Reusing just-in-time compiled code

Jan Vitek
Northeastern University, Boston
Czech Technical University, Prague

Thanks for having me here

I have too many slides

Work done in US, CZ, FR & IN with students and collaborators

How can JavaScript be that fast?

as bad as Python,
as fast as C,
how?

JIT my dear

```
function sorted(x) {
  for (var i=1; i<x.length; i++)
    if (x[i] < x[i-1]) return 0;
  return 1;
}
```

```
function sorted(x) {
  for (var i=1; i<x.length; i++)
    if (x[i] < x[i-1]) return 0;
  return 1;
}
```

1s	**2s**	**80s**
C++ gcc -O2	JavaScript V8	JavaScript V8 -no-opt

measurements obtained in 2019
called with a large vector of floats

```
function sorted(x) {
  for (var i=1; i<x.length; i++)
    if (x[i] < x[i-1]) return 0;
  return 1;
}
```

```
function `[]`(x,i) {
   if (typeof(x) != array) error()
   if (typeof(i) != int)
         i = convert(i, int)
   return get(x, i)
}
function `-`(a,b) {
   if (typeof(a) == float) {
      if (typeof(b) != float)
        b = convert(b, float)
      return subf(a.val, b.val)
   }
   if (typeof(b) == int) {
     …
   …
}
```

```
function `<`(a,b) {
   if (typeof(a) == float) {
      if (typeof(b) != float)
        b = convert(b, float)
      return ltf(a.val, b.val)
   }
   if (typeof(b) == int) {
     …
   …
}
```

Code is highly polymorphic,
but what if we could specialize to float arrays?

How can JavaScript be that fast?

```
function sorted(x) {
  for (var i=1; i<x.length; i++)
    t1 = get(x, i)
    t2 = subi(i,1)
    t3 = get(x,t2)
    t4 = t1.val
    t5 = t3.val
    t6 = ltf(t4,t5)
    if (t6) return 0
  return 1
}
```

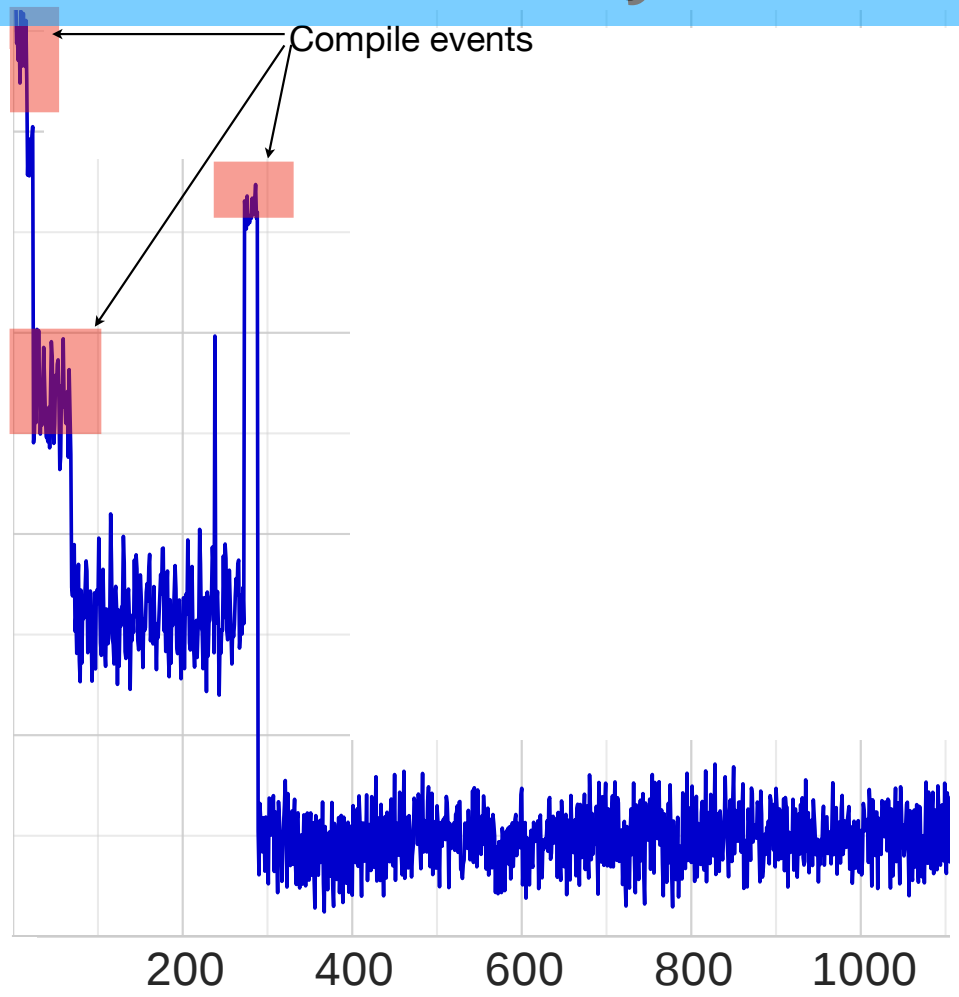
Faster version, but semantically incorrect

