Reusing JIT Compiled Code

Meetesh K Mehta

Manas Thakur

Sebastián Krynski

Hugo Musso Gualandi

Jan Vitek

Indian Institute of Technology Mandi

Indian Institute of Technology Bombay

Czech Technical University in Prague

Northeastern University

Reusing Just-in-Time Compiled Code

MEETESH KALPESH MEHTA, IIT Mandi, India
SEBASTIÁN KRYNSKI, Czech Technical University in Prague, Czechia
HUGO MUSSO GUALANDI, Czech Technical University in Prague, Czechia
MANAS THAKUR, IIT Bombay, India
JAN VITEK, Northeastern University, USA

Published at **OOPSLA ‘23**, Cascais, Portugal.

Idea of a JIT

```
make_coffee (...) {
    Get xx% coffee
    Get yy% milk
    ...
    Boil to zz degree
    Serve counter1;
}
```

A **sequence of instructions** to make coffee.

An execution takes **10 minutes**

But we want to serve every order within **2 minutes**.

JIT is the principle of

The perfect coffee, always

Idea of a JIT

```
make_coffee (...) {
    Get xx% coffee
    Get yy% milk
    ...
    Boil to zz degree
    Serve counter1;
}
```

A **sequence of instructions** to make coffee.

An execution takes **10 minutes**

But we want to serve every order within **2 minutes**.

JIT is the principle of

- **Profiling**
    - **Specialization**
        - **Amortization**

The perfect coffee, always

Idea of a JIT

```
make_coffee (...) {
    Get xx% coffee
    Get yy% milk
    ...
    Boil to zz degree
    Serve counter1;
}
```

10 minutes

```
12g coffee, 40 ml milk, 2g sugar
12g coffee, 40 ml milk, 1sp honey
4g coffee, 60 ml milk, 0g sugar
12g coffee, 40 ml milk, 1sp honey
12g coffee, 40 ml milk, 1sp honey
4g coffee, 60 ml milk, 0g sugar
```

The perfect coffee, always

Idea of a JIT

```
make_coffee (...) {
    Get xx% coffee
    Get yy% milk
    ...
    Boil to zz degree
    Serve counter1;
}
```

10 minutes + 10 sec profiling

```
12g coffee, 40 ml milk, 2g sugar
```

```
12g coffee, 40 ml milk, 1sp honey
```

```
4g coffee, 60 ml milk, 0g sugar
```

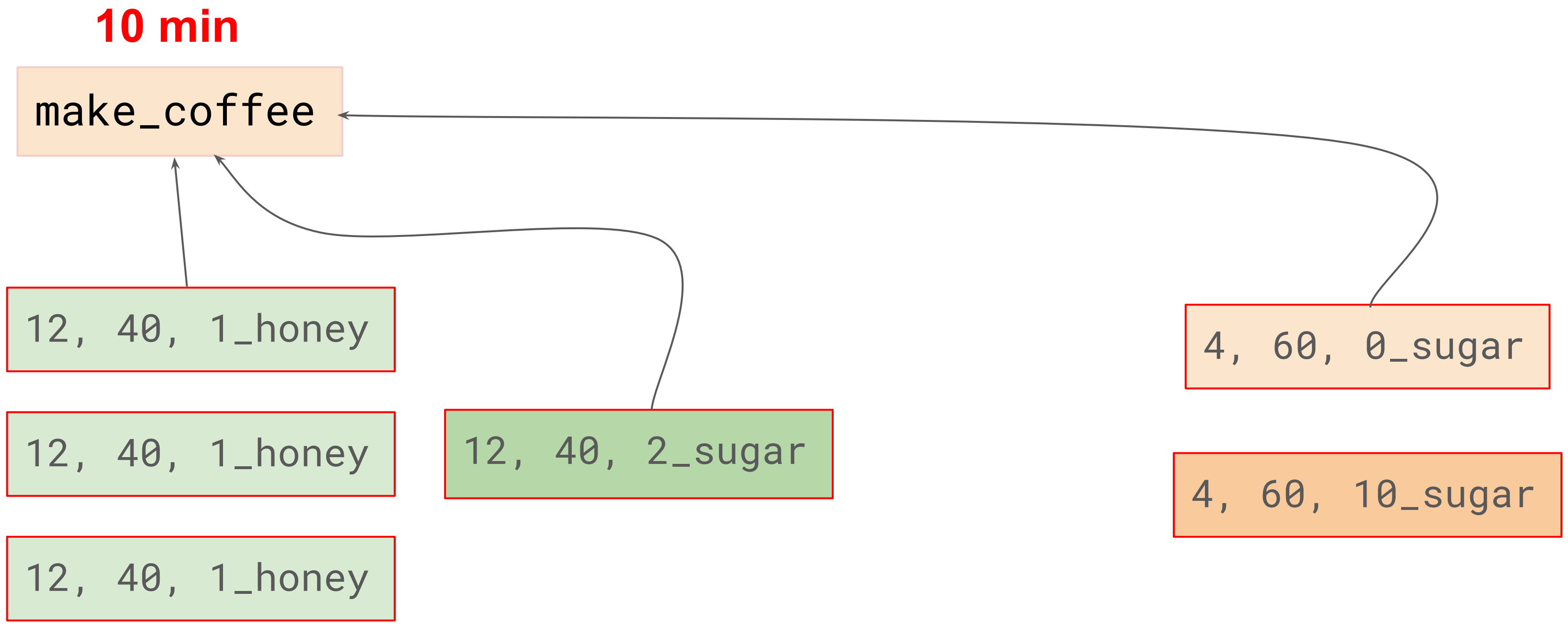
```
12g coffee, 40 ml milk, 1sp honey
```

```
12g coffee, 40 ml milk, 1sp honey
```

```
4g coffee, 60 ml milk, 10g sugar
```

