Pycket

A tracing JIT
For a functional language

Sam Tobin-Hochstadt
Indiana University

EPFL, 9/28/16

	AOT	JIT
OO		v8, Self HotSpot
FP	GHC, Gambit MLton, SBCL	

	AOT	JIT
OO	GCJ ...	v8, Self HotSpot
FP	GHC, Gambit MLton, SBCL	

	AOT	JIT
OO	GCJ ...	v8, Self HotSpot
FP	GHC, Gambit MLton, SBCL	???

A simple program

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

A simple program

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

506 ms (size 10000000)

A simple program

```
(define (dot u v)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

39 ms (size 10000000)

A simple program

```
(define (dot v1 v2)
  (define len (flvector-length v1))
  (unless (= len (flvector-length v2))
    (error 'fail))
  (let loop ([n 0] [sum 0.0])
    (if (unsafe-fx= len n) sum
        (loop (unsafe-fx+ n 1)
              (unsafe-fl+
               sum (unsafe-fl*
                    (unsafe-flvector-ref v1 n)
                    (unsafe-flvector-ref v1 n)))))))
```

29 ms (size 1000000)

A simple program

```
(define/contract (dot u v)
  ((vectorof flonum?) (vectorof flonum?)
   . -> . flonum?)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

933 ms (size 10000000)

Success?

✓ Fast code

✓ Generic operations and contracts

✗ You can only pick one

What do contracts and generic functions have in common?

What do contracts and generic functions have in common?

Indirection

MAYBE YOU CAN
HAVE YOUR CAKE
AND EAT IT TOO.

+ pypy

=

With added cake ...

```
(define (dot v1 v2)
  (define len (flvector-length v1))
  (unless (= len (flvector-length v2))
    (error 'fail))
  (let loop ([n 0] [sum 0.0])
    (if (unsafe-fx= len n) sum
        (loop (unsafe-fx+ n 1)
              (unsafe-fl+
               sum (unsafe-fl*
                    (unsafe-flvector-ref v1 n)
                    (unsafe-flvector-ref v1 n)))))))
```

8 ms (size 10000000)

With added cake …

```
(define (dot u v)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

11 ms (size 1000000)

With added cake ...

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

12 ms (size 10000000)

With added cake ...

```
(define/contract (dot u v)
  ((vectorof flonum?) (vectorof flonum?)
    . -> . flonum?)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

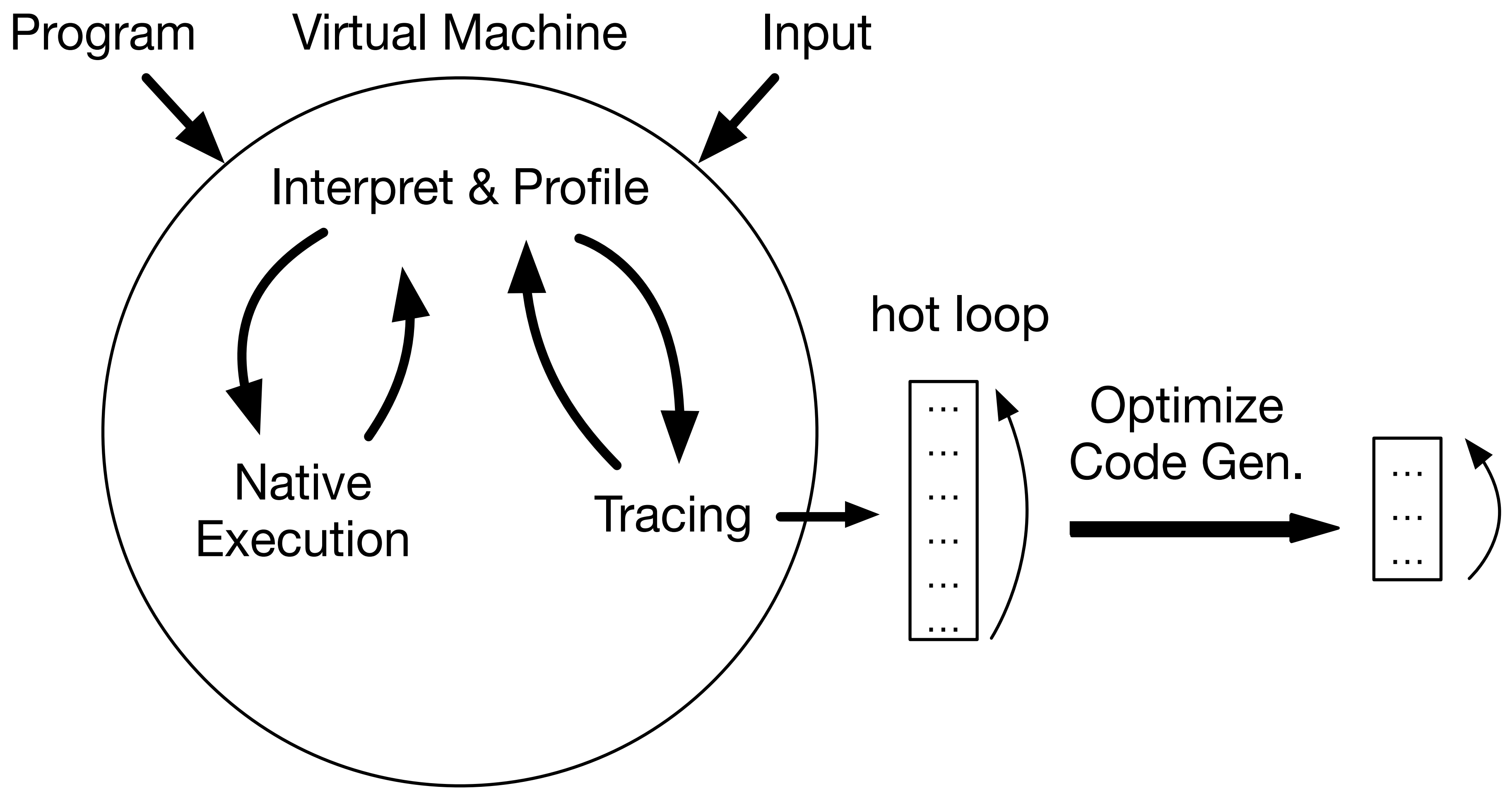
17 ms (size 1000000)

How does it work?

