

Reusing just-in-time compiled code



Jan Vitek

Northeastern University, Boston
Czech Technical University, Prague
Charles University, Prague

To appear OOPSLA 2023

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

How can JavaScript be that fast?

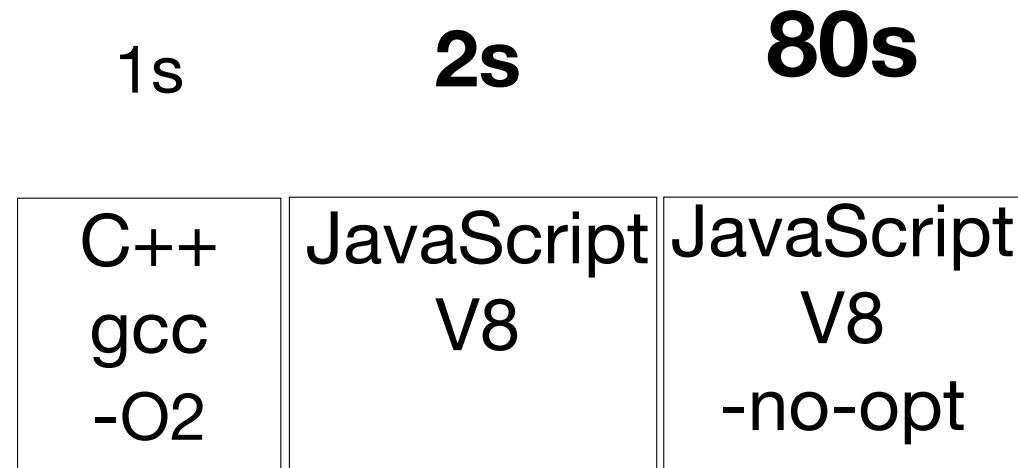


as bad as Python,
as fast as Java,
how?

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;  
}
```



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  
}
```

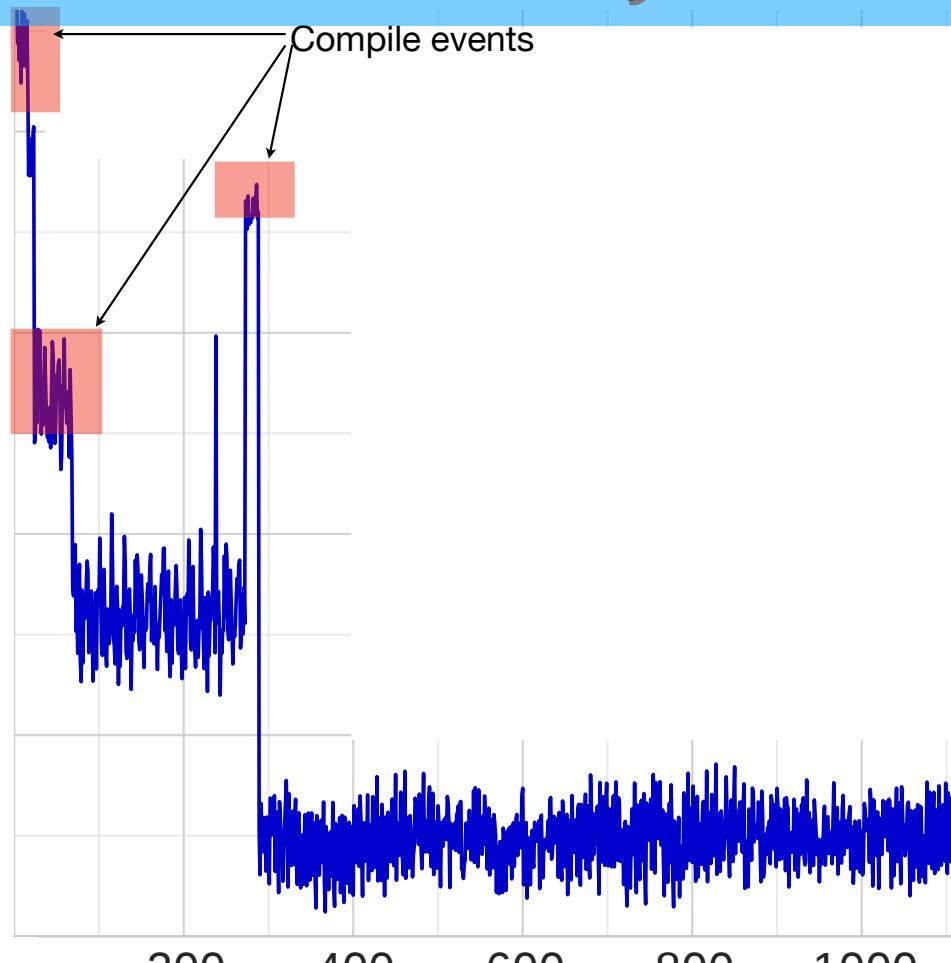
Faster version, but semantically incorrect

