12.06	6.17
pidigits	1.00	1.00	0.62	0.98	0.95	0.60
spectralnorm	1.00	1.33	1.89	127.33	127.25	128.10
float	1.00	0.95	1.05	8.64	8.67	17.71
richards	1.00	0.94	1.21	29.53	29.25	50.13
chaos	1.00	1.17	1.55	40.88	25.69	68.28
deltablue	1.00	0.85	1.33	30.08	29.14	23.46
go	1.00	1.08	1.99	6.79	6.66	15.41
mean	1.00	1.02	1.05	12.15	11.68	15.34

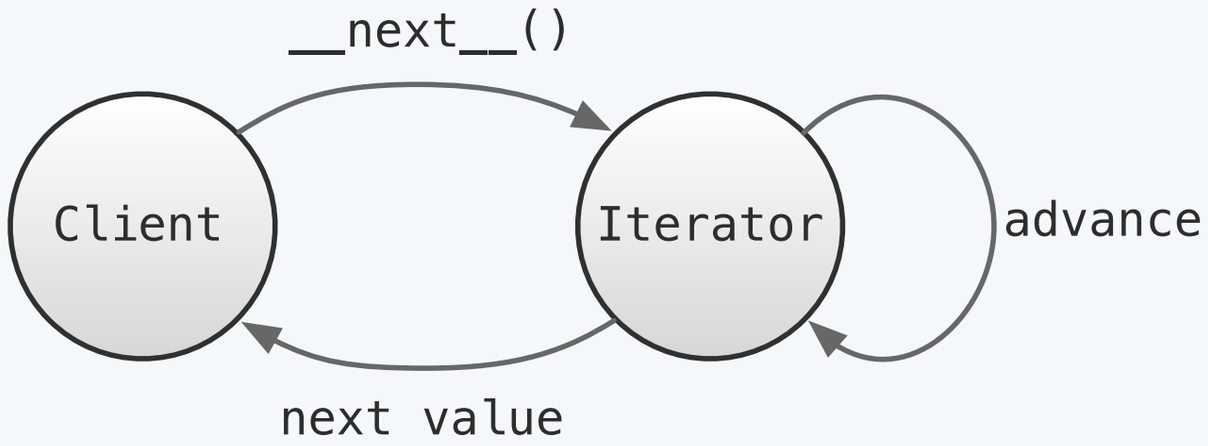
ZipPy is competitive with PyPy and a fast Python3 on the JVM

UNIVERSITY OF CALIFORNIA

UCI

Python iterators

- Iterators are ubiquitous
  - Implement iterator protocol
  - Built-in iterators
  - User-defined iterators
- **Generators** are user-defined iterators using special control-flow construct (yield)
- **Generators** exist in other languages too, like C#, PHP,…



Python generators

```
l = []
for i in fib(10):
  if i % 2 == 0:
    l.append(i)

# [2, 8, ..]
```

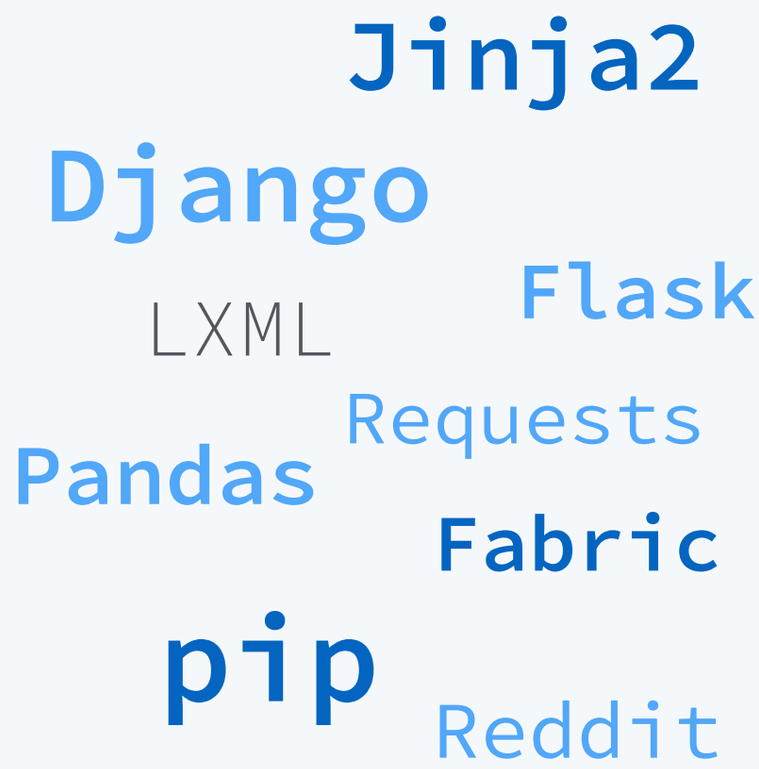
consumer loop

```
def fib(n):
  a, b = 0, 1
  for i in range(n):
    a, b = b, a+b
    yield a

# 1, 1, 2, 3, 5, 8..
```

generator function

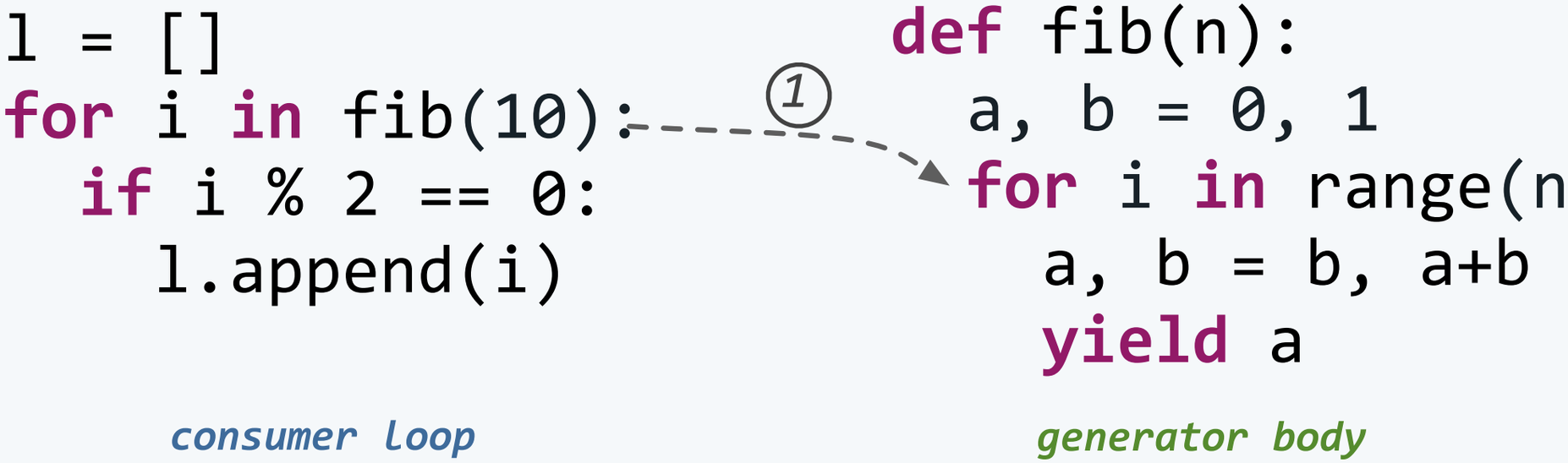
Python generators

- We surveyed the use of generators in Python programs
- 90% of the top 50 Python projects on PyPI and GitHub use generators
- Given its popularity the performance of generators are critical to Python programs