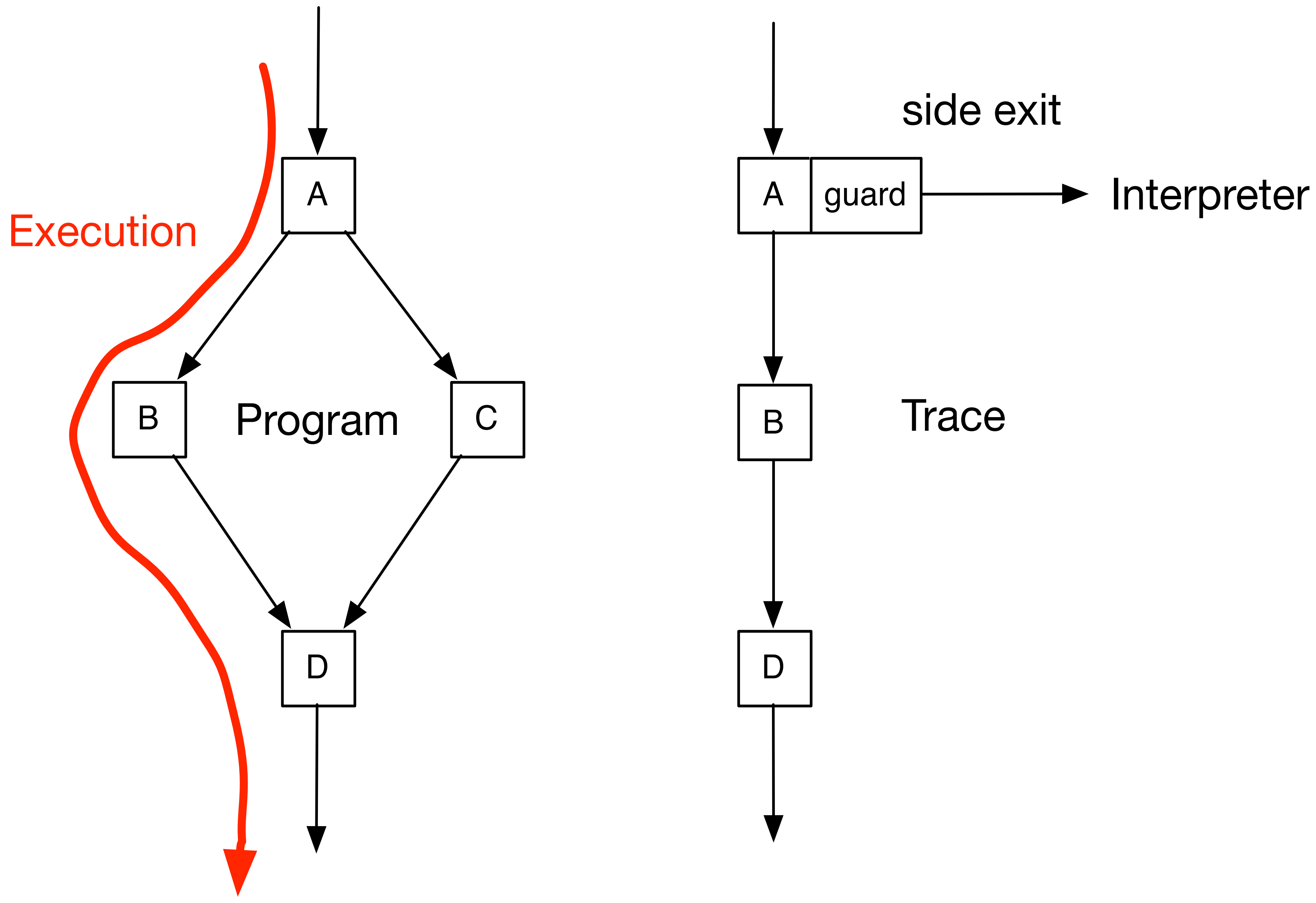
Tracing JIT

1. Interpret Program
2. Find hot loop
3. Record operations for one iteration
4. Optimize
5. Switch to new code

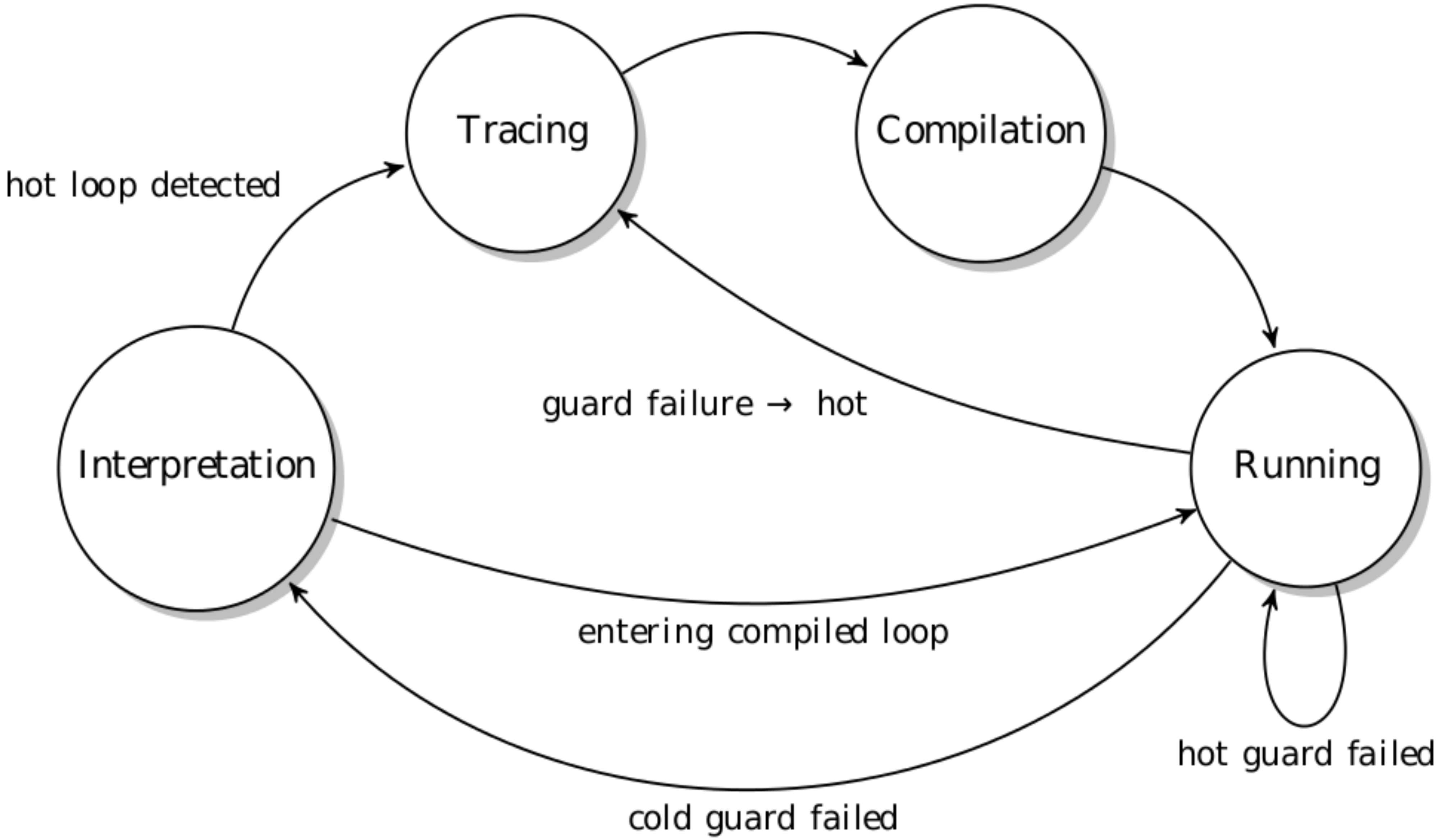
Tracing JIT


Program
Virtual Machine
Input
Interpret & Profile
Native Execution
Tracing
hot loop
...
Optimize Code Gen.