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

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

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

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
```

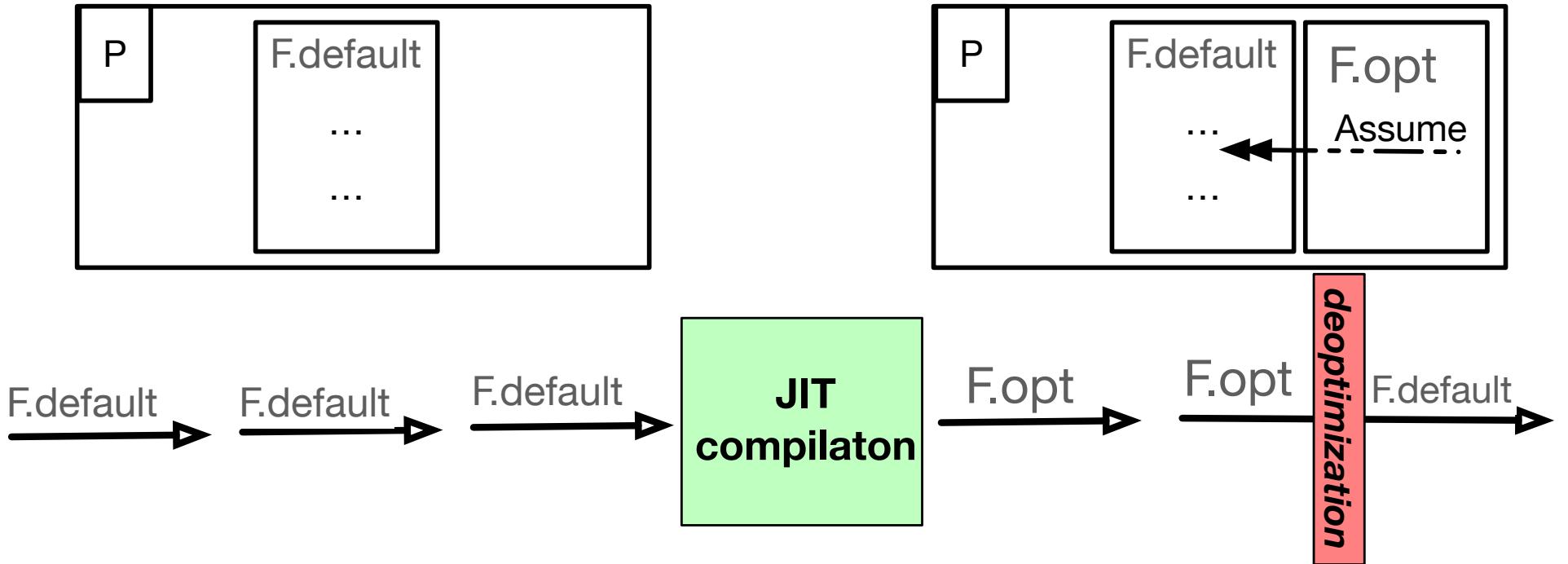
```
function f(x, y, z)
version optimized:
    L4:
        anchor f.default.L1=[x, y, z, r1=1]
        r2=y*y
        assume [x=1, z=75] ⚓ L4
        return 5626+r2
version default:
```

...

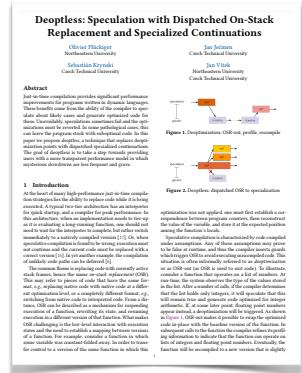
deoptimization is reified into:

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, 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)
```

Different uses of same function
are likely to deopt

```
function max(a, b=a, warn=FALSE)
```

```
version default:
```

```
if (warn) print(...)  
return (b < a) ? a : b
```

```
version v1 ( $\top, \perp, \perp$ ):
```

```
return a
```

```
version v2 (int[1], float[1],  $\perp$ ):
```

```
r1=a[[1]] as float
```

```
r2=b[[1]]
```

```
return ltf(r2,r1) ? a : b
```

```
x = read()
```

```
y = x[[1]] as int
```

```
z = max(x) +
```

```
max(y, 0)
```