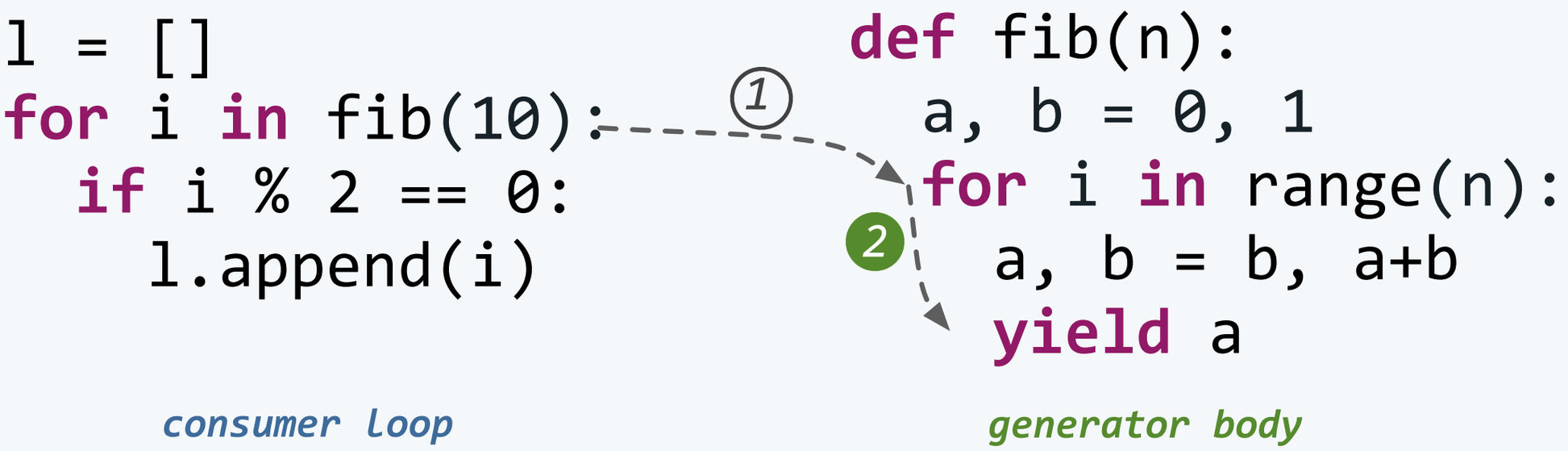
Python generators are slow…

Generator Execution



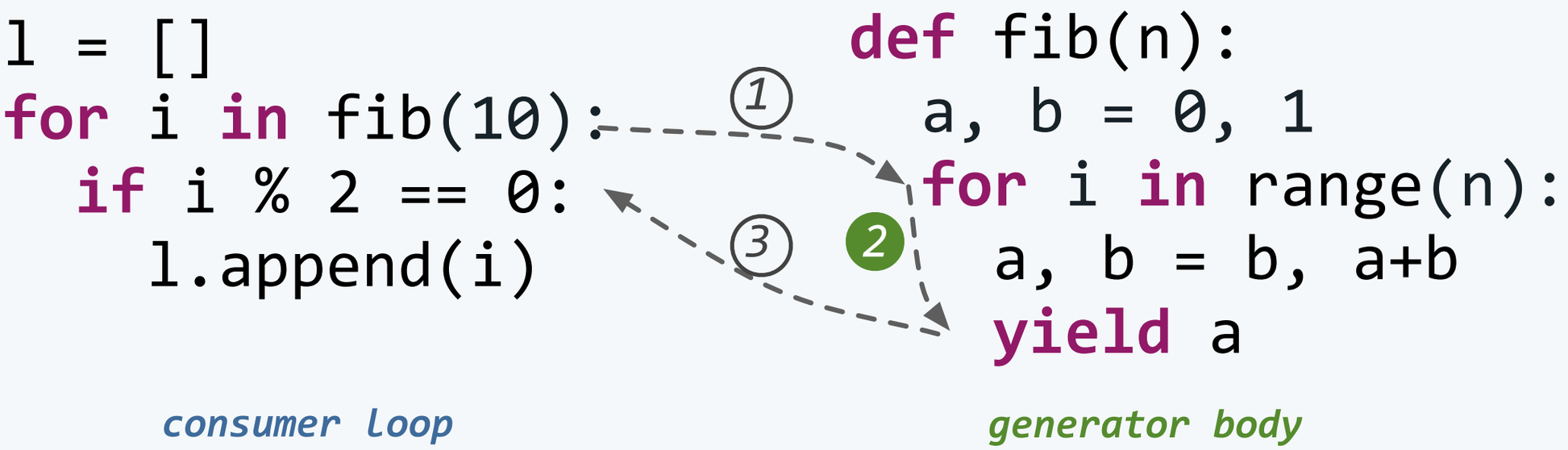
1.The implicit call to __*next*__ and resume execution

Generator Execution



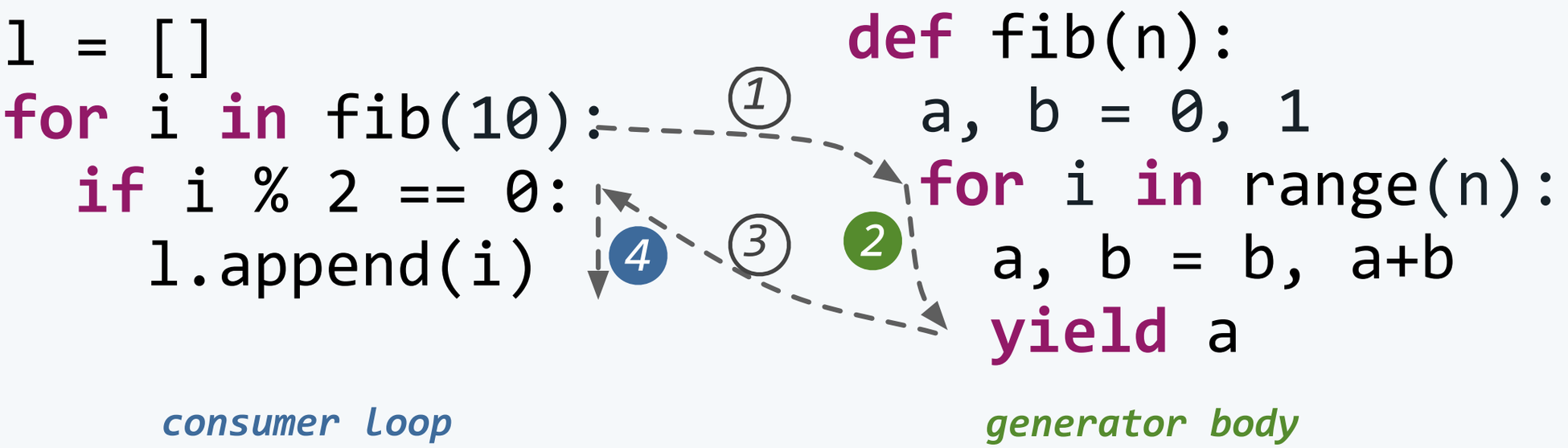
1. The implicit call to *__next__* and resume execution
2. Evaluate the next value in generator body

Generator Execution



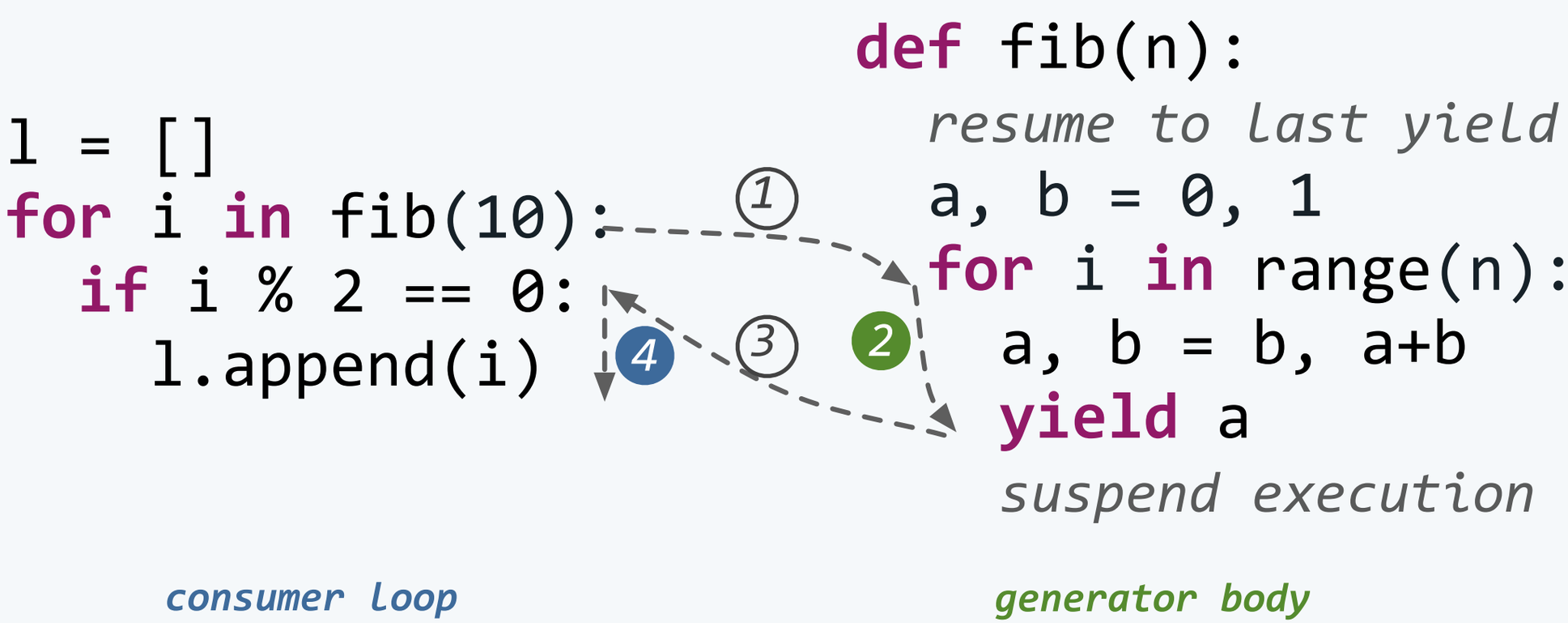
1. The implicit call to *__next__* and resume execution
2. Evaluate the next value in generator body
3. Suspend execution and return to the caller

Generator Execution



1. The implicit call to *__next__* and resume execution
2. Evaluate the next value in generator body
3. Suspend execution and return to the caller
4. Consume the generated value

Generator Overheads



- Only step 2 and 4 do the real work
- Python call is expensive
- Resume and suspend add additional costs and prevent frame optimizations

UNIVERSITY OF CALIFORNIA

UCI

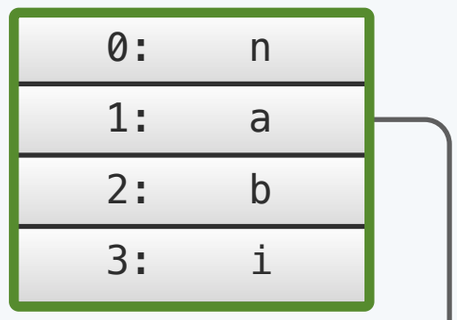
Naive Inlining

```
l = []
g = fib(10)
while True:
    resume to last yield
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
        yield a
        suspend execution
    i = a
    if i % 2 == 0:
        l.append(i)
except StopIter:
```