Tracing JIT


Execution
A
B
Program
C
D
side exit
A
guard
Interpreter
B
Trace
D

Tracing JIT



(Diagram from Antonio Cuni)

Resulting Optimizations

Inlining (happens for free)

Constant propagation

Allocation Removal

Dot product Inner Loop

```
label(acc, idx1, idx2, len1, len2, arr1, arr2)
 check loop counters
guard(idx1 < len1)
guard(idx2 < len2)
 fetch elements
val1     = getarrayitem_gc(arr1, idx1)
val2     = getarrayitem_gc(arr2, idx2)
 computation
prod     = val1 * val2
acc_new  = acc + prod
 increment counters
idx1_new = idx1 + 1
idx2_new = idx2 + 1
 loop back
jump(acc_new, idx1_new, idx2_new, len1, len2, arr1, arr2)
```

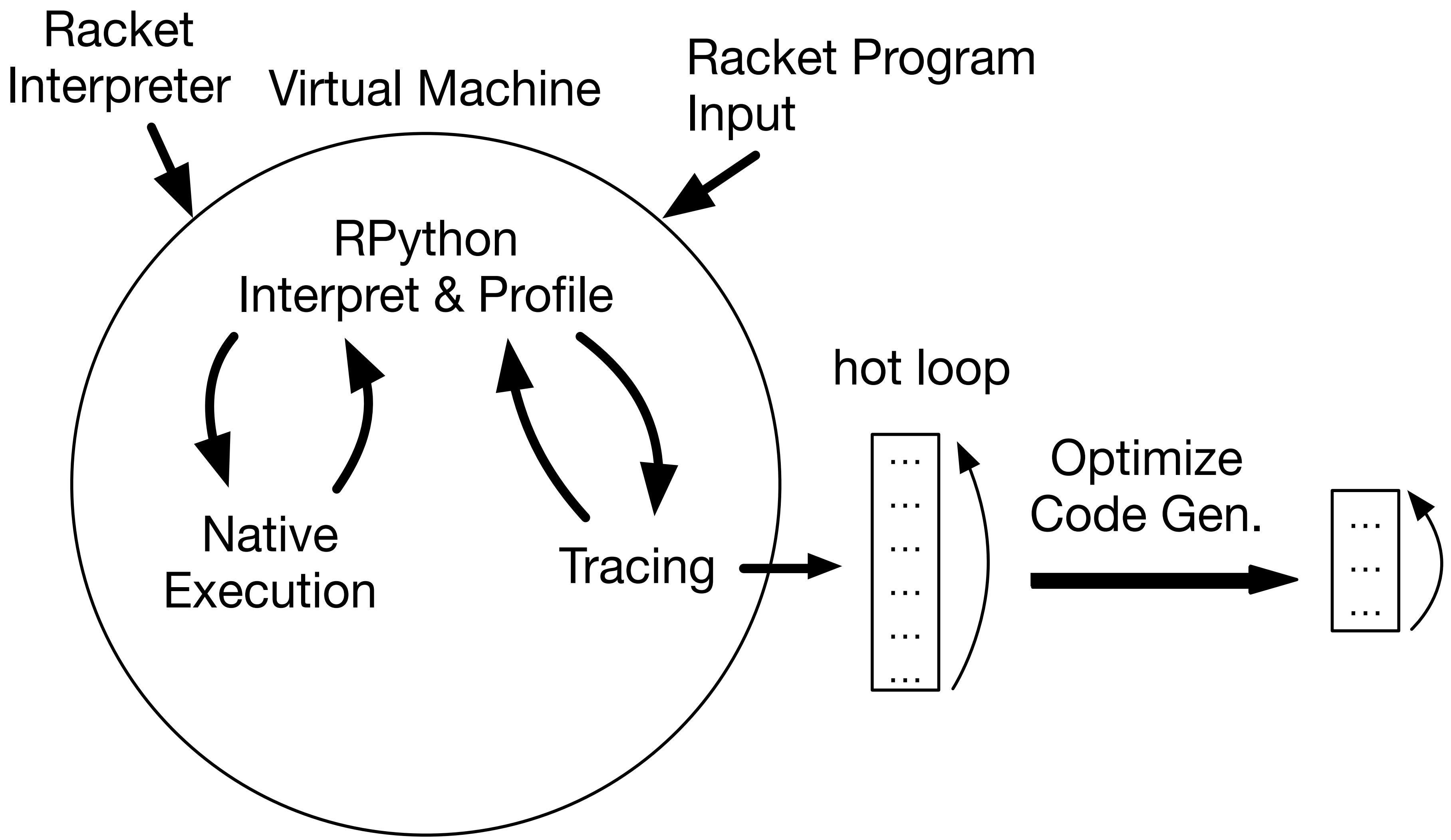
Meta-tracing: the magic part

We didn't write a JIT or an optimizer!

We didn't write a JIT or an optimizer!

RPython creates a JIT from an interpreter

Meta-tracing JIT



CEK Machine

$$e ::= x \mid \lambda x.\, e \mid e\ e$$

$$\kappa ::= [] \mid \mathsf{arg}(e, \rho)::\kappa \mid \mathsf{fun}(v, \rho)::\kappa$$

$$\begin{aligned}
\langle x, \rho, \kappa \rangle &\longmapsto \langle \rho(x), \rho, \kappa \rangle \\
\langle (e_1\ e_2), \rho, \kappa \rangle &\longmapsto \langle e_1, \rho, \mathsf{arg}(e_2, \rho)::\kappa \rangle \\
\langle v, \rho, \mathsf{arg}(e, \rho')::\kappa \rangle &\longmapsto \langle e, \rho', \mathsf{fun}(v, \rho)::\kappa \rangle \\
\langle v, \rho, \mathsf{fun}(\lambda x.\, e, \rho')::\kappa \rangle &\longmapsto \langle e, \rho'[x \mapsto v], \kappa \rangle
\end{aligned}$$

CEK Advantages

Fast continuations

Tail recursion

Arbitrary size stack

CEK Advantages

Fast continuations

Tail recursion

Arbitrary size stack

Allocation everywhere

From CEK to JIT

1. Whole-program type inference
2. Translation to C
3. Adding JIT based on hints

Main Interpreter Loop

```
try:
    while True:
        driver.jit_merge_point()
        if isinstance(ast, App):
            prev = ast
        ast, env, cont = ast.interpret(env, cont)
        if ast.should_enter:
            driver.can_enter_jit()
except Done, e:
    return e.values
```