How can JavaScript be that fast?

```
function sorted(x) {
 DeoptIf(typeof x != floatarray)
 for (var i=1; i<x.length; i++)
    t1 = get(x, i)
    t2 = subi(i,1)
    t3 = get(x,t2)
    t4 = t1.val
    t5 = t3.val
    t6 = ltf(t4,t5)
    if (t6) return 0
  return 1
}
```

correct, but requires on-stack-replacement (OSR)
if argument misspeculation occurs

How can JavaScript be that fast?



Fasta JS benchmark,
V8, Linux, i7-4790

A brief history of just in time compilation

[Mitchell J.G. 1970]
Lazy compilation of uncommon branches (aka TracingJIT)
[Deutsch, Shiffman 1984]
Inline caches speculate on method receiver class
[Chambers, Ungar 1989]
Specialization based on receiver types
[Holzle, Chamber, Ungar 1992]
Dynamic deoptimization of optimized code

[Barrett ea. *Virtual Machine Warmup Blows Hot and Cold.* OOPSLA17]

Speculations

Formally Verified Speculation and Deoptimization in a JIT Compiler

AURÈLE BARRIÈRE, Univ Rennes, Inria, CNRS, IRISA, France
SANDRINE BLAZY, Univ Rennes, Inria, CNRS, IRISA, France
OLIVIER FLÜCKIGER, Northeastern University, USA
DAVID PICHARDIE, Univ Rennes, Inria, CNRS, IRISA, France
JAN VITEK, Northeastern University / Czech Technical University, USA

[POPL21]

Correctness of Speculative Optimizations with Dynamic Deoptimization

OLIVIER FLÜCKIGER, Northeastern University, USA
GABRIEL SCHERER, Northeastern University, USA and INRIA, France
MING-HO YEE, AVIRAL GOEL, and AMAL AHMED, Northeastern University, USA
JAN VITEK, Northeastern University, USA and CVUT, Czech Republic

[POPL19]

```
function f(x, y, z)
version default:
  r1=1
 L1:
  r2=x*y
  if z==7
    r3=x*x
    r4=y*y
    return r1+r3+r4
  else
    return r2
```

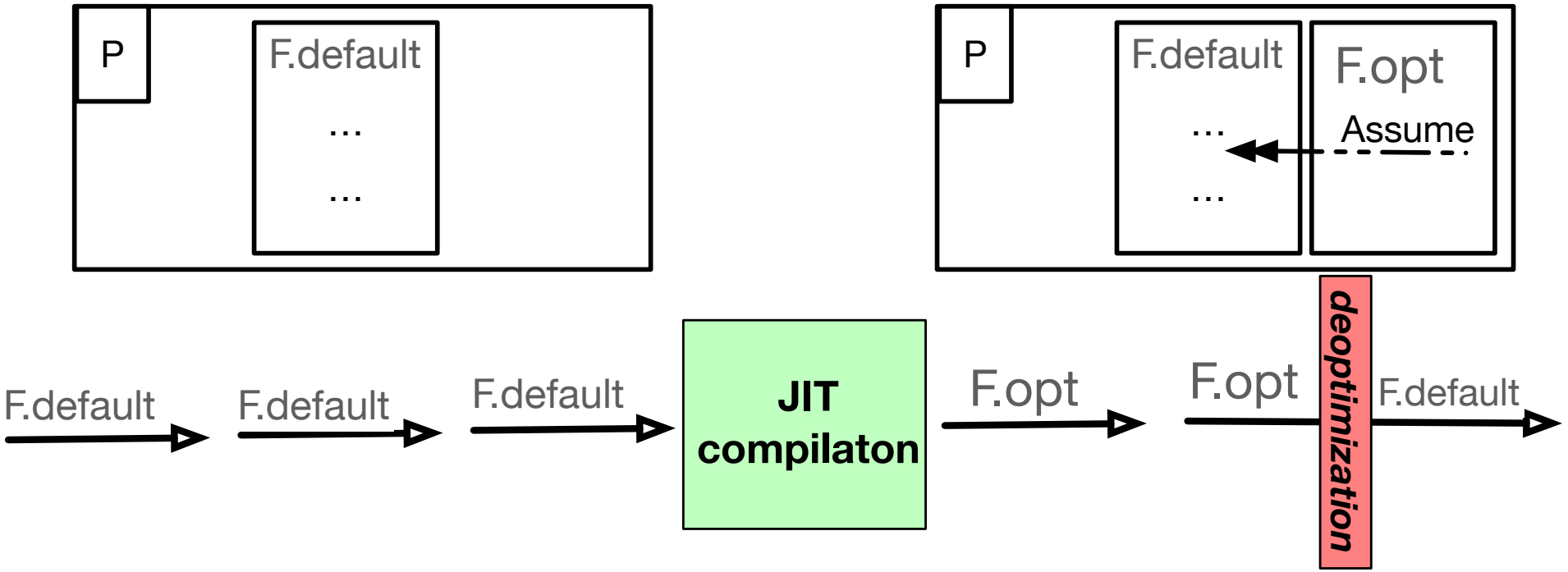
```
function f(x, y, z)
version optimized:
  L4:
   anchor f.default.L1=[x, y, z, r1=1]
   r2=y*y
   assume [x=1, z=75] ⚓ L4
   return 5626+r2
version default:
   …
```