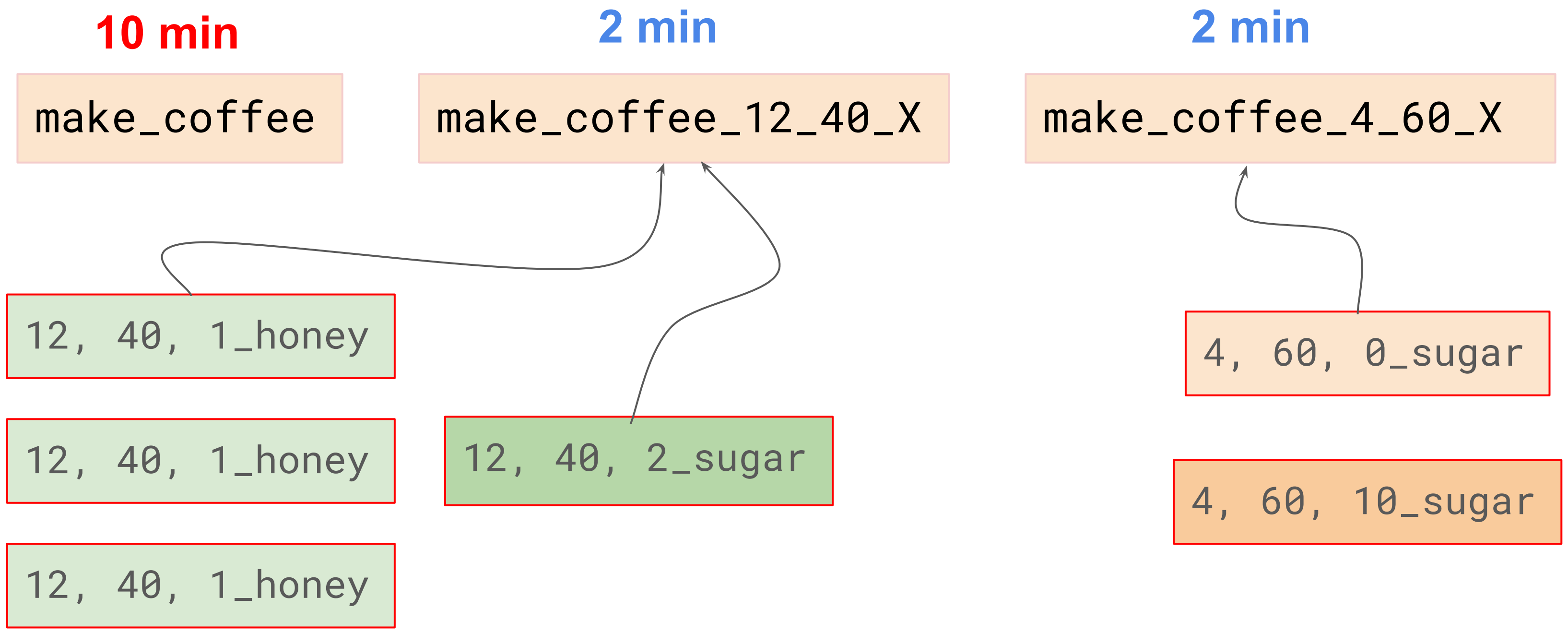
The perfect coffee, always

Idea of a JIT


10 min
make_coffee
12, 40, 1_honey
12, 40, 1_honey
12, 40, 1_honey
12, 40, 2_sugar
4, 60, 0_sugar
4, 60, 10_sugar