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

```
function max(a, b=a, warn=FALSE)  
version default:
```

```
| if (warn) print(...)  
| return (b < a) ? a : b
```

```
version v1 ( $\top, \perp, \perp$ ):
```

```
| return a
```

```
version v2 (int[1], float[1],  $\perp$ ):
```

```
| r1=a[[1]] as float
```

```
| r2=b[[1]]
```

```
| return ltf(r2, r1) ? a : b
```

```
x = read()
```

```
y = x[[1]] as int
```

```
z = max(x) +
```

```
max(y, 0)
```

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

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

version v1 (T, L, L):
| return a

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

Reuse



Reusing JITed Code

* Jan Vitek *

Reusing Just-in-Time Compiled Code

MEETESH KALPESH MEHTA, Indian Institute of Technology Mandi

SEBASTIÁN KRYNSKI, Czech Technical University in Prague

HUGO MUSSO GUALANDI, Czech Technical University in Prague

MANAS THAKUR, Indian Institute of Technology Bombay

JAN VITEK, Northeastern University

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 R, a compiler for the R programming language, and evaluating its performance over a set of benchmarks and real-world programs. We show that our approach can significantly improve slow warmup times related to compilation while preserving the overall peak performance. The evaluation compares the performance profiles of programs with and without our scheme, with a focus on the just-in-time overhead.

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

Additional Key Words and Phrases: Specialization, Code reuse.

[OOPSLA 23]

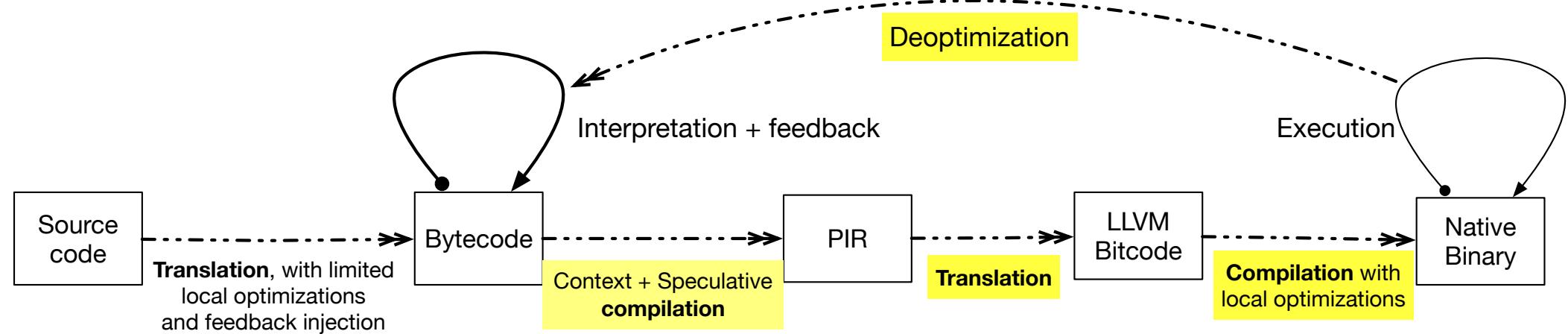
Bertinoro

* September 2023

* act 4

* Reuse

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

Oliver Flückiger
Northwestern University
Guilherme Chiar
Czech Technical University
Jan Jezerník
Czech Technical University
Ming-Ho Iee
Northwestern University
Jakob Hahn
Northwestern University
Jan Vitek
Northwestern / Czech Technical U.

Abstract
The R programming language continues a number of features that have been hard to unify and implement effectively: dynamic typing, reflection, lazy evaluation, vectorized primitive types, first-class classes, and environments. This combination of features makes R difficult to optimize. We propose an intermediate representation (IR) for R that addresses these issues by separating them into two distinct parts: environments and evaluation. The combination of these features makes environments and evaluation very similar to each other, and thus, compiler optimizations based on them interact well. We also propose a new form of environment representation with explicit support for first-class environments and efficient lazy evaluation. We describe two dataplane analyses for PIR: the first infers where environments are used and the second infers where arguments are evaluated. Finally, we show how to use these analyses to guide environment creation and inline functions.

CCS Concepts – Compiler design and its engineering – Compiler optimization; Function evaluation.