each function can be compiled to multiple ‘versions’

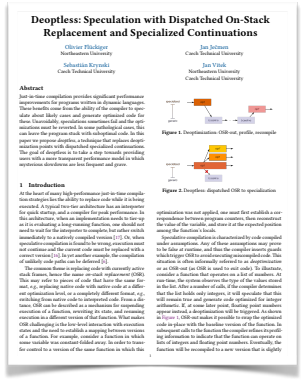
```
function f(x, y, z)
version default:
  r1=1
L1:
  r2=x*y
  if z==7
    r3=x*x
    r4=y*y
    return r1+r3+r4
  else
    return r2
```

```
function f(x, y, z)
version optimized:
 L4:
  anchor f.default.L1=[x, y, z, r1=1]
  r2=y*y
  assume [x=1, z=75] ⚓ L4
  return 5626+r2
version default:
  …
```

deoptimization is reified into:
anchor = how to reconstruct the stack
assume = guard condition



deoptimization jumps into less-optimized function (OSR-out),
we could also jump into more optimized function (OSR-in)


Deoptless: Speculation with Dispatched On-Stack Replacement and Specialized Continuations

[Deoptless: Speculation with Dispatched On-Stack Replacement and Specialized Continuations, **PLDI**'22]

Contexts

Contextual Dispatch for Function Specialization

OLIVIER FLÜCKIGER, Northeastern University, USA
GUIDO CHARI, ASAPP INC, Argentina
MING-HO YEE, Northeastern University, USA
JAN JEČMEN, Czech Technical University, Czechia
JAKOB HAIN, Northeastern University, USA
JAN VITEK, Northeastern University, USA and Czech Technical University, Czechia

[OOPSLA 20]

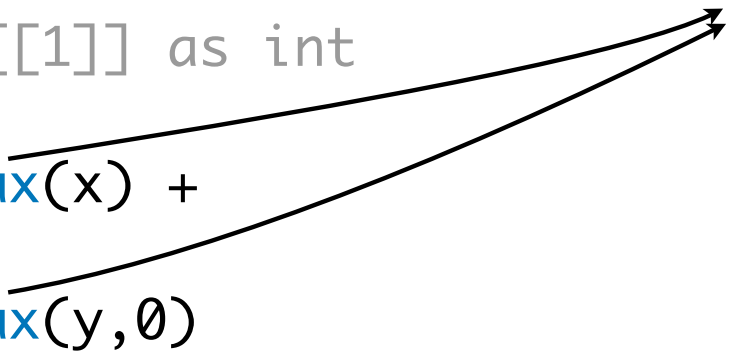
how to reduce count of run-time deopts?
more versions, fewer assumes

```
x = read()

y = x[[1]] as int

z = max(x) +

    max(y,0)
```



```
function max(a, b=a, warn=FALSE)
version default:
  if (warn) print(…)
  return (b < a) ? a : b

version v1 (⊤,⊥,⊥):
  return a

version v2 (int[1],float[1],⊥):
  r1=a[[1]] as float
  r2=b[[1]]
  return ltf(r2,r1) ? a : b
```