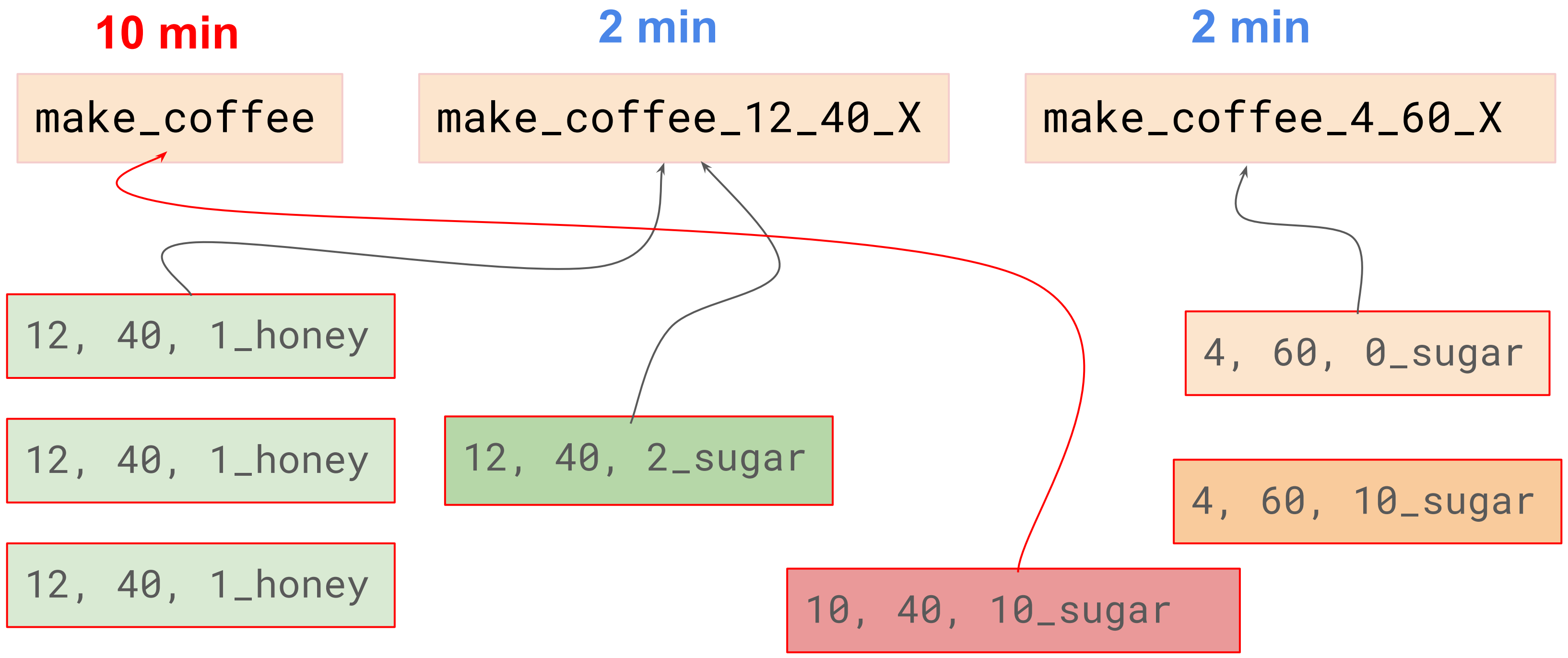
Idea of a JIT

Specialization



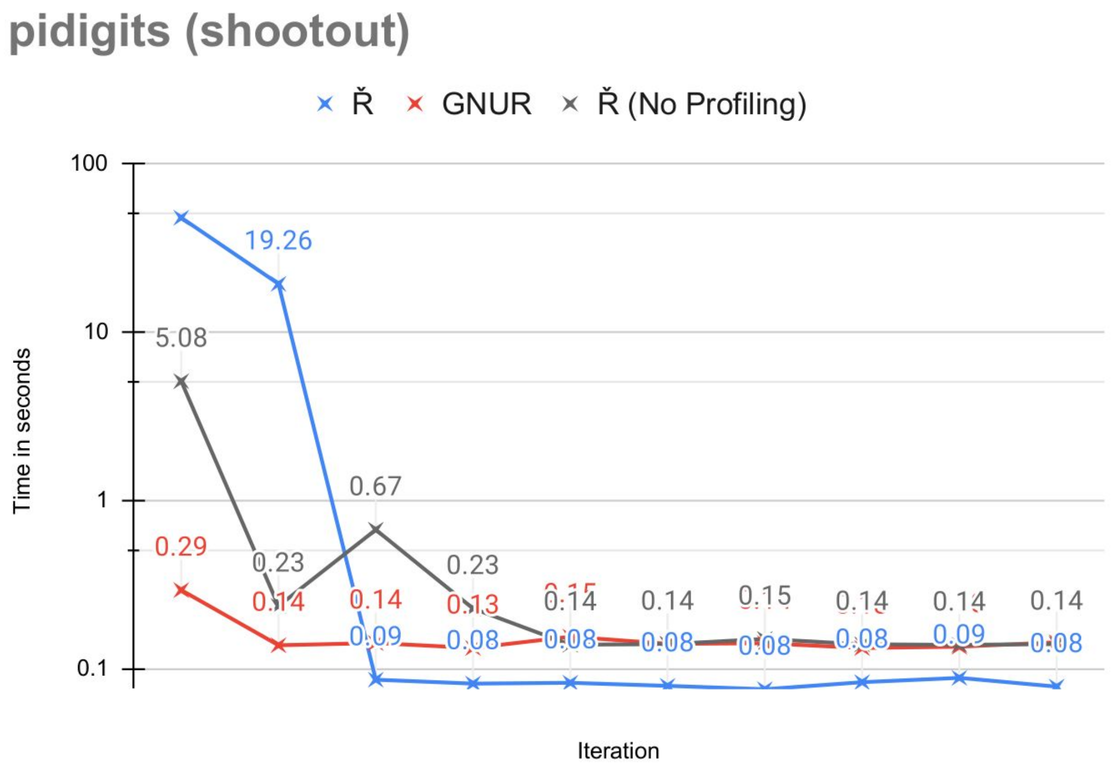
The perfect coffee, always

Idea of a JIT

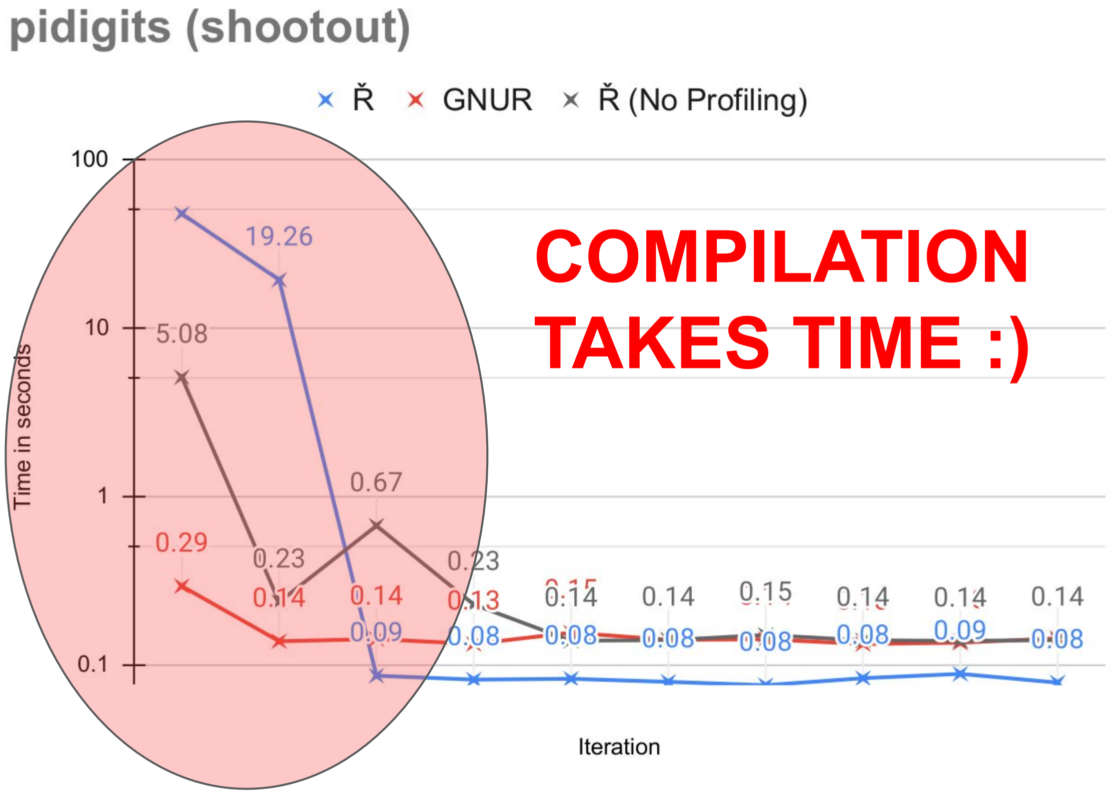


The perfect coffee, always

Motivation: Why reusing compiled code is useful

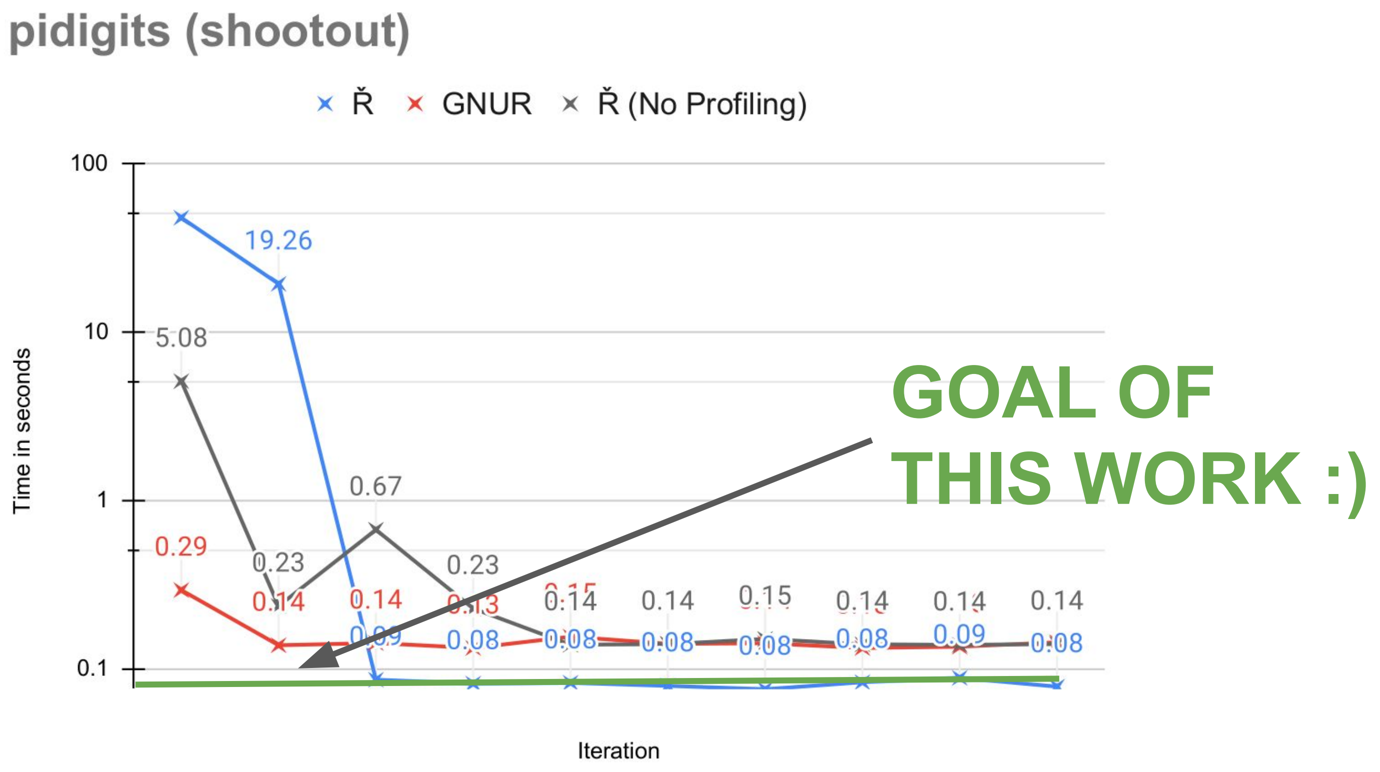


Motivation: Why reusing compiled code is useful



COMPILATION TAKES TIME :)

Motivation: Why reusing compiled code is useful



Motivation: Why reusing JIT compiled code is **hard**

```
foo <- function(x, y, z=TRUE) {
    x;
    if (z) res <- m1(x, y)
    else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

`foo(□,□)`

If we supply just **two arguments**

Are these predicates true?

Motivation

```