Keywords – IR, first-class environment, lazy evaluation, R, ACM Reference Format: Oliver Flückiger, Guilherme Chiar, Jan Jezerník, Ming-Ho Iee, Jakob Hahn, and Jan Vitek. 2019. R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments. In *Proceedings of the 20th ACM SIGPLAN Symposium on Language Design and Implementation* (DLS '19), October 26–28, 2019, Athens, Greece. ACM, New York, NY, USA, 1–15. DOI: <https://doi.org/10.1145/3331184.3331184>

1 Introduction
The R programming language continues a number of features that have been hard to unify and implement effectively: dynamic typing, reflection, lazy evaluation, vectorized primitive types, first-class classes, and environments. This combination of features makes R difficult to optimize. We propose an intermediate representation (IR) for R that addresses these issues by separating them into two distinct parts: environments and evaluation. The combination of these features makes environments and evaluation very similar to each other, and thus, compiler optimizations based on them interact well. We also propose a new form of environment representation with explicit support for first-class environments and efficient lazy evaluation. We describe two dataplane analyses for PIR: the first infers where environments are used and the second infers where arguments are evaluated. Finally, we show how to use these analyses to guide environment creation and inline functions.

2 Function(x)
In most languages, it is compiled to a memory or register access. From the point of view of a static analyzer, it executes the function body and returns the result. Not so in R. Consider a function doubling its argument:

```
double(x) { return 2*x; }
```

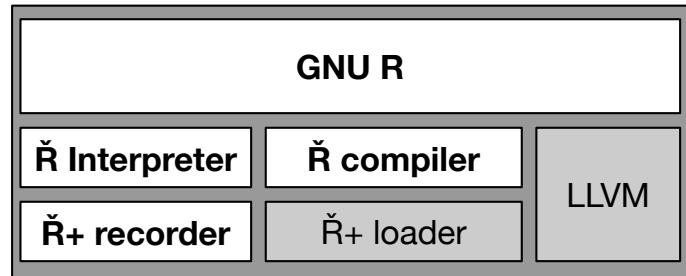
In most languages, a compiler can assume it is equivalent to `double(x) { return x+x; }` and generate whatever code is needed to evaluate the expression at once. R variables are bound in environments, which are first-class values that are passed around. If `x` is a variable in one environment, whenever an argument is accessed for the first time, it may trigger a side-effecting computation – which can interfere with the binding of `x` in another environment. In the body of a function, a compiler must reason about effects of the function body on the environment. For example, consider the following code in R:

```
double(x) { x = 42; return x+x; }
```

When a compiler sees `x = 42`, it cannot know whether `x` is a variable or an expression that defines `x` and causes the second scope of `x` to fail. While statically, a compiler must be ready for both cases, it is often not possible to do so, and thus, it is impossible to statically resolve the binding structure of `x`.

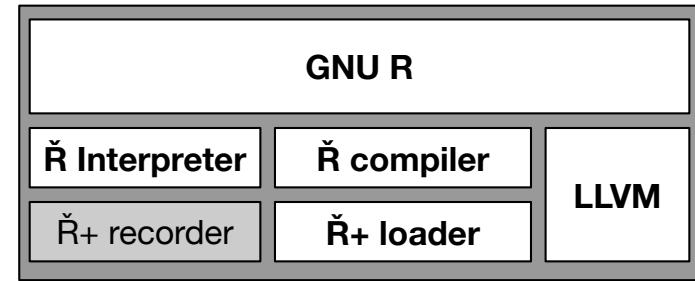
Ŕ+ allows JIT record and replay

Ŕ+ in recording mode

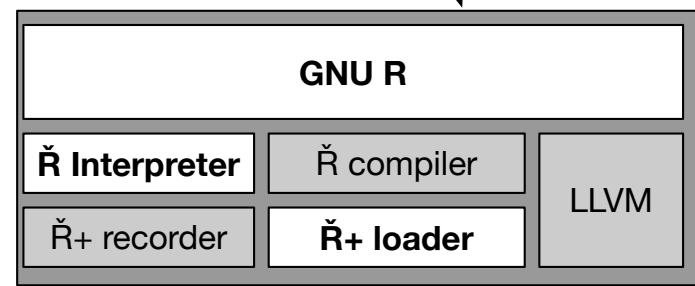


Recording output portable
intermediate code files
(LLVM bitcode and metadata)

Ŕ+ in replay mode



Replay loads executables
(native code and metadata)



Ŕ+ in jitless replay mode

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

When is it safe to reuse a compiled function?