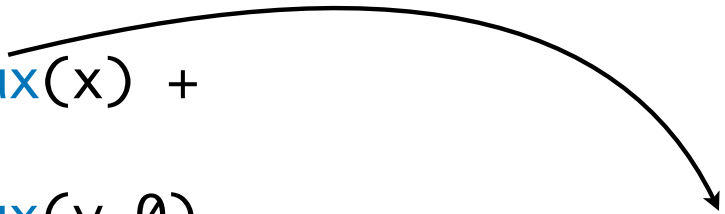
Different uses of same function
are likely to deopt

```
x = read()

y = x[[1]] as int

z = max(x) +

    max(y,0)
```



```
function max(a, b=a, warn=FALSE)
version default:
  if (warn) print(…)
  return (b < a) ? a : b

version v1 (⊤,⊥,⊥):
  return a

version v2 (int[1],float[1],⊥):
  r1=a[[1]] as float
  r2=b[[1]]
  return ltf(r2,r1) ? a : b
```

Contextual dispatch uses best version of a function for each call site

```
x = read()

y = x[[1]] as int

z = max(x) +

    max(y,0)
```

```
function max(a, b=a, warn=FALSE)
version default:
  if (warn) print(…)
  return (b < a) ? a : b

version v1 (⊤,⊥,⊥):
  return a

version v2 (int[1],float[1],⊥):
  r1=a[[1]] as float
  r2=b[[1]]
  return ltf(r2,r1) ? a : b
```

Contexts are constructed at run-time based on static *and* dynamic information

Reuse

Reusing Just-in-Time Compiled Code

ANONYMOUS AUTHOR(S)

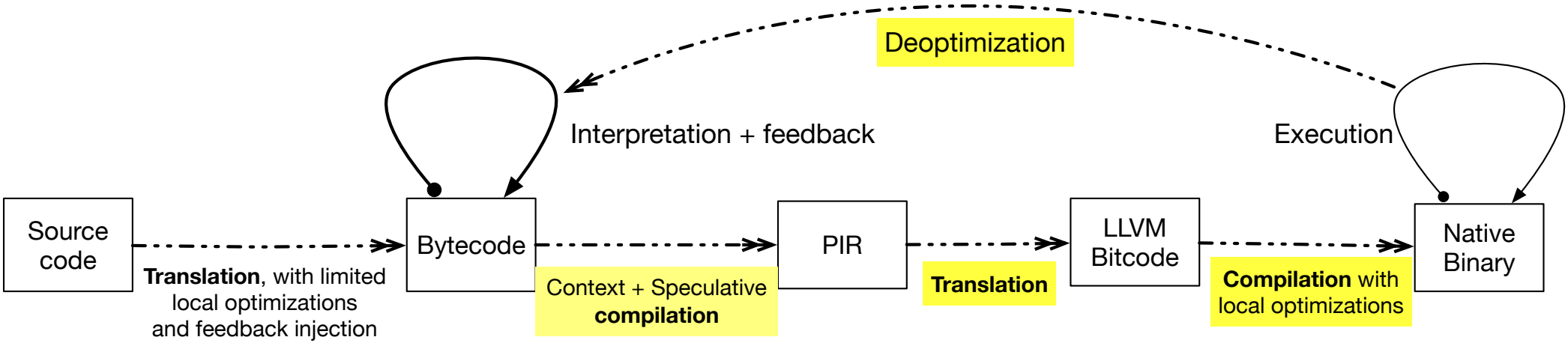
Most code is executed more than once. If not entire programs then, at least, libraries often remain unchanged from one program run to the next. Just-in-time compilers expend considerable effort gathering insights about code they compiled many times before, and often end up generating the same binary over and over again. This paper explores how to reuse compiled code across runs of different programs to cut down on the warm-up costs of dynamic languages. We present an approach that uses *speculative contextual dispatch* to select versions of functions from an *off-line curated code repository*. That repository is a persistent database of previously compiled functions indexed by the context under which they were compiled. Performance can be improved by curating the repository off-line to remove redundant code and optimize dispatch. We assess practicality by extending Ř, a compiler for the R programming language, and evaluating its performance over a set of benchmarks and real-world programs.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; *Dynamic analysis*;

Additional Key Words and Phrases: Specialization, Code reuse.

[XXX 23]

Trouble in Paradise?



JIT warmup times can be high.
JITs have no memory across runs,
learns same truths over and over again.

[R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments, DLS19]

R Melts Brains

An IR for First-Class Environments and Lazy Effectful Arguments

Olivier Flückiger, Northeastern University
Guido Chari, Czech Technical University
Jan Ječmen, Czech Technical University
Ming-Ho Yee, Northeastern University
Jakob Hain, Northeastern University
Jan Vitek, Northeastern / Czech Technical U.