foo <- function(x, y, z=TRUE) {
    # x;
    if (z) res <- m1(x, y)
    else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

We can always remove line 2

foo(□,□)

If we supply just **two arguments**

Are these predicates true?

Motivation

```
foo <- function(x, y, z=TRUE) {
    # x;
    if (z) res <- m1(x, y)
    # else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

foo(□,□)

If we supply just **two arguments**

Are these predicates true?

We can always remove line 2

Else condition is never taken

Motivation

```
foo <- function(x, y, z=TRUE) {
    # x;
    if (z) res <- m1(x, y)
    # else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

foo(□,□)

If we supply just **two arguments**

Are these predicates true?

~~We can always remove line 2~~

~~Else condition is never taken~~

Motivation

Lazy Evaluation

```
foo <- function(x, y, z=TRUE) {
    x;
    if (z) res <- m1(x, y)
    else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

```
foo(□,□)
```

```
x <- function()
    assign("z",FALSE,
    sys.frame(-1))

foo(x(),2)
```

Motivation

```r
foo <- function(x, y, z=TRUE) {
    x;
    if (z) res <- m1(x, y)
    else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

```r
foo(□,□)
```

False branch is taken

```r
x <- function()
    assign("z",FALSE,
    sys.frame(-1))

foo(x(),2)
```

Motivation

```
foo <- function(x, y, z=TRUE) {
    # x;
    if (z) res <- m1(x, y)
    # else res <- m2(g, y)
    res <- bar(res, g)
    res
}
```