Other hints

Immutable Data

Loop unrolling

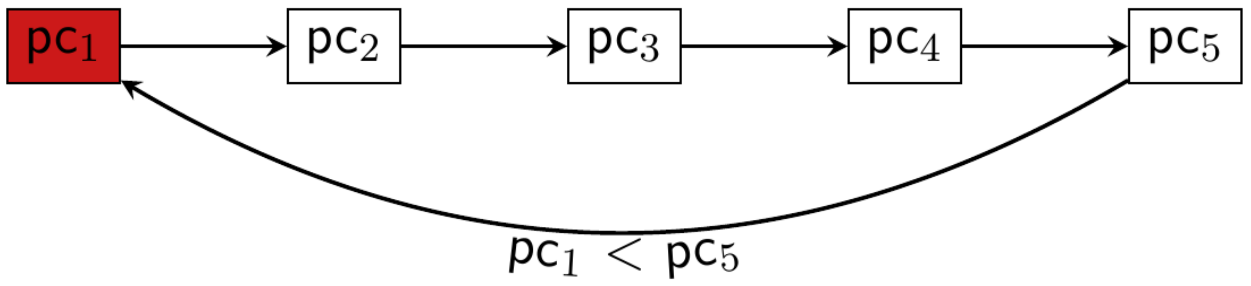
Constant functions

Specialization

A loop by any other name

Detecting loops

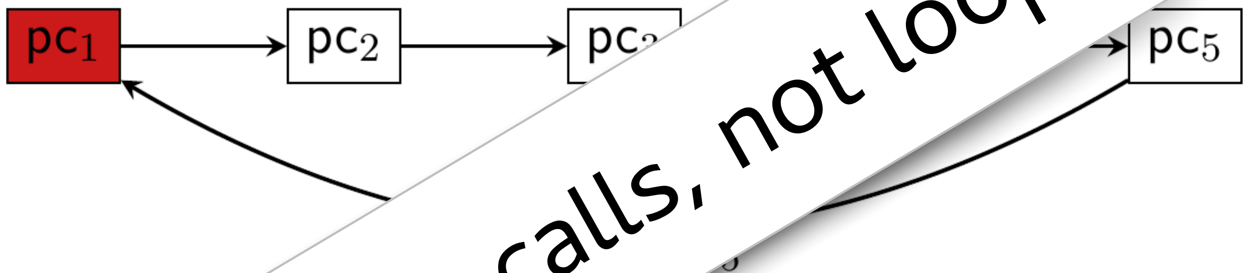
Record back-edges in control flow graph



The default approach

Detecting loops

Record back-edges in contr ow graph



Function calls, not loops

e default approach

Detecting loops

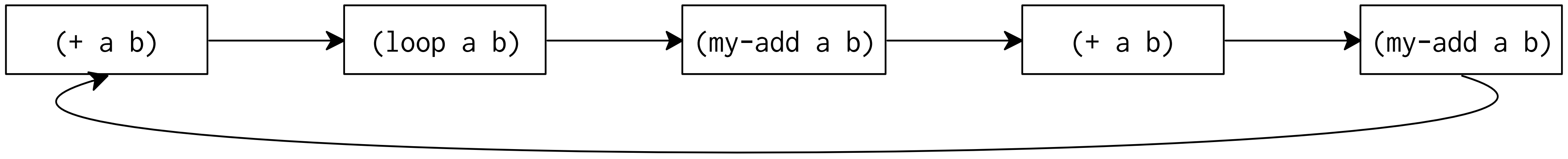
A loop is a repeated AST node

Detecting loops

A loop is a repeated AST node

```
(define (my-add a b) (+ a b))
(define (loop a b)
  (if (= a b) 1
      (loop (my-add a b)
            (my-add a b))))
```

Trace from hot node back to itself

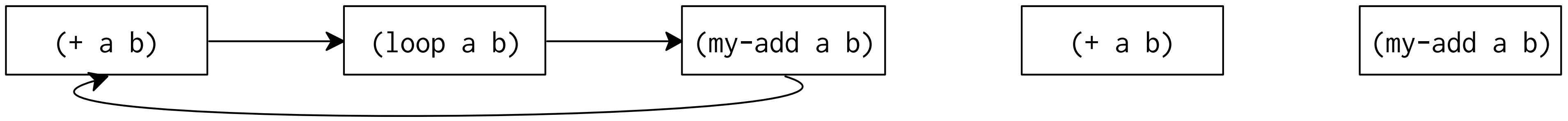


Detecting loops

A loop is a repeated AST node

```
(define (my-add a b) (+ a b))
(define (loop a b)
  (if (= a b) 1
      (loop (my-add a b)
            (my-add a b))))
```

Trace from hot node back to itself



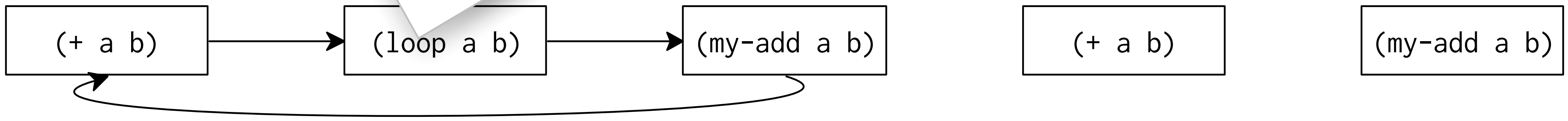
Detecting loops

A loop is a repeated AS node

```
(define (my-add a b)          )
(define (loop a b
  (if (= a b)
      (l          a b)
              add a b))))
```