Abstract

The R programming language combines a number of features considered hard to analyze and implement efficiently: dynamic typing, reflection, lazy evaluation, vectorized primitive types, first-class closures, and extensive use of native code. Additionally, variable scopes are reified at runtime as first-class environments. The combination of these features renders most static program analysis techniques impractical, and thus, compiler optimizations based on them ineffective. We present our work on PIR, an intermediate representation with explicit support for first-class environments and effectful lazy evaluation. We describe two dataflow analyses on PIR: the first enables reasoning about variables and their environments, and the second infers where arguments are evaluated. Leveraging their results, we show how to elide environment creation and inline functions.

CCS Concepts • Software and its engineering → Compilers.

Keywords IR, first-class environment, lazy evaluation, R

ACM Reference Format

Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3359619.3359744

1 Introduction

The R language [11] presents interesting challenges for implementers. R is a dynamic imperative language with vectorized operations, copy-on-write of shared data, a call-by-need evaluation strategy, context-sensitive lookup rules, multiple dispatch, and first-class closures. A rich reflective interface and a permissive native interface allow programs to inspect and modify most of R's runtime structures. This paper focuses on the interplay of first-class, mutable environments and lazy evaluation. In particular, we focus on their impact on compiler optimizations.

One might see the presence of `eval` as the biggest obstacle for static reasoning. With `eval`, text can be turned to code and perform arbitrary effects. However, the expressive power of `eval` can be constrained by careful language design. Julia, for instance, has a reflective interface that does not hamper efficient compilation [1]. Even an unconstrained `eval` is bound by what the language allows; for example, most programming languages do not allow code to delete a variable. Not so in R. Consider one of the most straightforward expressions in any language, variable lookup:

```
f <- function(x) x
```

In most languages, it is compiled to a memory or register access. From the point of view of a static analyser, this expression usually leaves the program state intact. Not so in R. Consider a function doubling its argument:

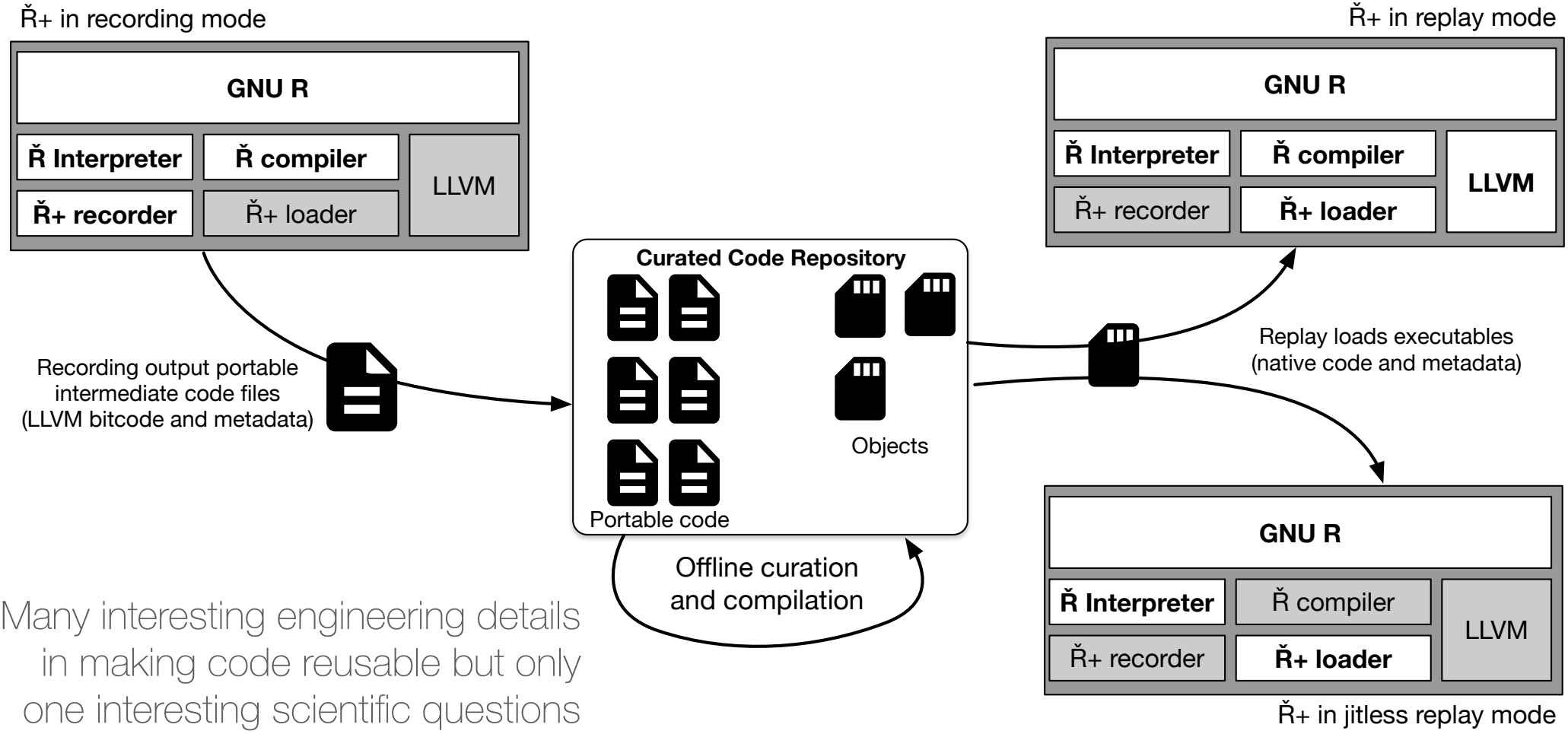
```
g <- function(x) x+x
```

In most languages, a compiler can assume it is equivalent to `2*x` and generate whichever code is most efficient. At the very least, one could expect that both lookups of `x` resolve to the same variable. Not so in R.

Difficulties come from two directions at once. R variables are bound in environments, which are first-class values that can be modified. In addition, arguments are evaluated lazily; whenever an argument is accessed for the first time, it may trigger a side-effecting computation – which could modify any environment. Consequently, to optimize the body of a function, a compiler must reason about effects of the functions that it calls, as well as the effects from evaluating its arguments. In the above example, `+` could walk up the call stack and delete the binding for variable `x`. One could also call `g` with an expression that deletes `x` and causes the second lookup of `x` to fail. While unlikely, a compiler must be ready for it. Considering these examples in combination with `eval`, it is impossible to statically resolve the binding structure of

ACM ISBN 978-1-4503-6996-1/19/10…$15.00
https://doi.org/10.1145/3359619.3359744

Ř+ allows JIT record and replay



Many interesting engineering details in making code reusable but only one interesting scientific questions

When is it safe to reuse a compiled function?

$$compile(f) \rightarrow V_1 \ldots compile(f) \rightarrow V_2$$

Compilations of same source code may yield different binaries.

What is the soundness criteria for reuse?

```
function f(a) {
  print(a+global)
}
```

Let's optimize a simple function
operating over integers and strings
and reading a global variable

```
function f(a)
version default:
 r1 = global
L1:
 r2 = a
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

The default version with local variables runs interpreted

```
function f(a)
version default:
 r1 = global
L1:
 r2 = a
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
global = 42
f(42)
```

```
[ r1  , r2  , r3  , r4  ]
[ int , int , ADD , int ]
```

Interpreter records feedback including type approximations, branches and function pointers

```
function f(a)
version default:
 r1 = global
L1:
 r2 = a
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
global = 42
f(42)
```

```
[ r1  , r2  , r3  , r4  ]
[ int , int , ADD , int ]
```

```
version V1(int):
 r1 = global
L2:
 anchor f.default.L1=[a, r1=r1]
 r2 = a
 assume [r1:int] ⚓ L2
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
function f(a)
version default:
 r1 = global
L1:
 r2 = a
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
global = 42
f(42)
```

```
[ r1   , r2   , r3   , r4   ]
[ int , int , ADD , int ]
```

```
version V2(int):
  r1 = global
L2:
 anchor f.default.L1=[a, r1=r1]
 r2 = a
 assume [r1:int] ⚓ L2
 r3 = &add
 assume [r3 = 0xADD] ⚓ L2
 r4 = iadd(r2, r1)
 print(r4)
```

```
function f(a)
version default:
 r1 = global
L1:
 r2 = a
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
global = 42
f(42)
f("42")
```

```
[ r1   , r2   , r3   , r4   ]
[ int , str , ADD , int ]
```

```
version V3(int|str):
  r1 = global
L2:
 anchor c=[a, r1=r1]
 r2 = a
 assume [r1:int, r2~int] ⚓ L2
 r3 = convert_int(r2)
 r4 = &add
 r5 = r4(a, r1)
 print(r5)
```

```
(int)[int,int,ADD,int]
version V1:
  r1 = global
L2:
 anchor f.default.L1=[a, r1=r1]
 r2 = a
 assume [r1:int] ⚓ L2
 r3 = &add
 r4 = r3(r2, r1)
 print(r4)
```