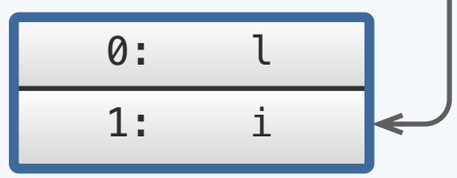
generator body

consumer loop

generator frame

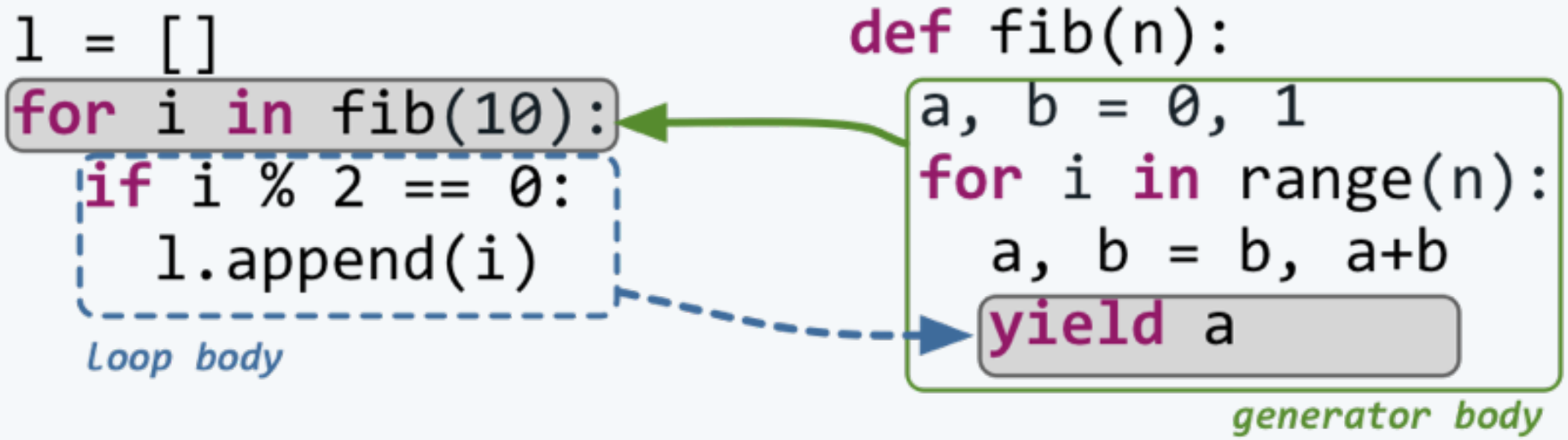


caller frame



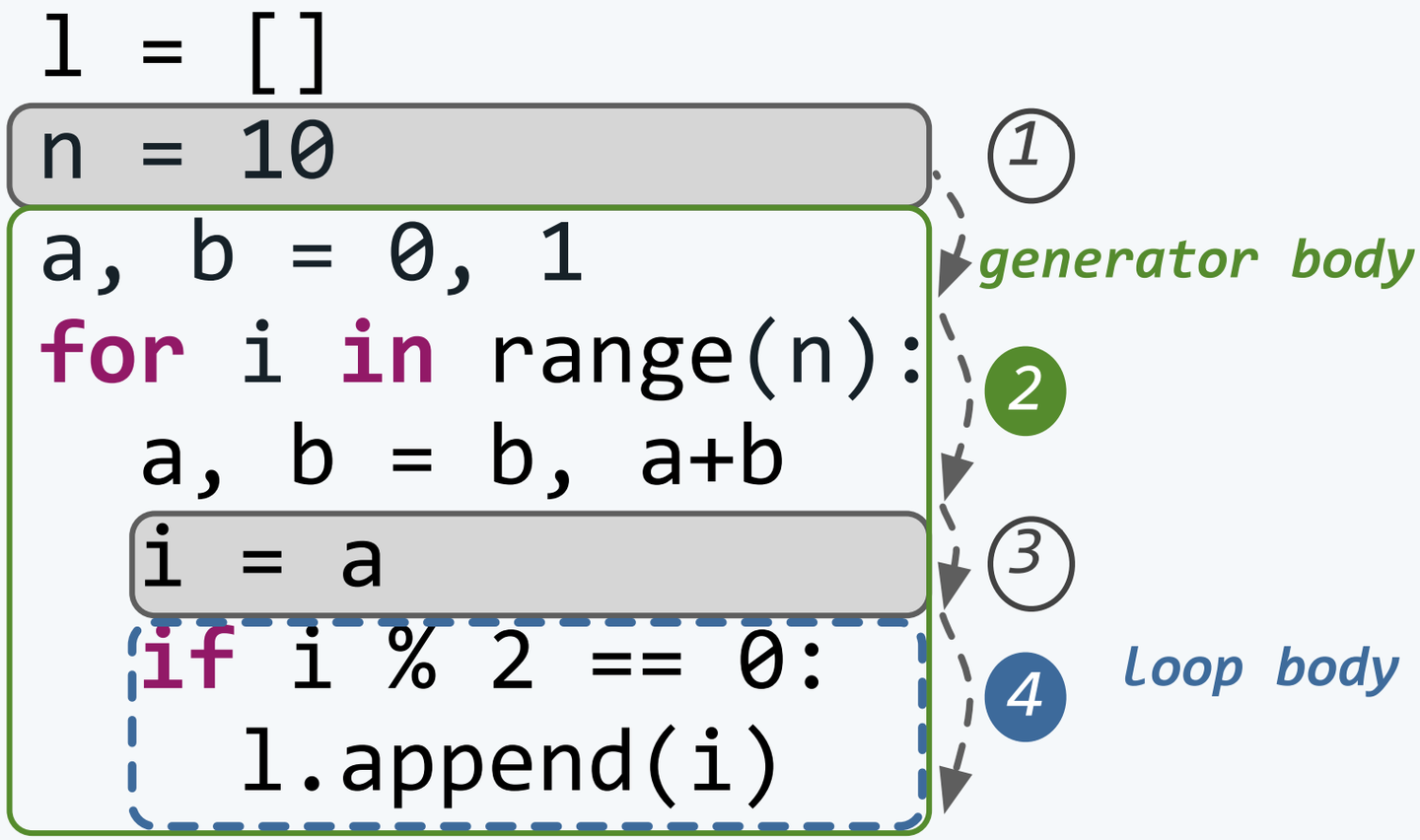
- Desugar the consumer loop and inline __next__ directly
- The suspend and resume handling still persists

Generator Peeling



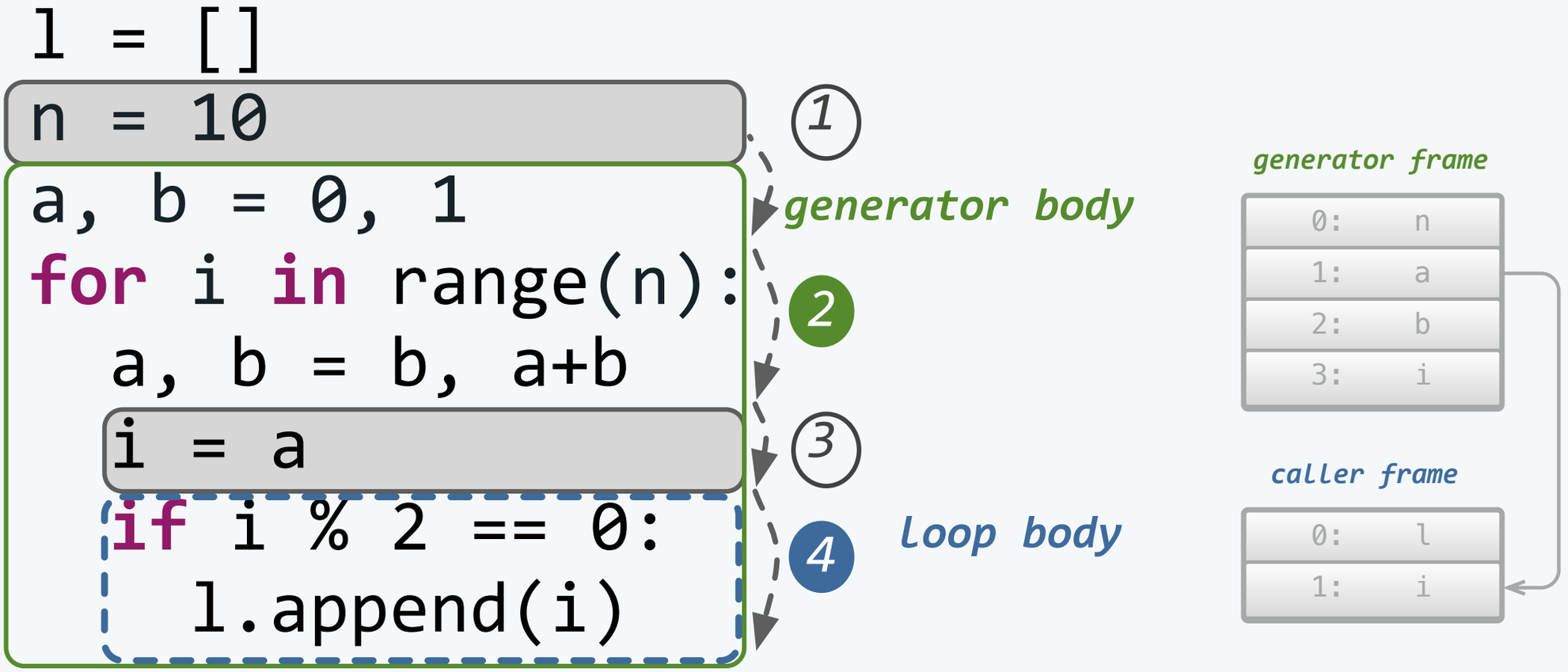
- Specialize the loop over generator at runtime
- Merge yield with consumer loop body