Call-Site Context

foo(□,□)

We can always remove line 2, if x is non-reflective

Else condition is never taken, if x is non-reflective

Motivation

```
foo’ <- function(x, y, z=TRUE) {
    if (z) res <- m1(x, y)
    res <- bar(res, g)
    res
}
```

Call-Site Context

foo(□,□)

We can always remove line 2, if x is non-reflective

Else condition is never taken, if x is non-reflective

Motivation

```
foo’ <- function(x, y, z=TRUE) {
    if (z) res <- m1(x, y)
    res <- bar(res, g)
    res
}
```

We call this a **version** of foo

Call-Site Context

foo(□,□)

We can always remove line 2, if x is non-reflective

Else condition is never taken, if x is non-reflective

Motivation

```
foo” <- function(x, y, z=TRUE) {
    res <- x + y
    res <- ns::add(res, g)
    res
}
```

```
m1 <- function(a, b) a + b
bar <- ns::add
```

```
foo(10, 20)
```

```
g <- matrix(...)
```

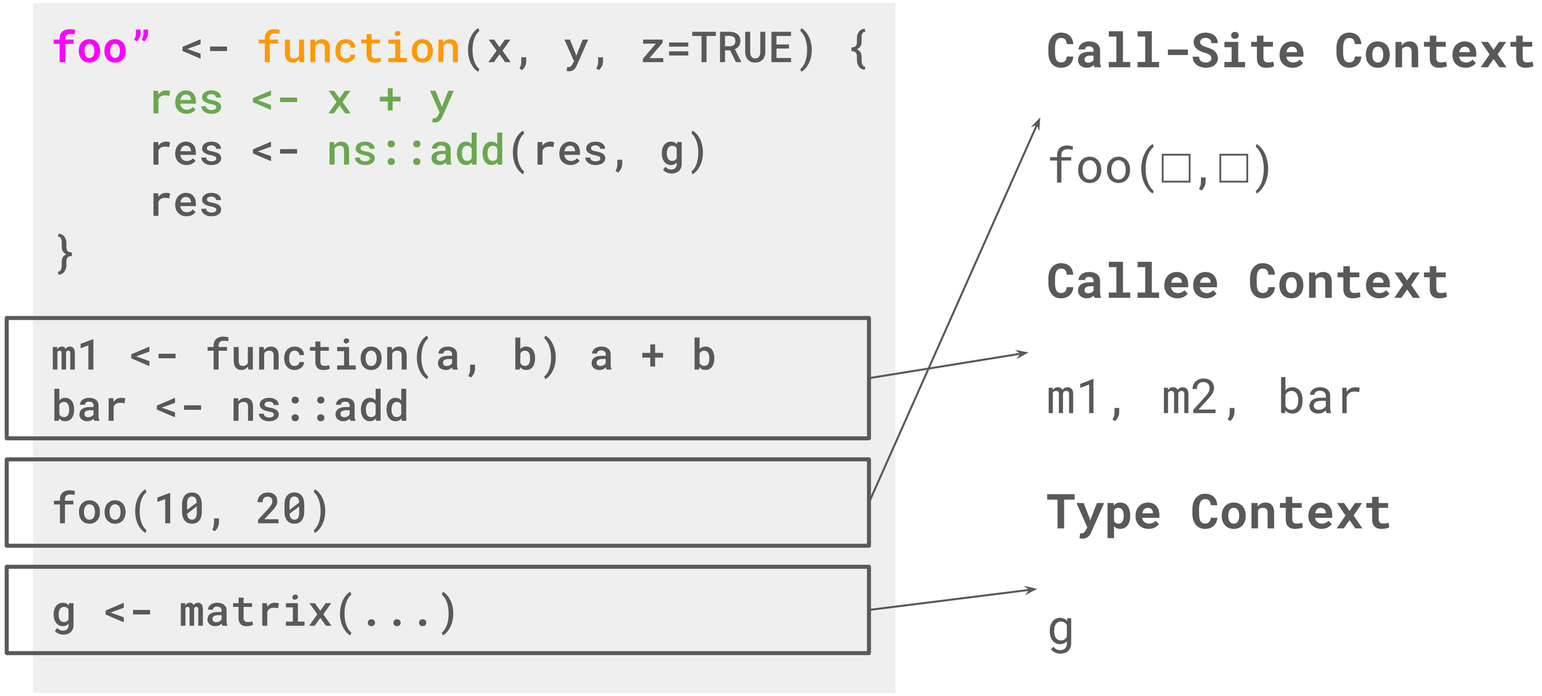
Call-Site Context

foo(□,□)

Callee Context

Type Context

Motivation



```
foo” <- function(x, y, z=TRUE) {
    res <- x + y
    res <- ns::add(res, g)
    res
}
```

```
m1 <- function(a, b) a + b
bar <- ns::add
```

```
foo(10, 20)
```

```
g <- matrix(...)
```