```
(int)[int,int,ADD,int]
version V2:
  r1 = global
L2:
 anchor f.default.L1=[a, r1=r1]
 r2 = a
 assume [r1:int] ⚓ L2
 r3 = &add
 assume [r3 = 0xADD] ⚓ L2
 r4 = iadd(r2, r1)
 print(r4)
```

```
(int|str)[int,int|str,ADD,int]
 version V3(int|str):
   r1 = global
 L2:
  anchor c=[a, r1=r1]
  r2 = a
  assume [r1:int, r2~int] ⚓ L2
  r3 = convert_int(r2)
  r4 = &add
  r5 = r4(a, r1)
  print(r5)
```

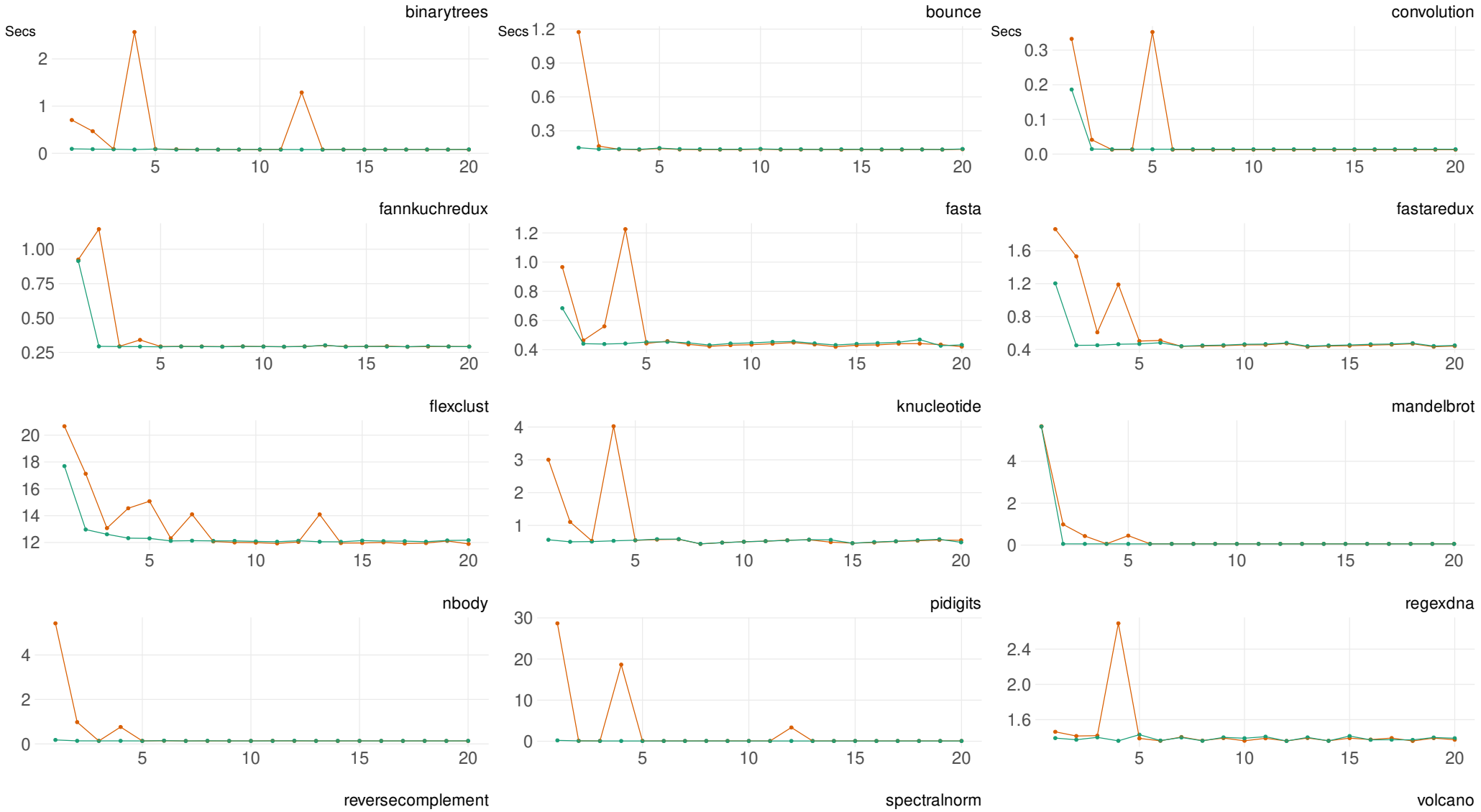
```
(int)[int,int,ADD,int]
 version V1:
  assume [r1:int] ⚓ L2
```

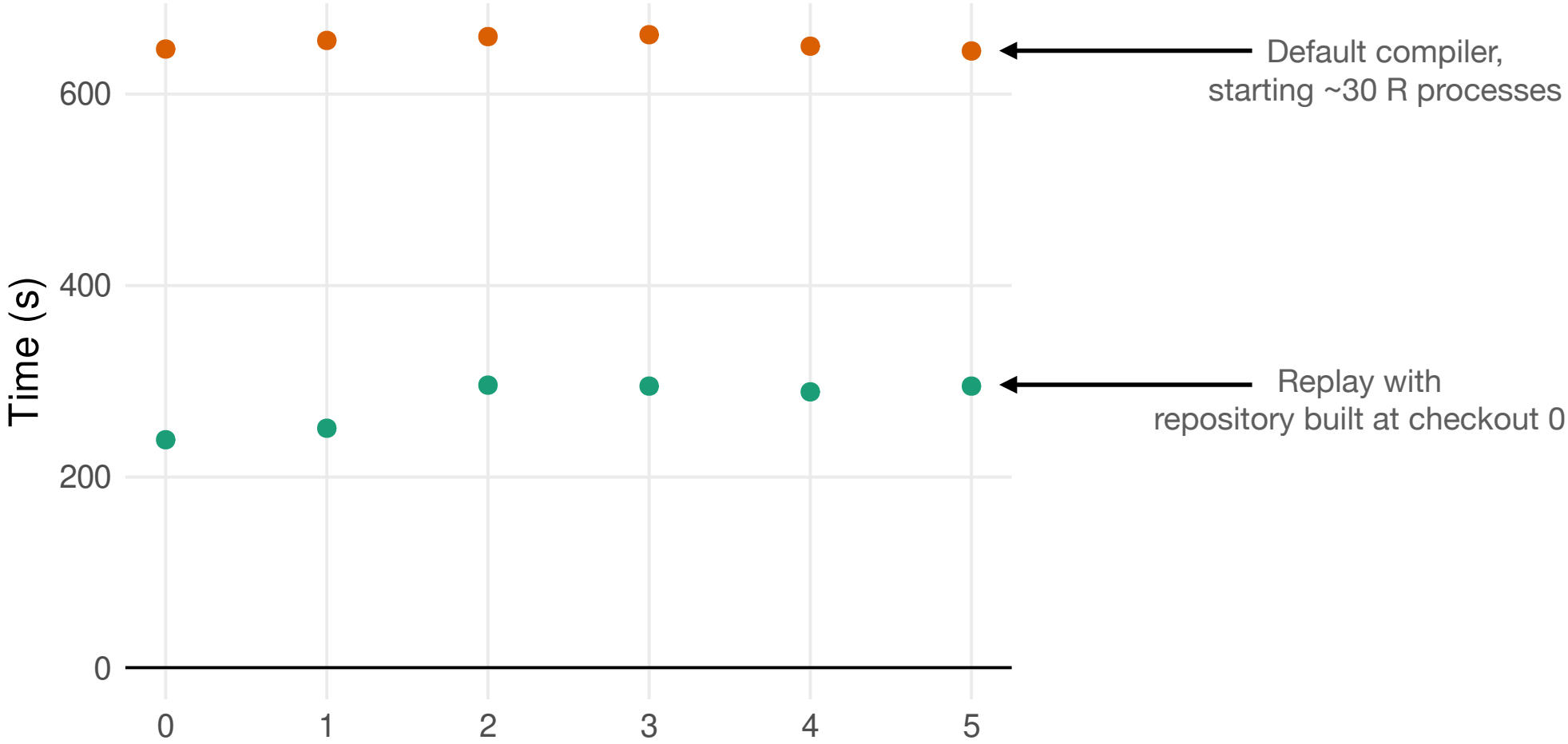
=

```
(int)[int,int,ADD,int]
 version V2:
  assume [r1:int] ⚓ L2
  assume [r3 = 0xADD] ⚓ L2
```

<

```
(int|str)[int,int|str,ADD,int]
 version V3(int|str):
  assume [r1:int, r2~int] ⚓ L2
```


binarytrees
bounce
convolution
fannkuchredux
fasta
fastaredux
flexclust
knucleotide
mandelbrot
nbody
pidigits
regexdna
reversecomplement
spectralnorm
volcano
Secs



Running test suite of RMarkdown,
6 GitHhub checkouts over 2 years

Conclusions

- JITs speculate for performance
- Reusing JITed code requires reifying compiler assumptions
- Warmup greatly reduced with a JITed code repository
- Opens many research directions