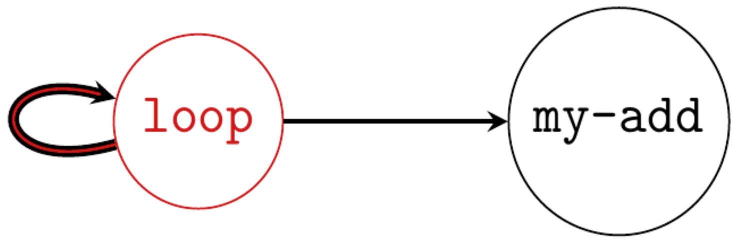
AST nodes lack context

Trace ot node back to itself

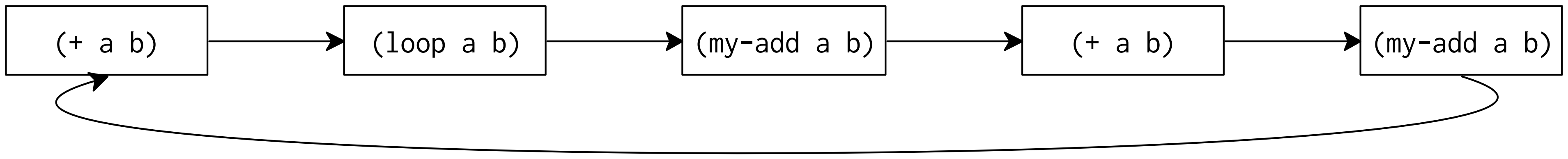


Detecting loops

Construct control flow graph dynamically



Combine with added context



Optimizations

Optimization in the interpreter

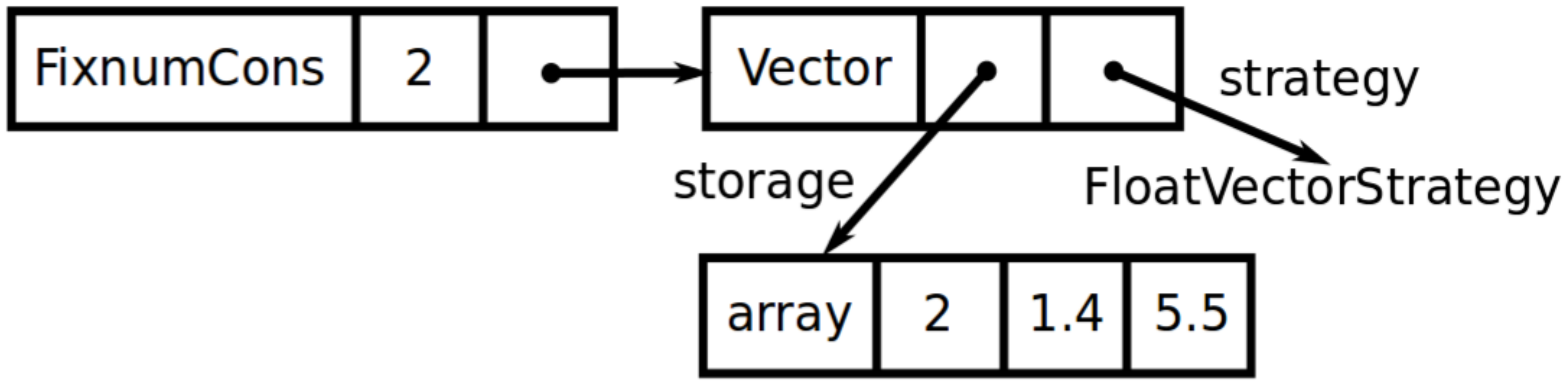
A-normalization

Assignment conversion

Environment optimization

Data structure specialization

Storage Strategies



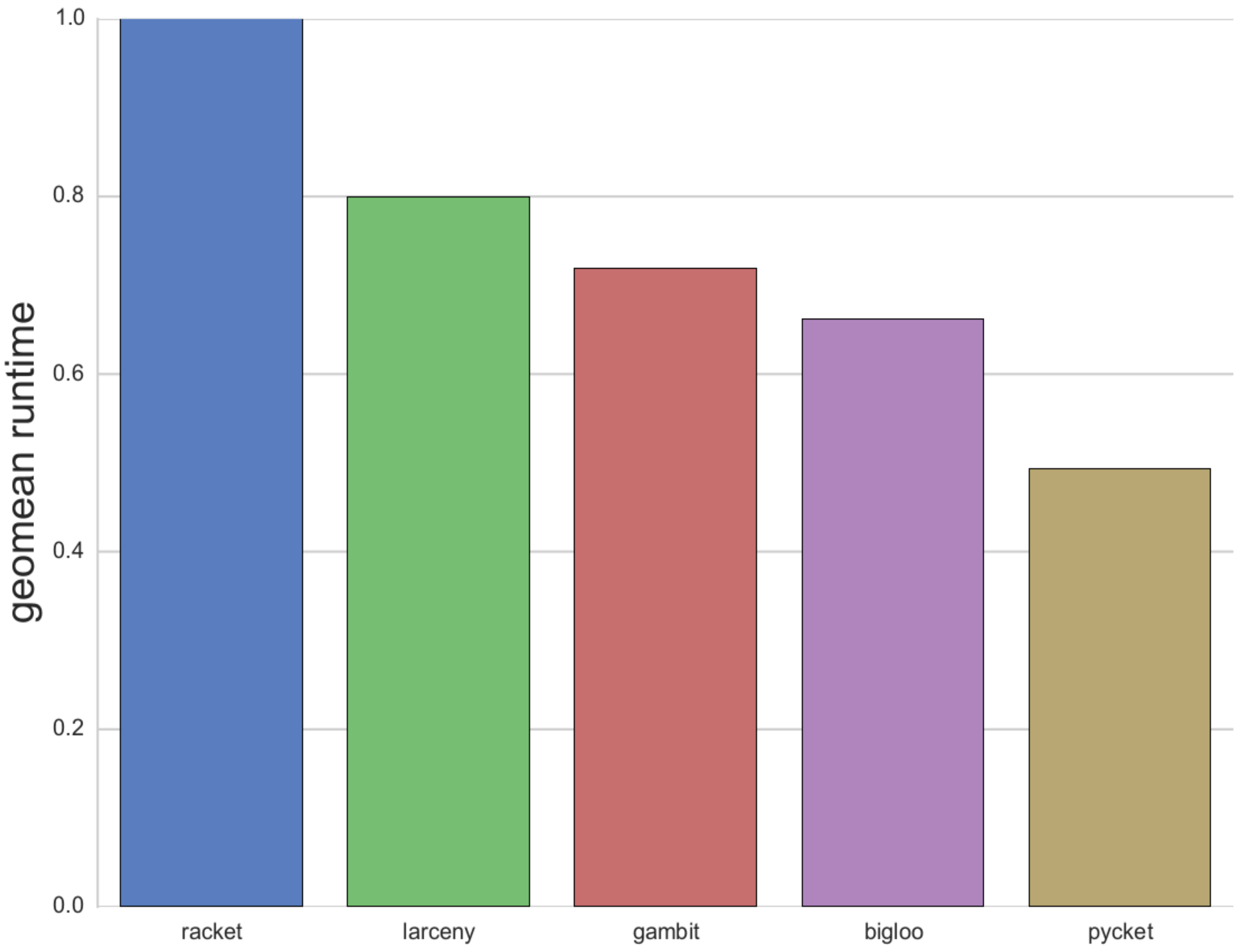
Optimizations we don't do

Closure conversion

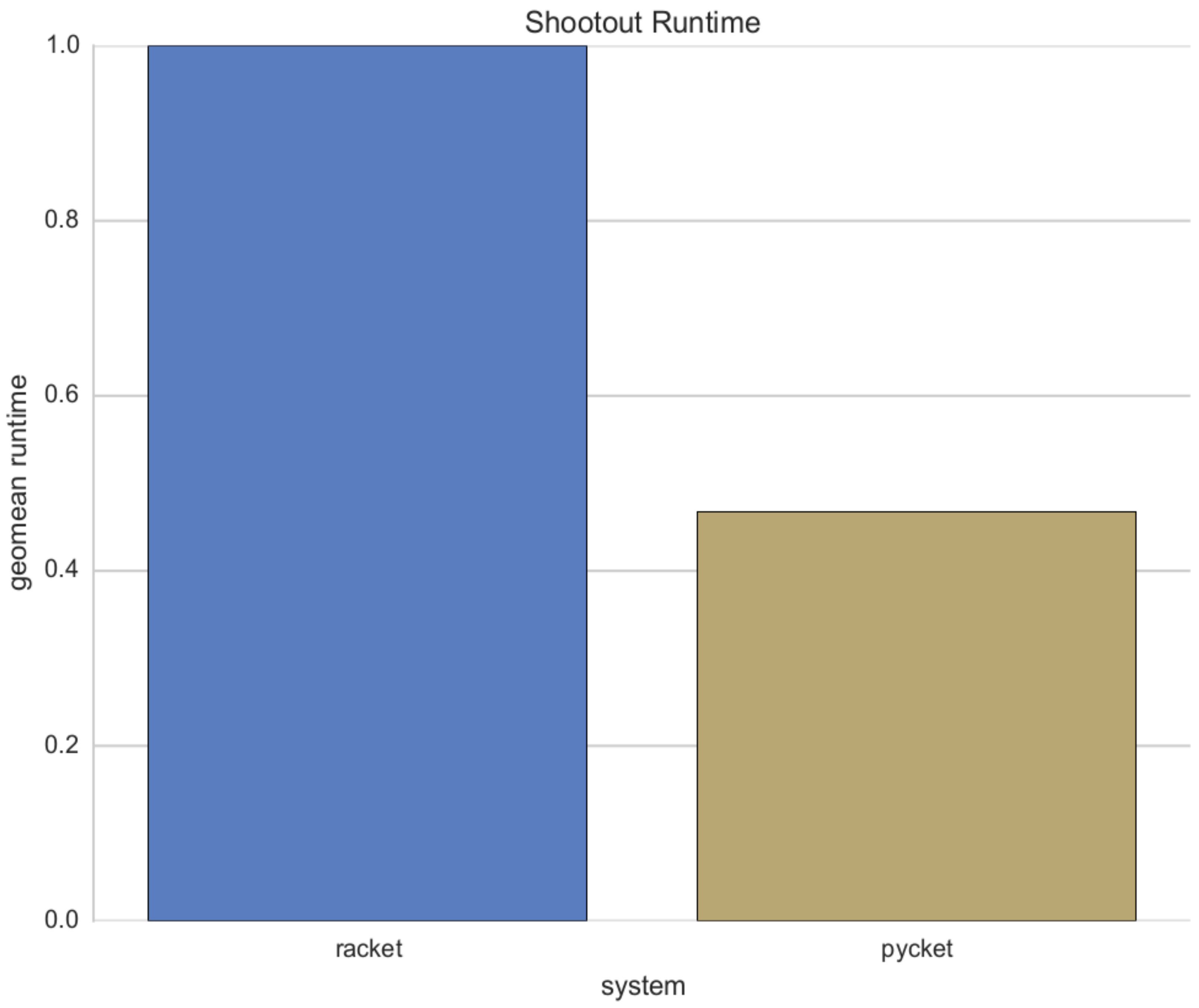
Pointer tagging (64-bit integers!)