Motivation

```
foo" <- function(x, y, z=TRUE) {
    res <- x + y
    res <- ns::add(res, g)

}
```

```
m1 <
bar <- ns::add
```

```
foo(10, 20)
```

```
g <- matrix(...)
```

Call-Site Context

foo(□,□)

m1, m2, bar

Type Context

g

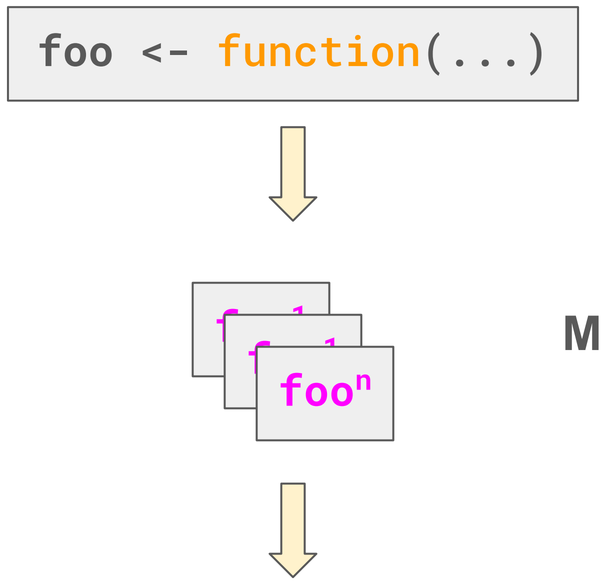
COMPILATION HAPPENS UNDER CONTEXT

Motivation

```
foo <- function(...)
```

One function

Motivation



```
foo <- function(...)
```

One function

Different users have different use-cases

Many different versions of the same function.

Reuse?

Motivation: Previous approaches

```
foo <- function(...)
```

One function

$\downarrow$

$\texttt{foo}^n$

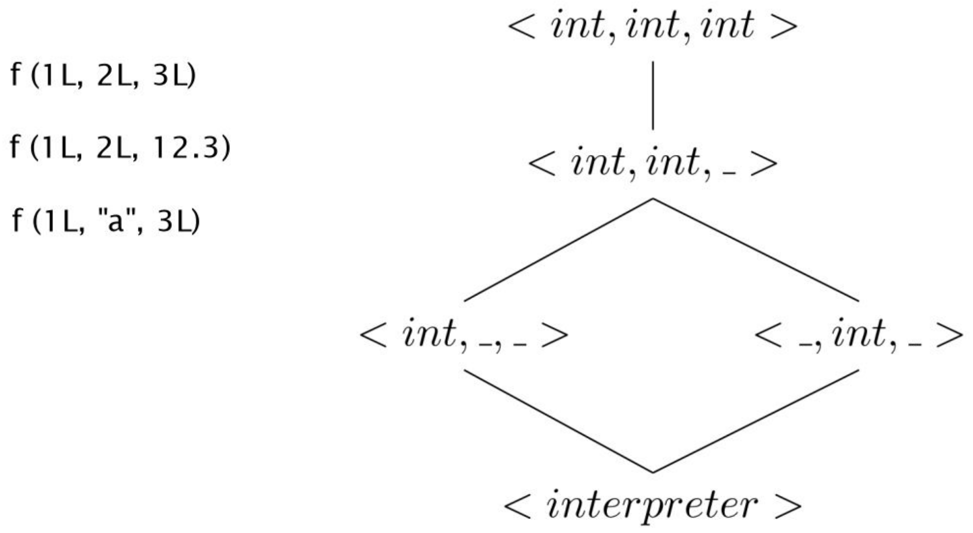
One size fits all, less optimized.