Generator Peeling



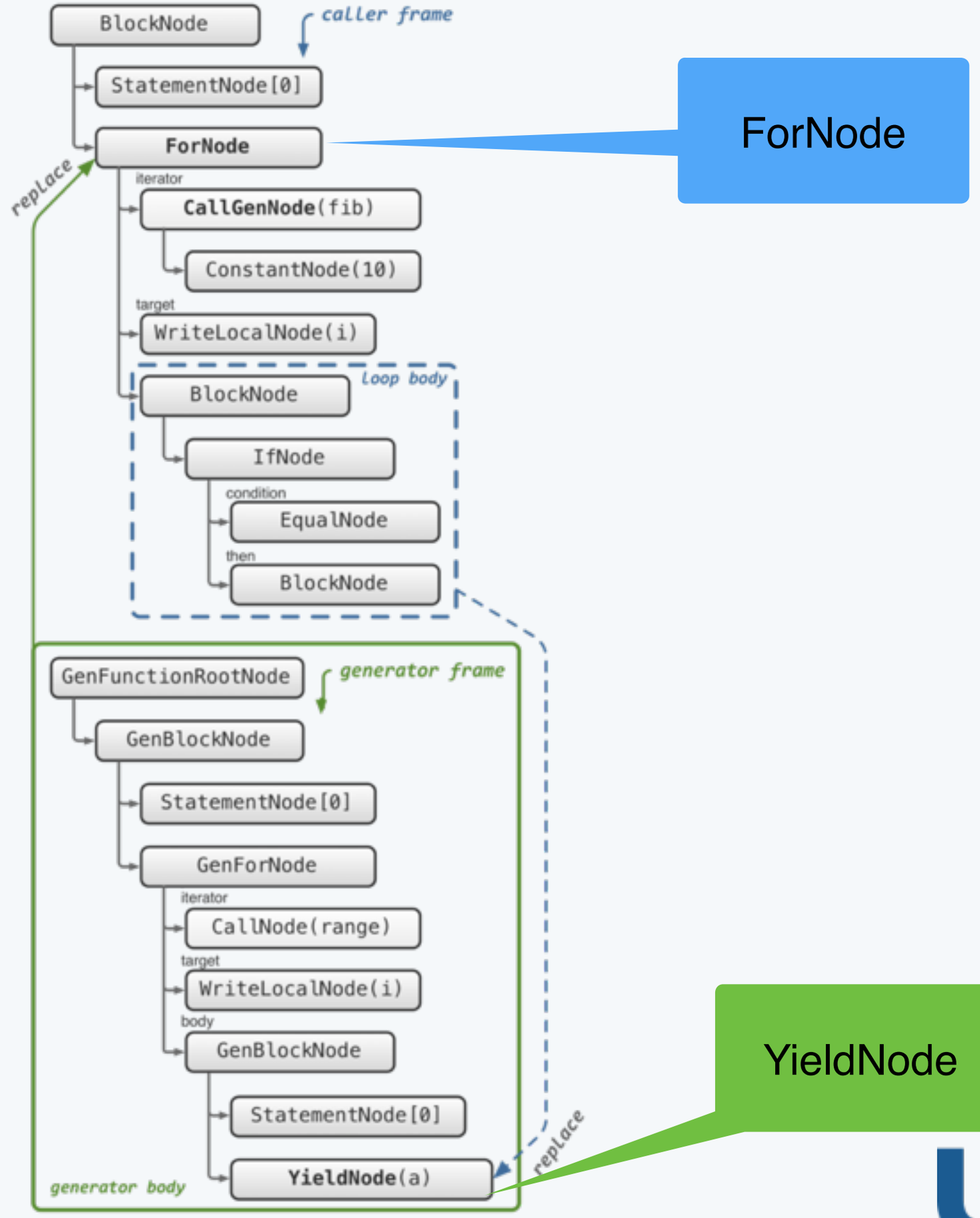
- Specialize the loop over generator at runtime
- Remove suspend and resume handling

Generator Peeling

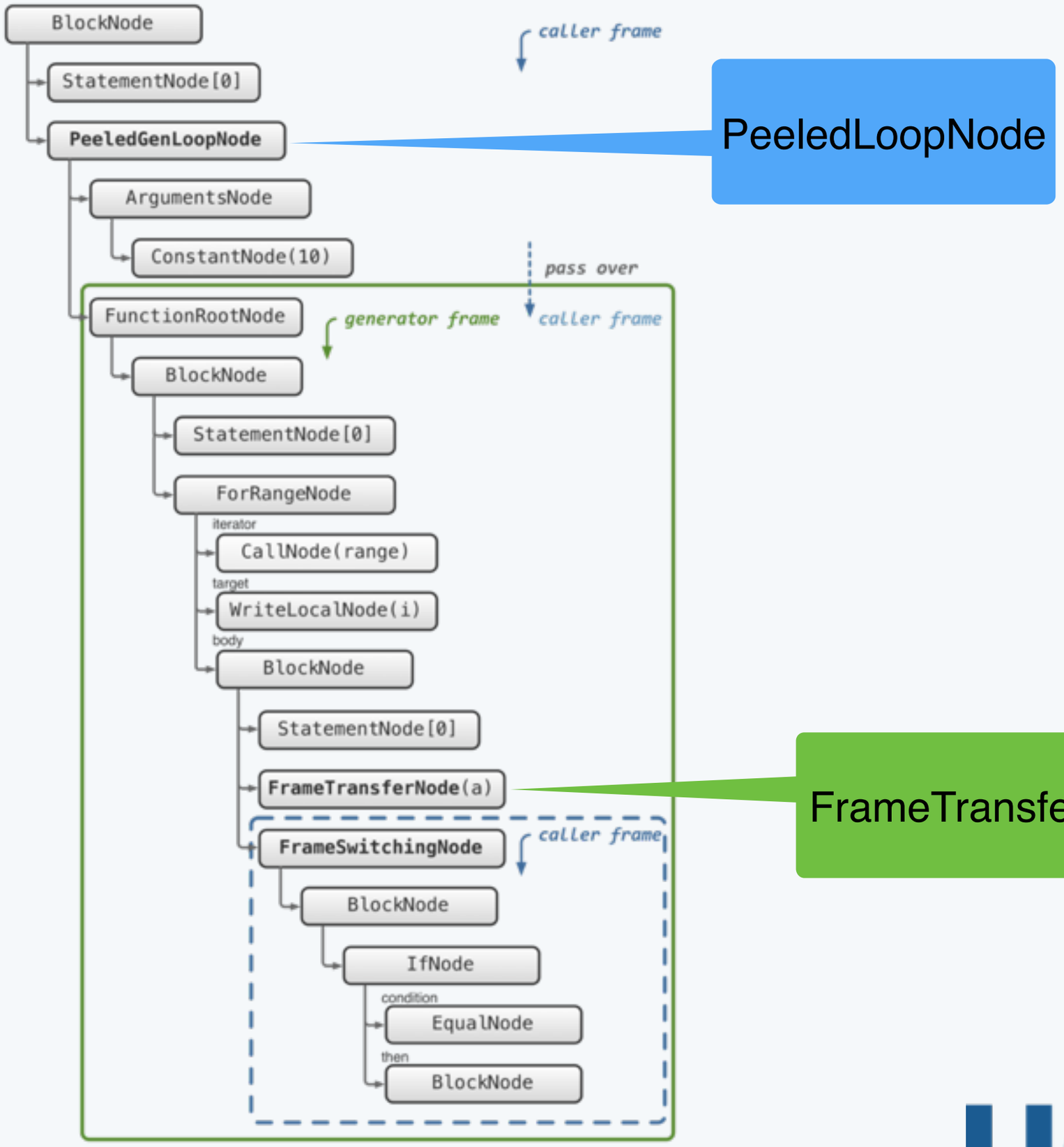


- Frames can be optimized during compilation

Before


BlockNode
caller frame
StatementNode[0]
ForNode
ForNode
replace
iterator
CallGenNode(fib)
ConstantNode(10)
target
WriteLocalNode(i)
loop body
BlockNode
IfNode
condition
EqualNode
then
BlockNode
GenFunctionRootNode
generator frame
GenBlockNode
StatementNode[0]
GenForNode
iterator
CallNode(range)
target
WriteLocalNode(i)
body
GenBlockNode
StatementNode[0]
YieldNode(a)
generator body
replace
YieldNode