$\text{compile}(f) \rightarrow V_1 \dots \text{compile}(f) \rightarrow V_2$

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]

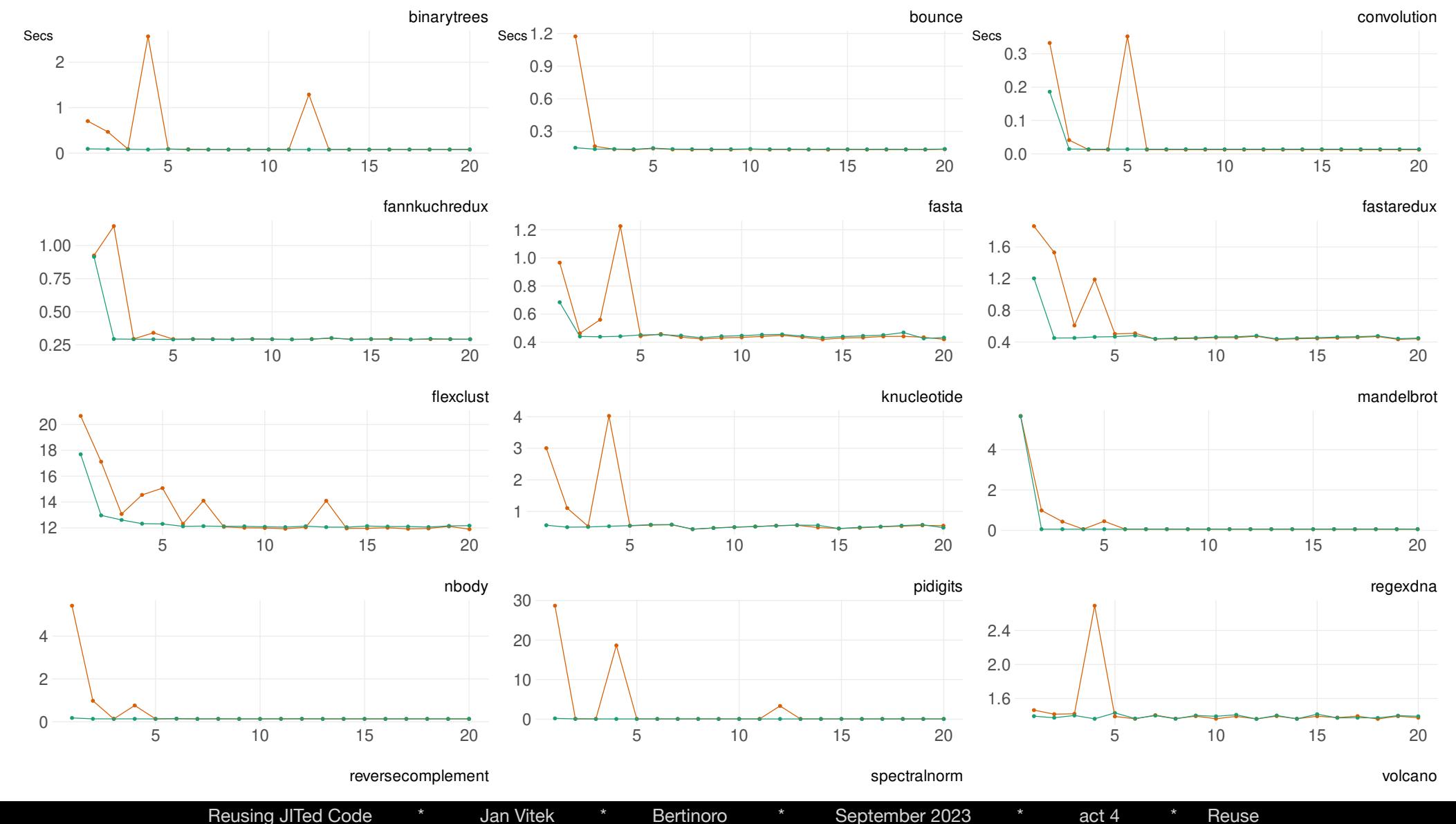
```
version V3(int|str):
r1 = global
L2:
anchor c=[a, r1=r1]
r2 = a
assume [r1:int, r2~int] ⚓ L2
r3 = convert_int(r2)
r4 = &add
r5 = r4(a, r1)
print(r5)
```