How well does it work?

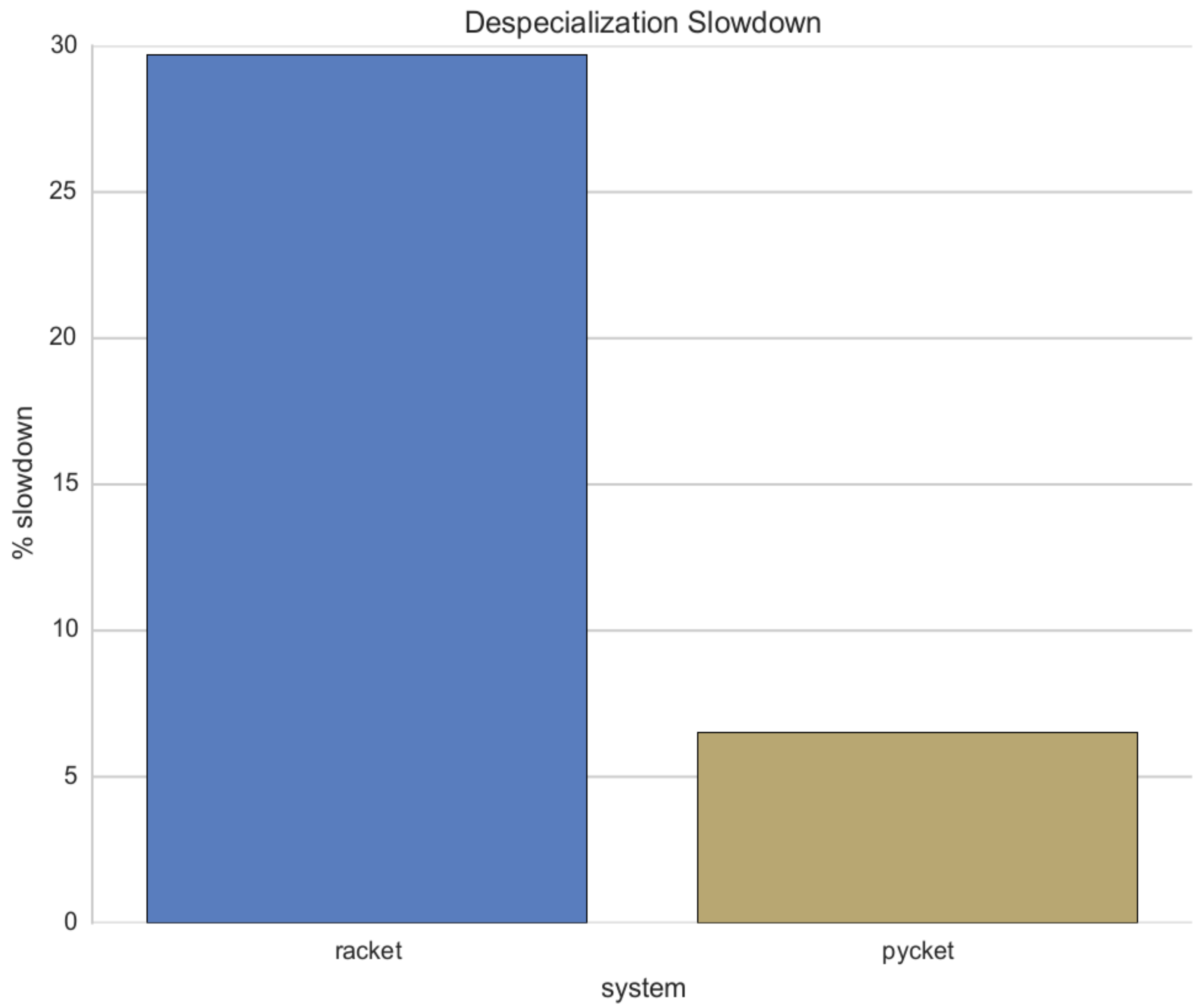
Scheme benchmarks



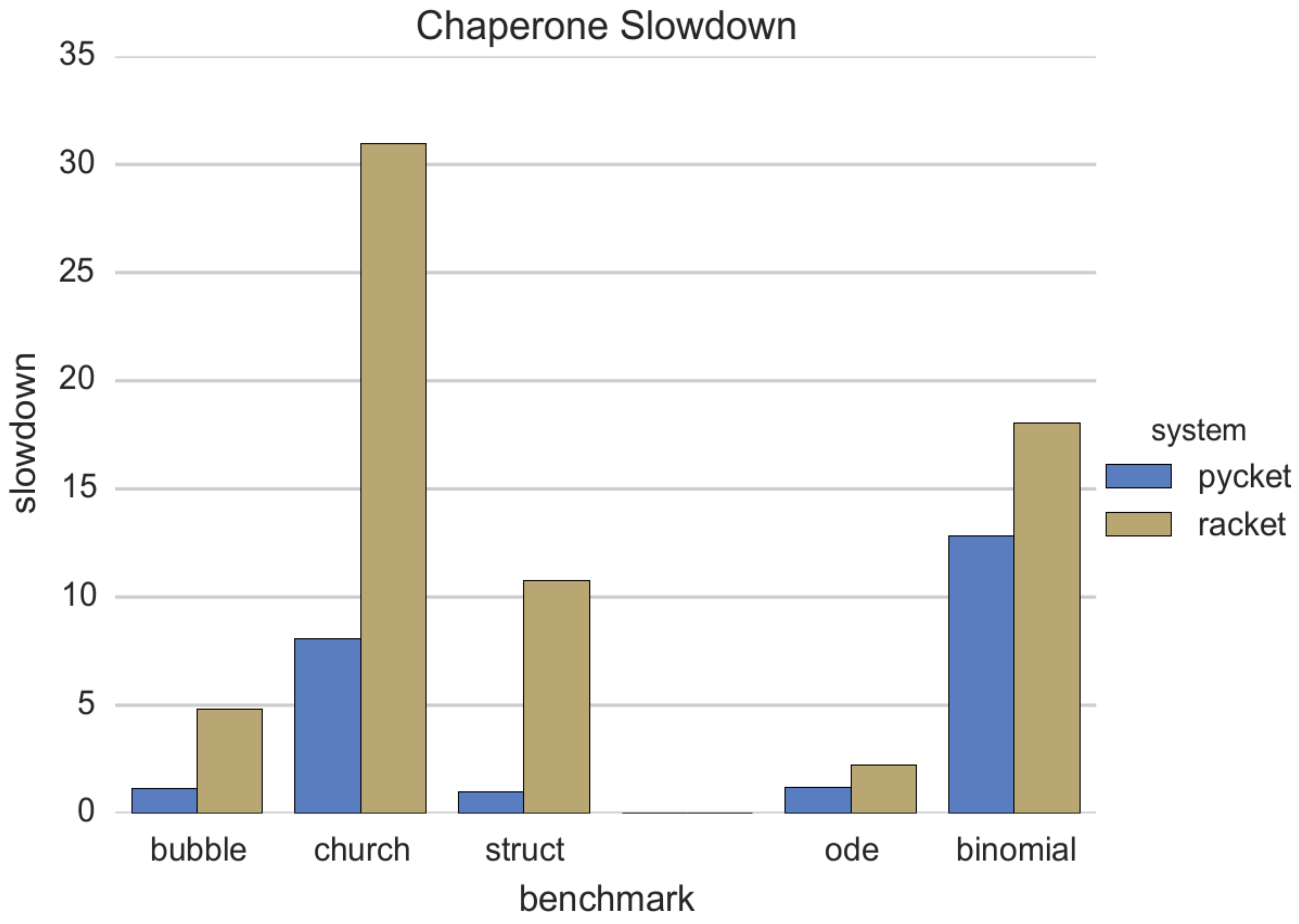
Shootout benchmarks



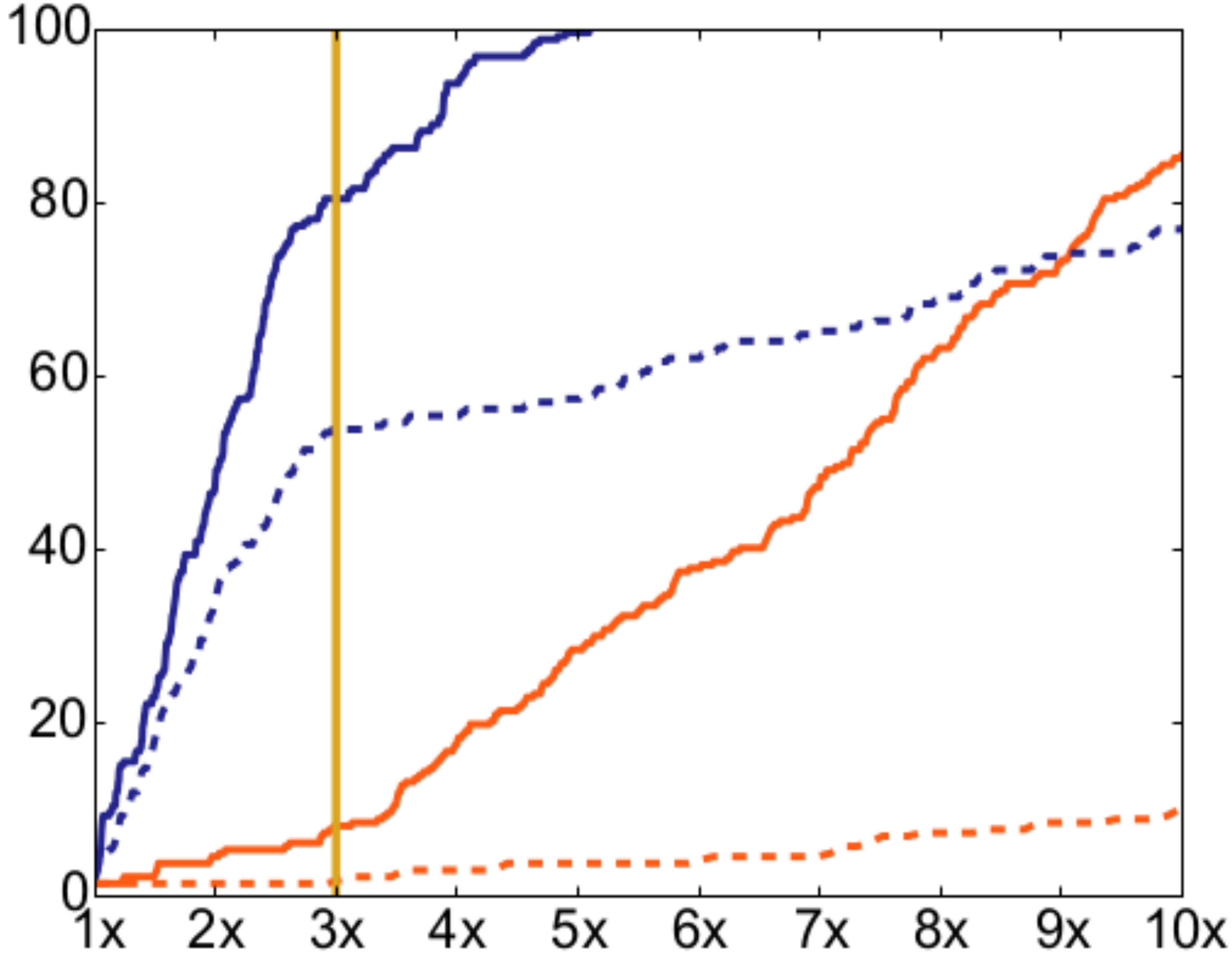
Shootout benchmarks



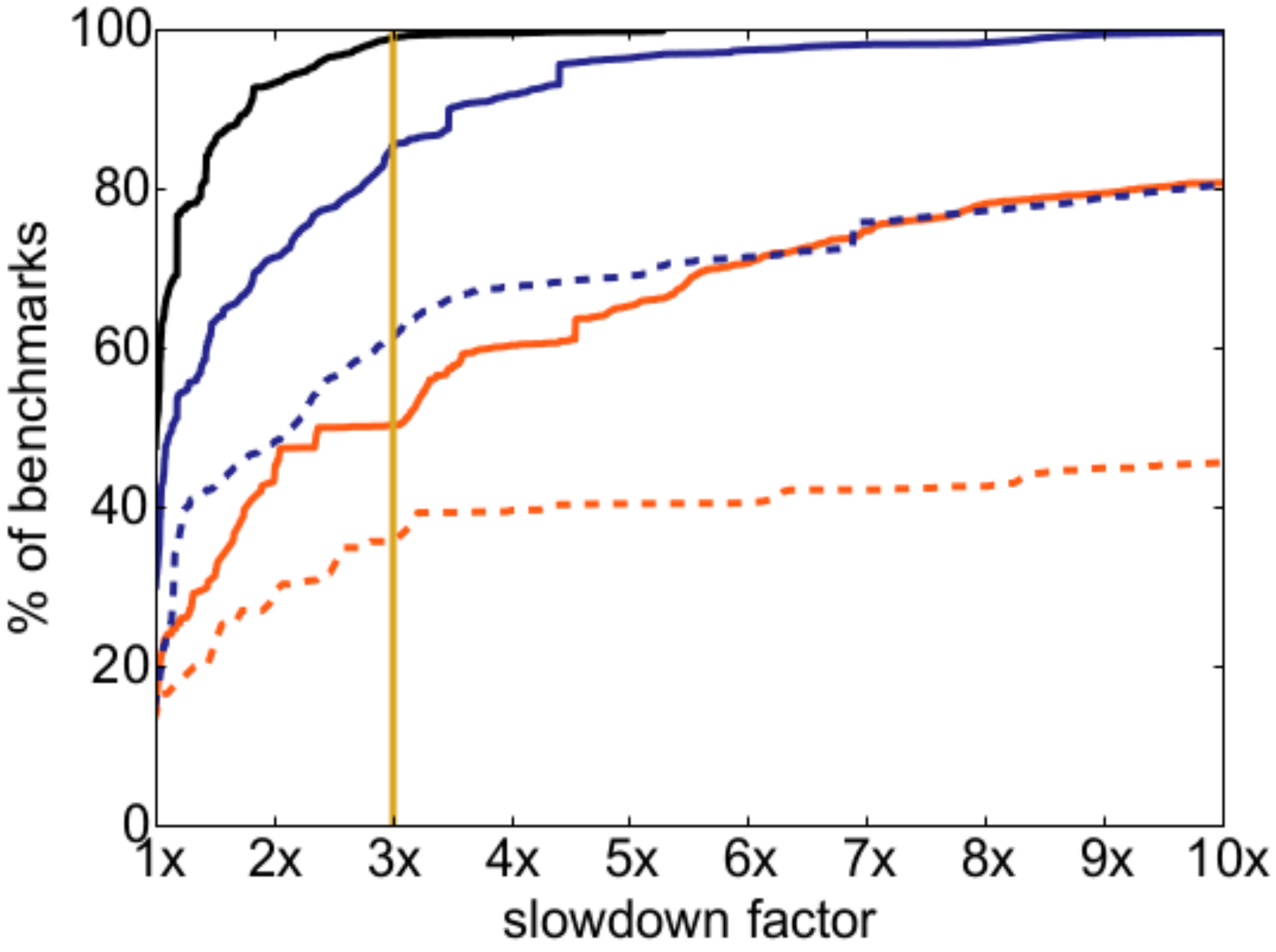
Contract benchmarks



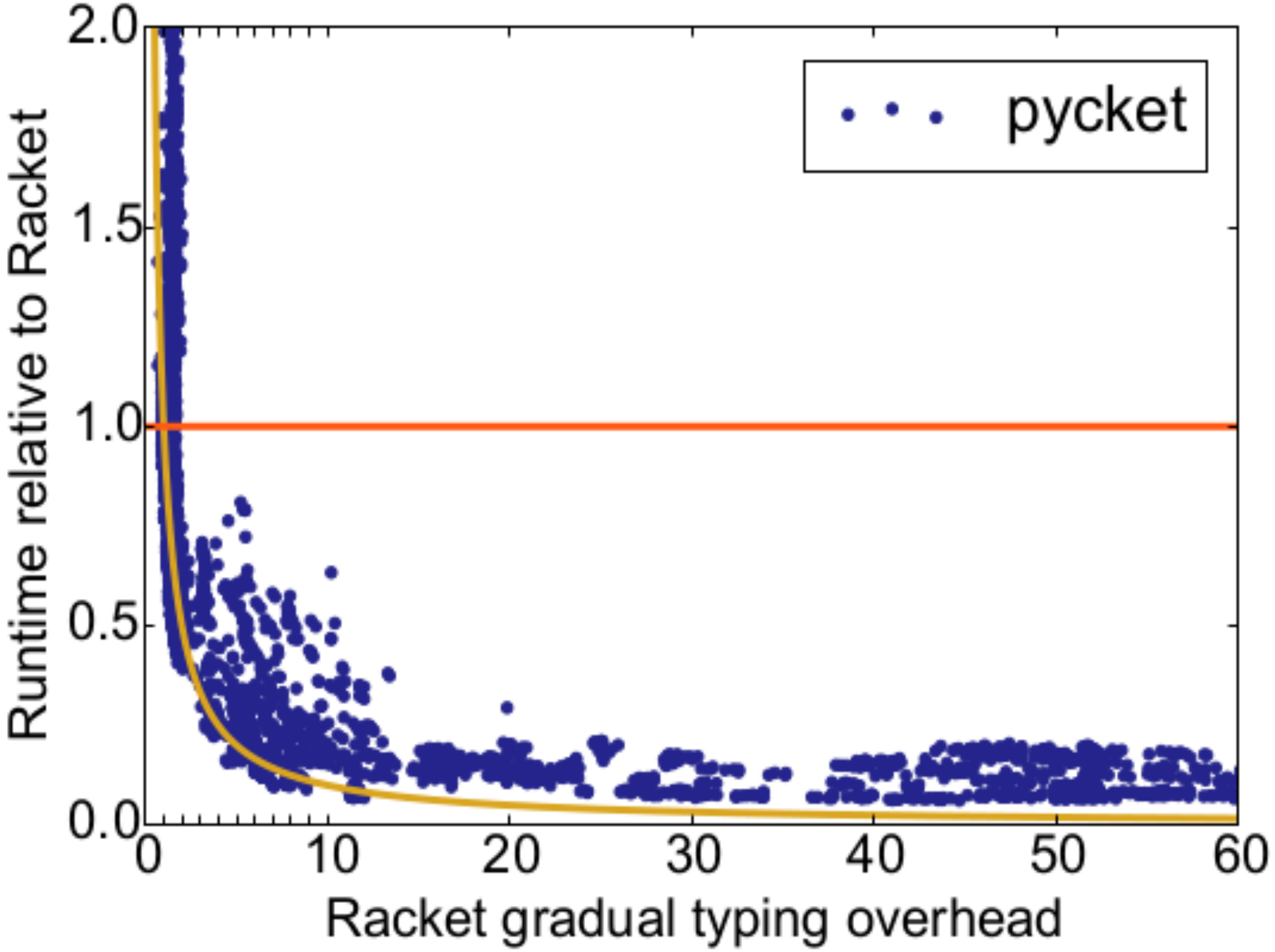
Gradual Typing benchmarks



Gradual Typing benchmarks



Gradual Typing benchmarks



The future of Pycket

What works ...

Basic Scheme

```
lambda, call-with-values, call/cc, complex?
```

Core Racket

```
continuation-marks, make-hash, contract
```

Structures and classes

```
struct, struct-property, object%, mixin
```

Input/output

```
print, read, call-with-input-file
```

Typed Racket

```
#lang typed/racket
```

Contracts

```
chaperone-procedure, make-contract
```

What doesn't work ...

Concurrency and parallelism

`thread, future, place`

FFI

`make-ctype, editor%`

DrRacket

Scribble

Compilation at runtime

`eval, compile`

Networking

`web-server, tcp-connect`

Next steps

AOT + JIT = ♥

Next steps

Expose tracing to programs

Next steps

Accelerate branchy programs

Tracing JIT compilers:

- ★ great for functional languages
- ★ great for generic functions
- ★ great for gradual typing

`github.com/samth/pycket`