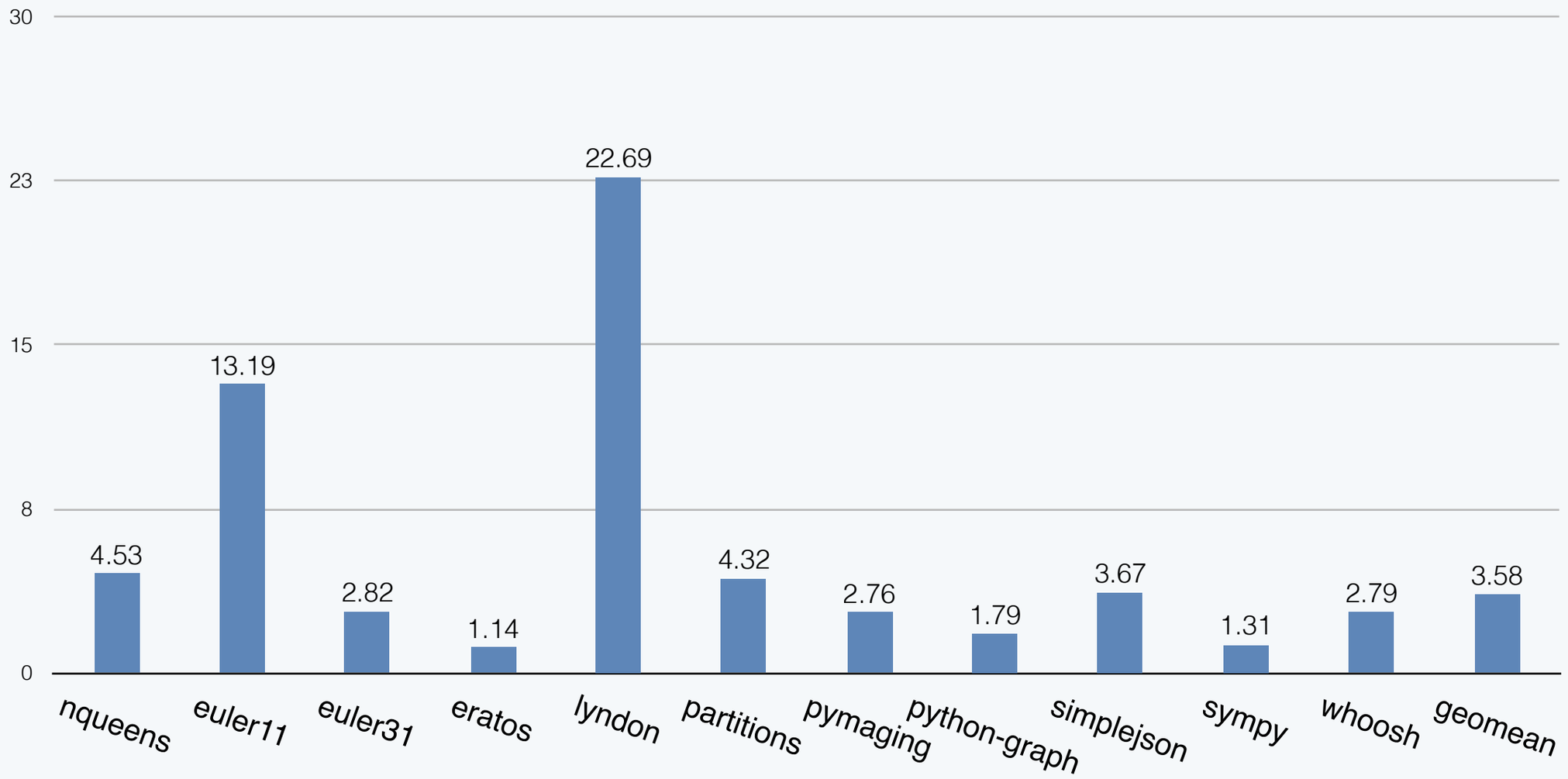
After


BlockNode
caller frame
StatementNode[0]
PeeledGenLoopNode
PeeledLoopNode
ArgumentsNode
ConstantNode(10)
pass over
FunctionRootNode
generator frame
caller frame
BlockNode
StatementNode[0]
ForRangeNode
iterator
CallNode(range)
target
WriteLocalNode(i)
body
BlockNode
StatementNode[0]
FrameTransferNode(a)
FrameTransferNode
FrameSwitchingNode
caller frame
BlockNode
IfNode
condition
EqualNode
then
BlockNode

The End Result

- Caller frame and generator frame can be optimized
- Peeling inlines the call to __next__
- No suspend and resume handling
- AST level transformation, independent from compilation

Speedups of Generator Peeling

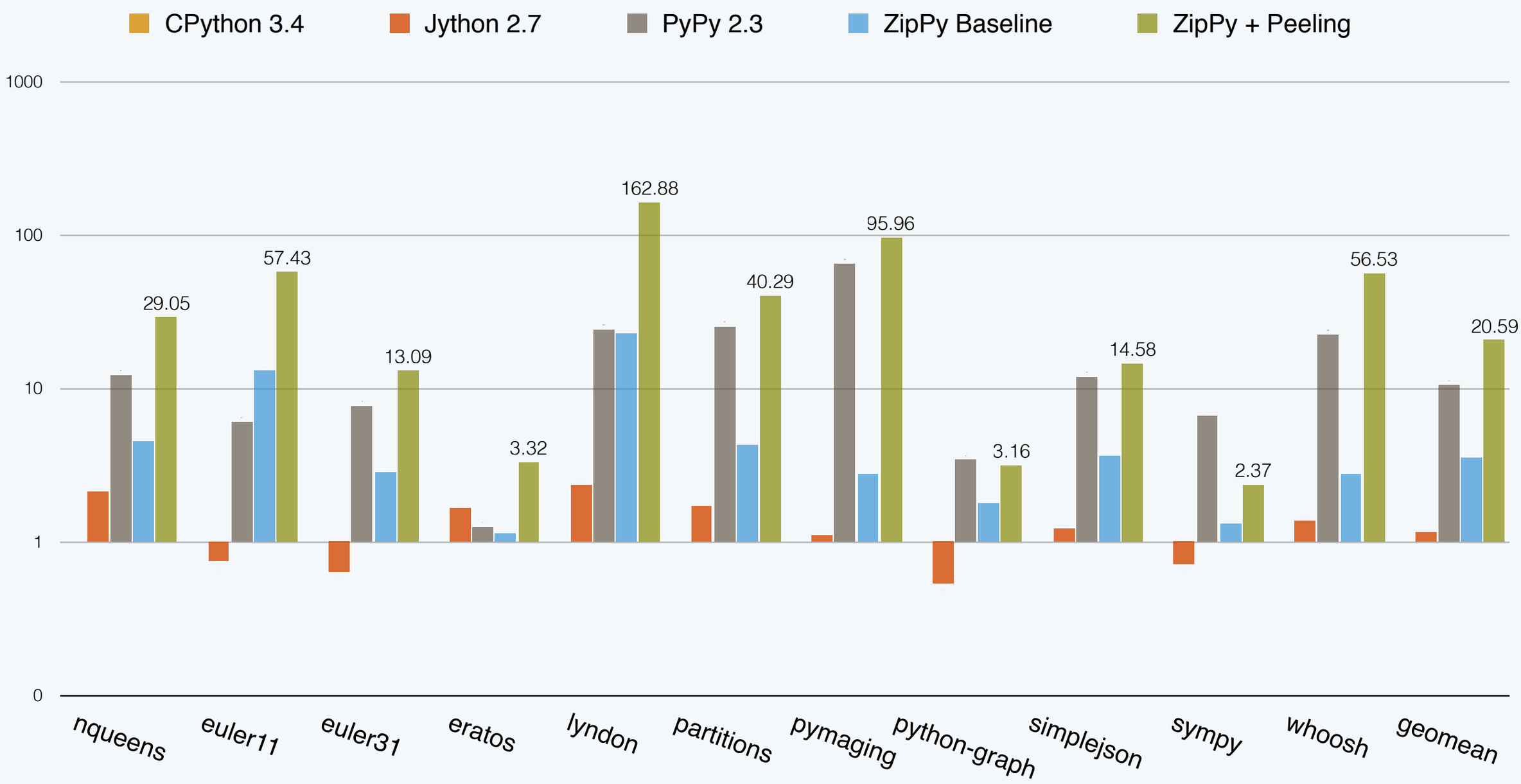


Measuring peak performance of ZipPy with and without Generator Peeling

UNIVERSITY OF CALIFORNIA

UCI

The Performance of ZipPy



Measuring peak performance of ZipPy with Generator Peeling

UNIVERSITY OF CALIFORNIA

UCI

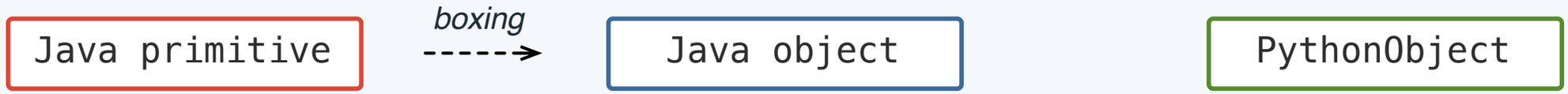
Generator peeling conclusions

- We present a dynamic program transformation that optimizes generators for optimizing AST interpreters
- Not restricted to ZipPy or Python
- As a result, programmers are free to enjoy generators' upsides