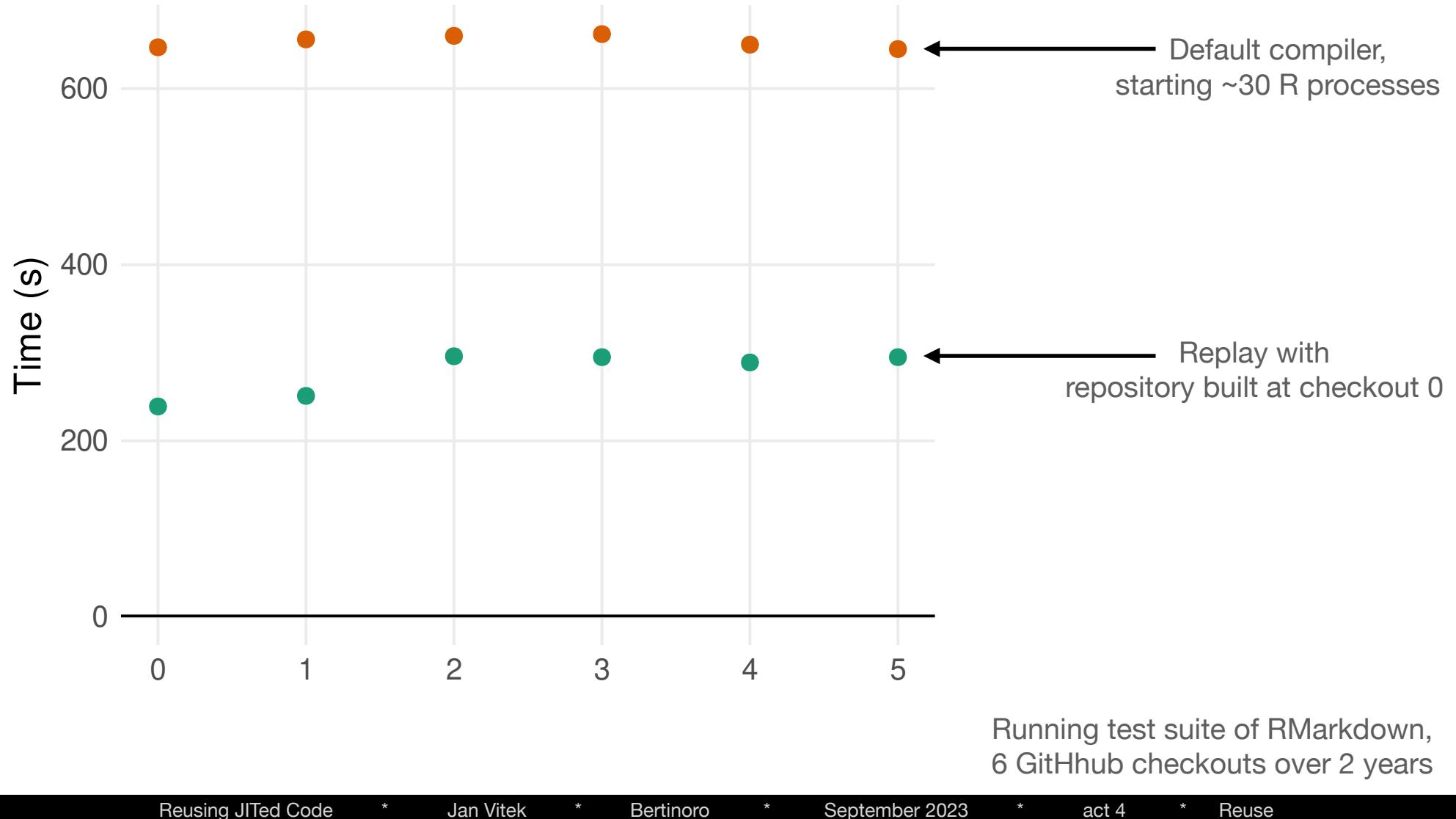
```
(int)[int,int,ADD,int]
version V1:
    r1 = global
L2:
    anchor f.default.L1=[a, r1=r1]
    r2 = a
    assume [r1:int] ⚡ L2
    r3 = &add
    r4 = r3(r2, r1)
    print(r4)
```

```
(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] = (int)[int,int,ADD,int] < (int|str)[int,int|str,ADD,int]  
version V1:  
assume [r1:int] ⚪ L2  
version V2:  
assume [r1:int] ⚪ L2  
assume [r3 = 0xADD] ⚪ L2  
version V3(int|str):  
assume [r1:int, r2~int] ⚪ L2
```





Conclusions



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