UNIVERSITY OF CALIFORNIA

UCI

Object model for dynamic languages



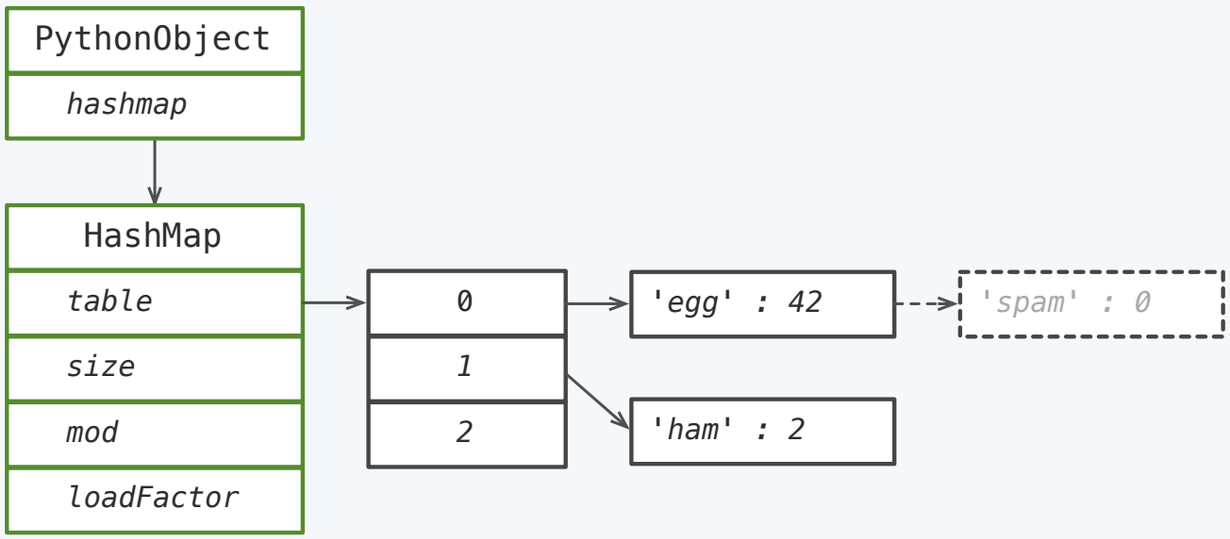
1: built-in numeric types

2: built-in immutable types

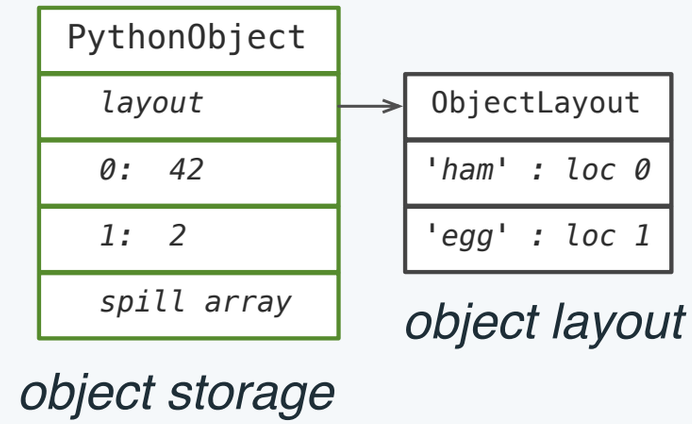
3: custom mutable types

Multiple data representations for built-in and custom types

Modeling mutable object in Python



HashMap based approach

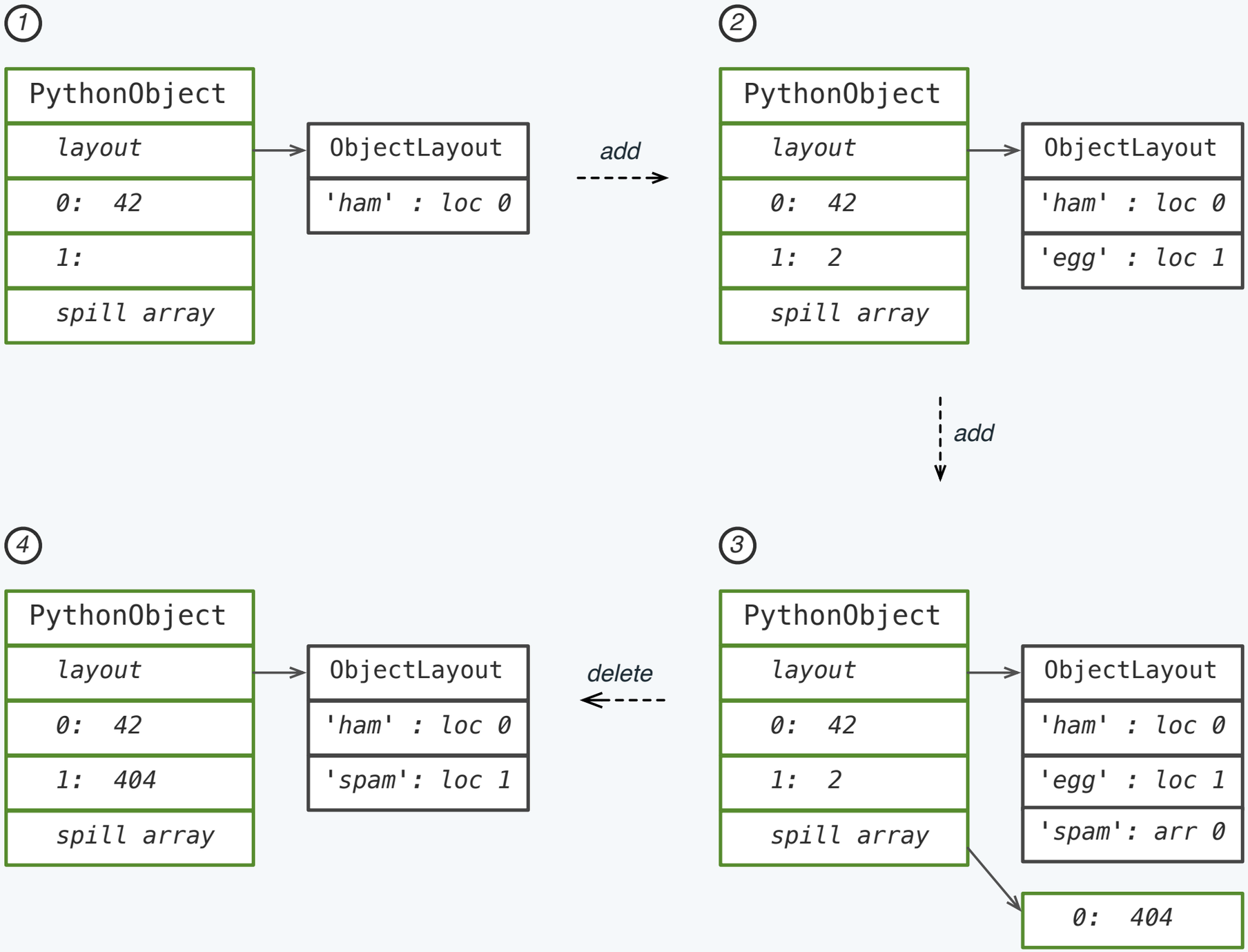


Hidden class approach

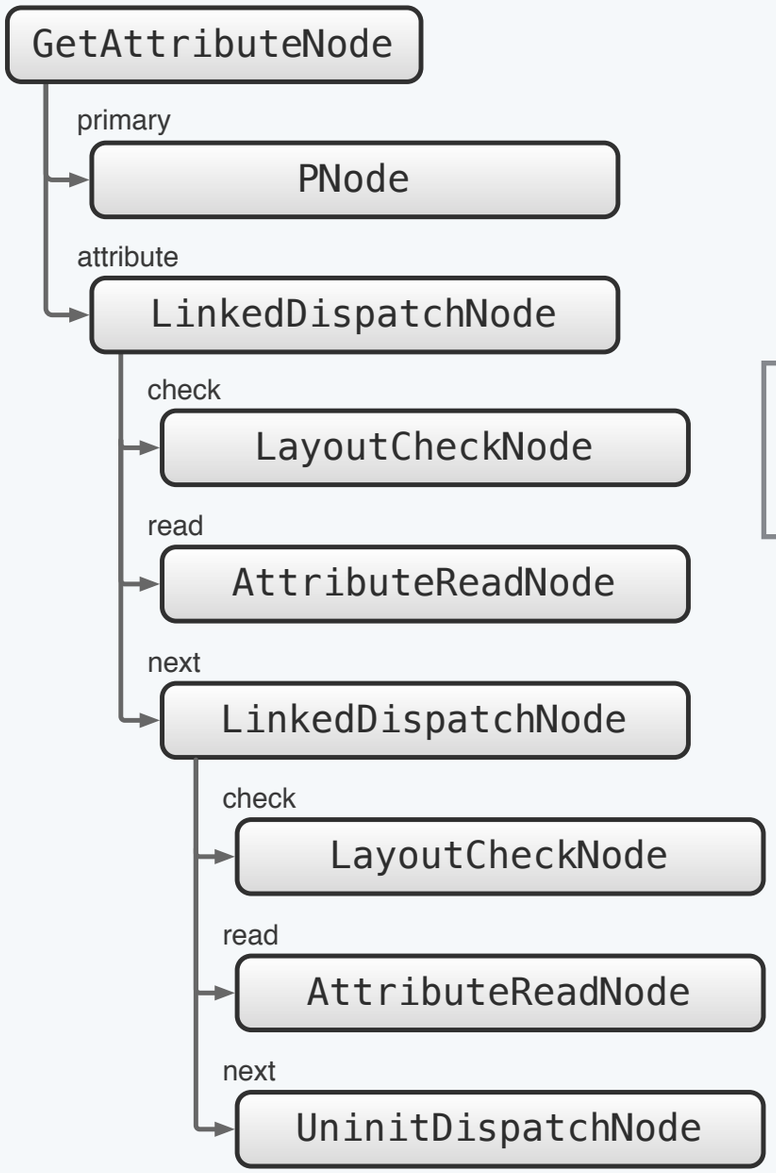
Implementation of object storage class

```
class FixedPythonObjectStorage extends PythonObject {

    static final int INT_LOCATIONS_COUNT = 5;
    protected int primitiveInt0;
    protected int primitiveInt1;
    protected int primitiveInt2;
    protected int primitiveInt3;
    protected int primitiveInt4;

    static final int DOUBLE_LOCATIONS_COUNT = 5;
    protected double primitiveDouble0;
    protected double primitiveDouble1;
    protected double primitiveDouble2;
    protected double primitiveDouble3;
    protected double primitiveDouble4;

    static final int OBJECT_LOCATIONS_COUNT = 5;
    protected Object fieldObject0;
    protected Object fieldObject1;
    protected Object fieldObject2;
    protected Object fieldObject3;
    protected Object fieldObject4;

    protected Object[] objectsArray = null;

    public FixedPythonObjectStorage(PythonClass pythonClass) {
        super(pythonClass);
    }
}
```

Implementation of object storage class



Inline caching for object accesses



dispatch chain

```
cmp $0xe830f77b,r11d    ; ObjectLayout
jne 0x00000001102e8ee9 ; next dispatch
mov rdi,0x640(%rsp)
```

JIT compiled dispatch node

Flexible storage class generation

```python
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

p = Point(1.2, 0.3)
# p.x == 1.2; p.y == 0.3
```

Python class Point

```
class Point extends FlexiblePythonObjectStorage {

    protected double x;
    protected double y;

    protected Object[] objectsArray = null;

    public Point(PythonClass pythonClass) {
        super(pythonClass);
    }
}
```

generated storage class for Point

Python object layout change

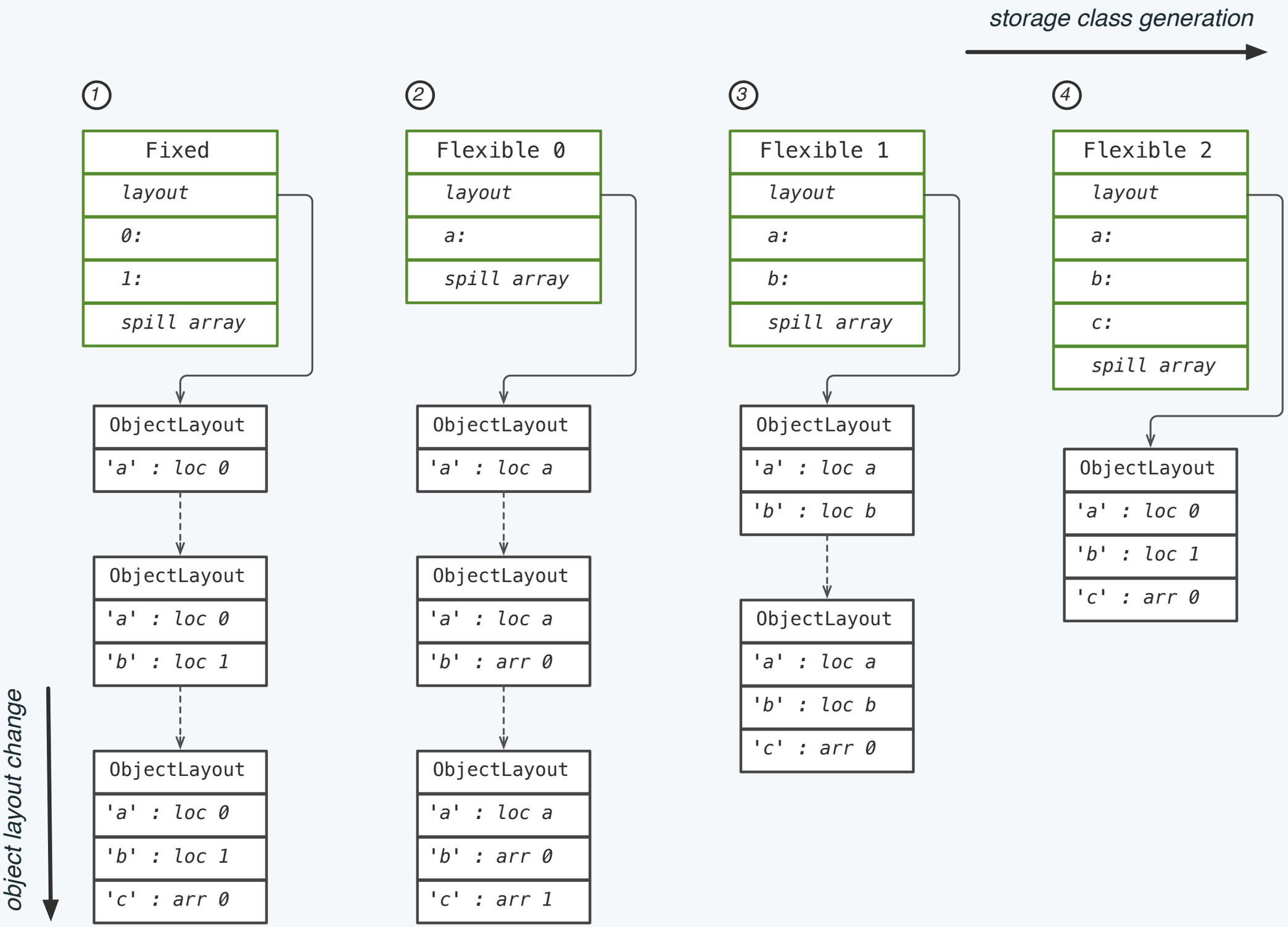
```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def addNeighbor(self, n):
    self.neighbors = n
```

Python class Point

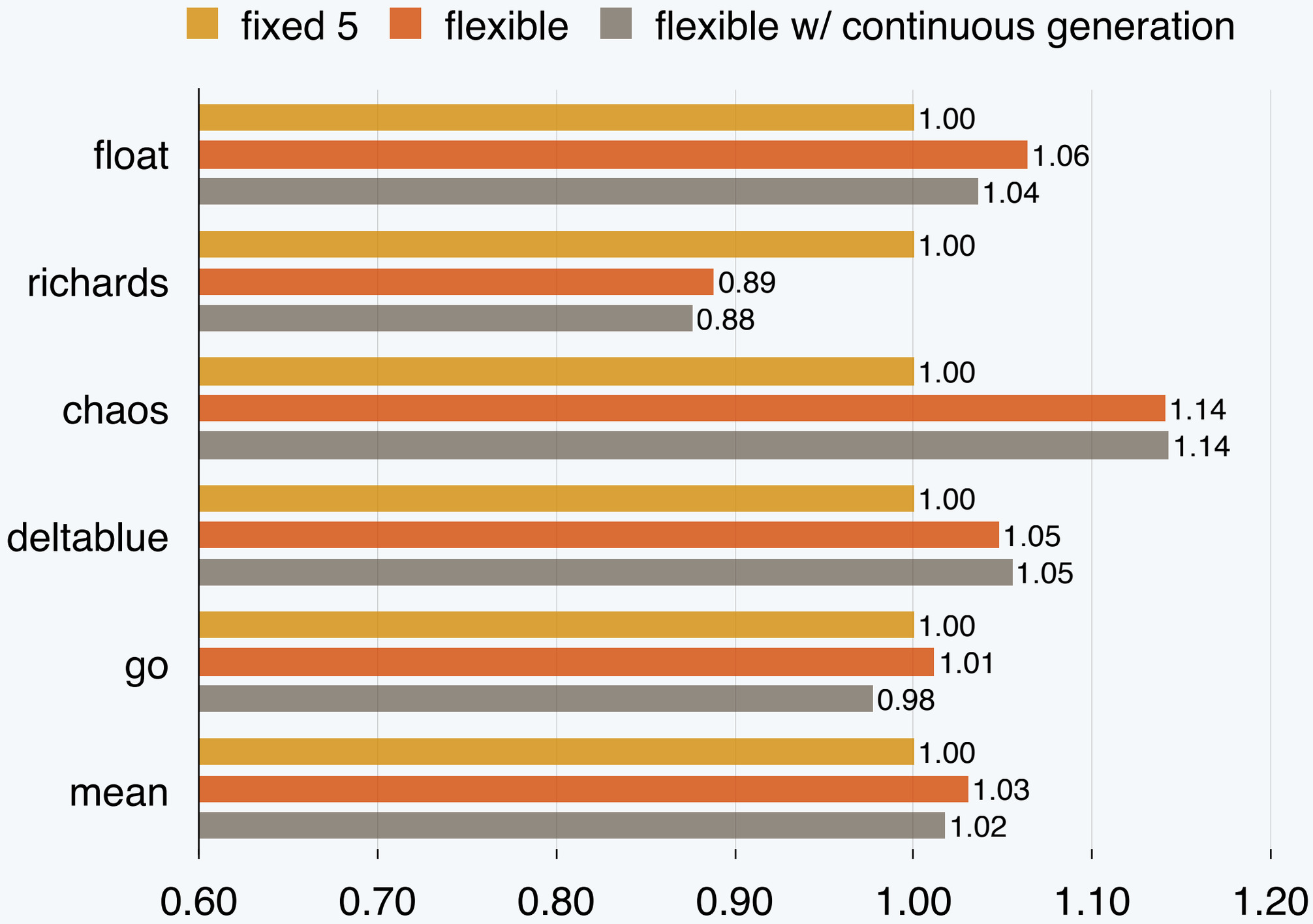
```
n = []

for i in range(5):
  p = Point(i*1.0, i*0.5)
  p.addNeighbors(n)
  n.append(p)
```

client code

Continuous storage class generation



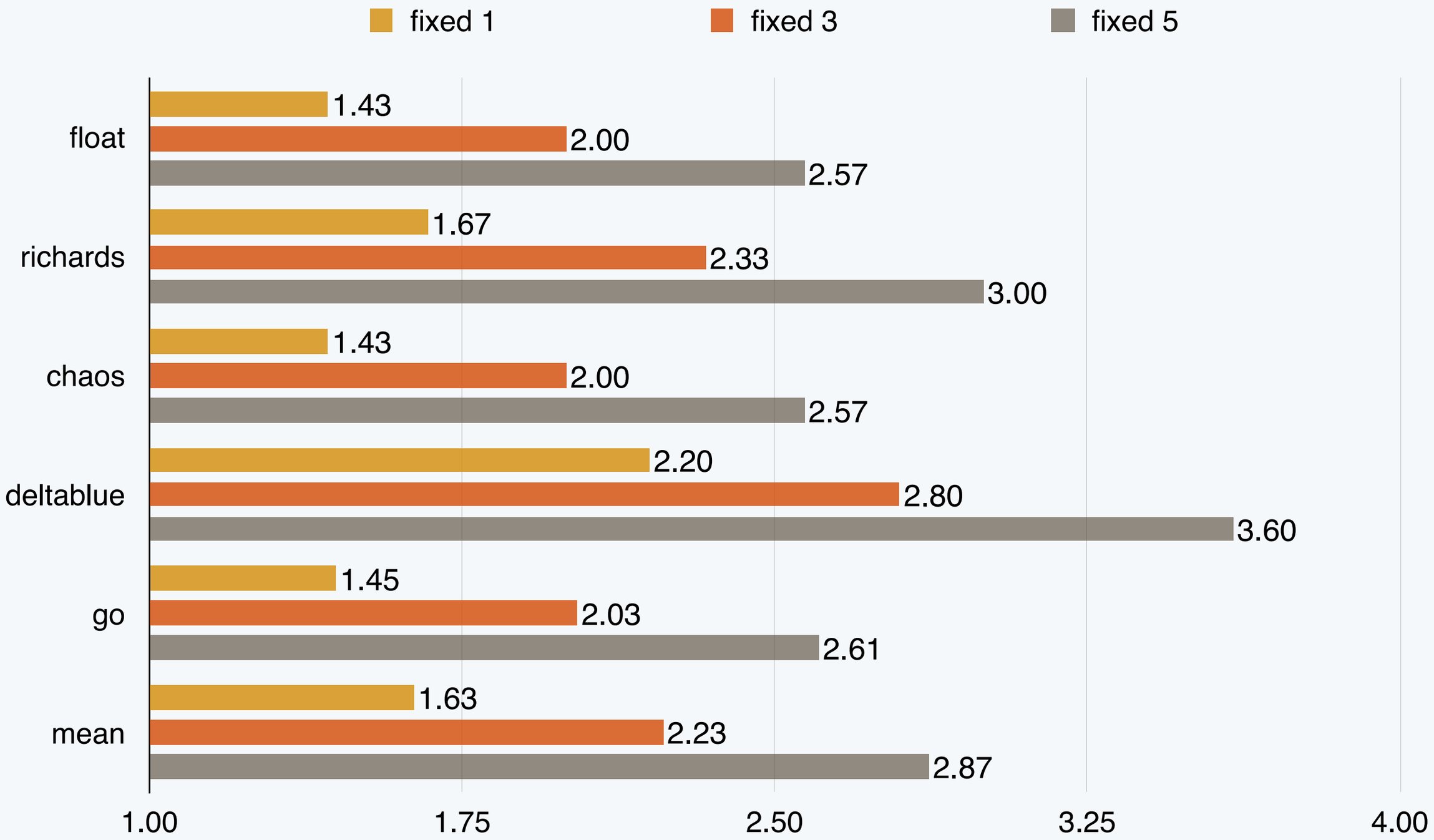
Performance of different object storage configurations



UNIVERSITY OF CALIFORNIA

UCI

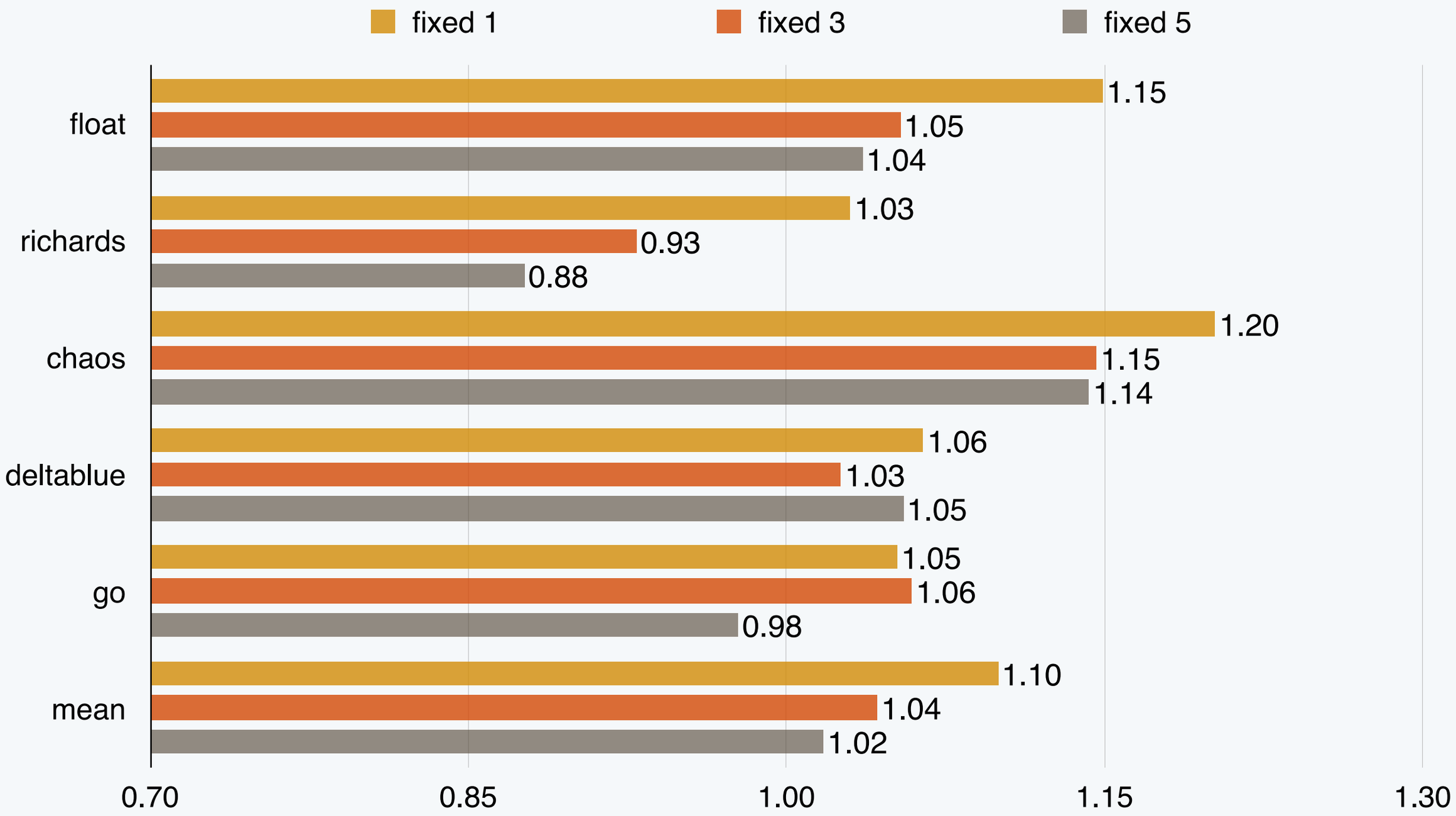
Memory usage of fixed object storages normalized to flexible object storage



UNIVERSITY OF CALIFORNIA

UCI

Slowdown of fixed object storages normalized to flexible object storage



UNIVERSITY OF CALIFORNIA

UCI

Flexible object storage conclusions

- There is always a trade-off when using fixed object storage
- Fixed object storage leads up to 20% loss on performance or 3.6x more memory usage
- Flexible object storage always optimizes the current state of the target Python class
- The coexistence of multiple storage classes can introduce overhead

UNIVERSITY OF CALIFORNIA

UCI

Our contributions

- Generator peeling: a runtime optimization targeting hosted interpreters
- It is not restricted to Python or the implementation of ZipPy/Truffle
- Flexible object storage: a space efficient object model technique for class-based dynamic languages
- Can be reused by other languages hosted on the JVM

Publications

- Wei Zhang, Per Larsen, Stefan Brunthaler, Michael Franz. **Accelerating Iterators in Optimizing AST Interpreters**. In *Proceedings of the 29th ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications, Portland, OR, USA, October 20-24, 2014 (OOPSLA '14), 2014.*

- Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, Michael Franz. **Efficient Hosted Interpreters on the JVM**. In *ACM Transactions on Architecture and Code Optimization, volume 11(1) pages 9:1–9:24, 2014.*

- Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Chen Li, Stefan Brunthaler, Per Larsen, Michael Franz. **Efficient Interpreter Optimizations for the JVM**. In *Proceedings of the 10th International Conference on Principles and Practice of Programming in Java, Stuttgart, Germany, September 11-13, 2013 (PPPJ '13), 2013.*

Question Please?