$\downarrow$

Lose peak performance

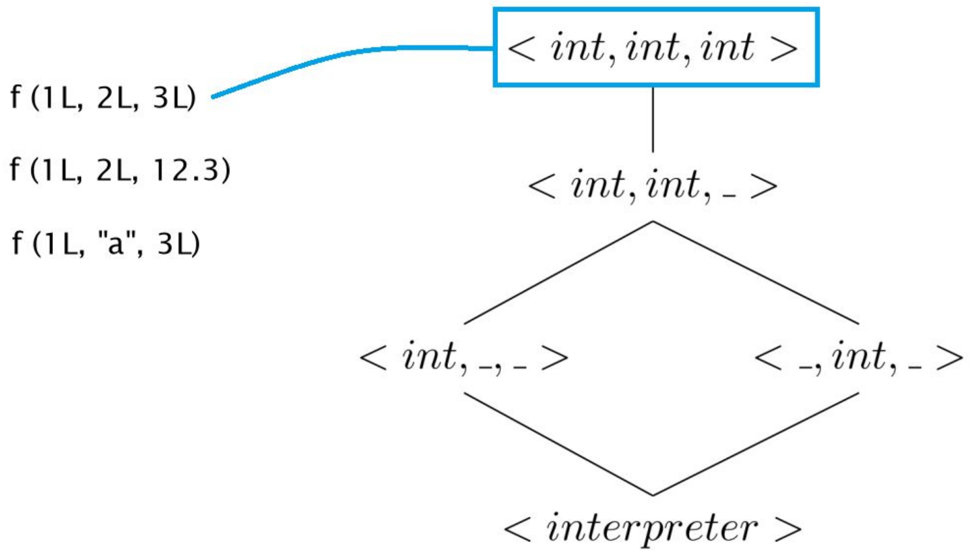
Background: Contextual Dispatch

Multiple versions of functions are dynamically dispatched on the basis of **call-site based contexts [Flückiger Et al.]**.



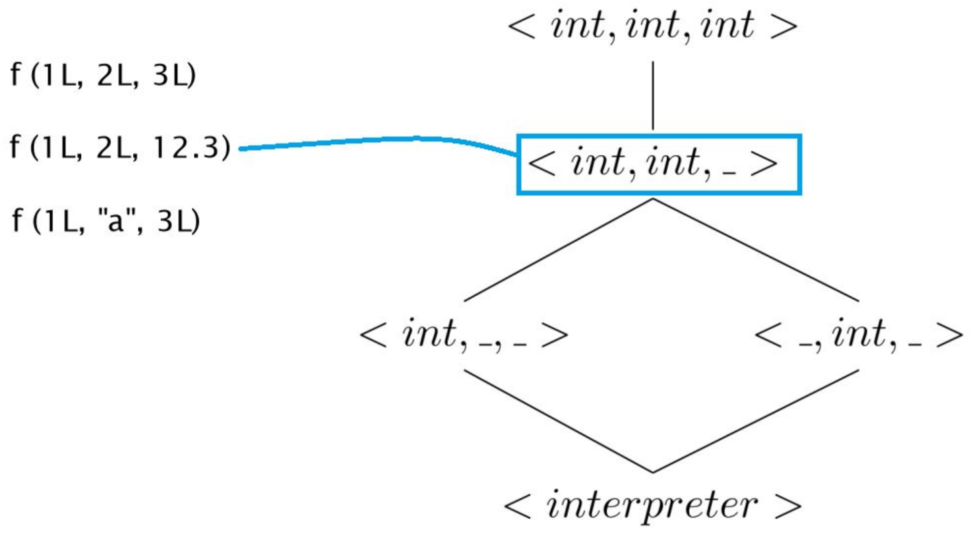
Background: Contextual Dispatch

Multiple versions of functions are dynamically dispatched on the basis of **call-site based contexts [Flückiger Et al.]**.



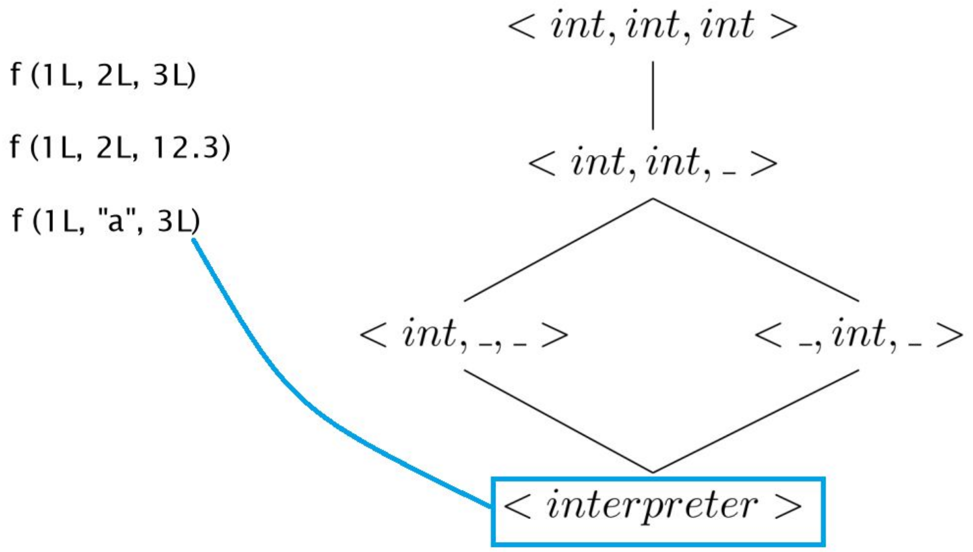
Background: Contextual Dispatch

Multiple versions of functions are dynamically dispatched on the basis of **call-site based contexts [Flückiger Et al.]**.



Background: Contextual Dispatch

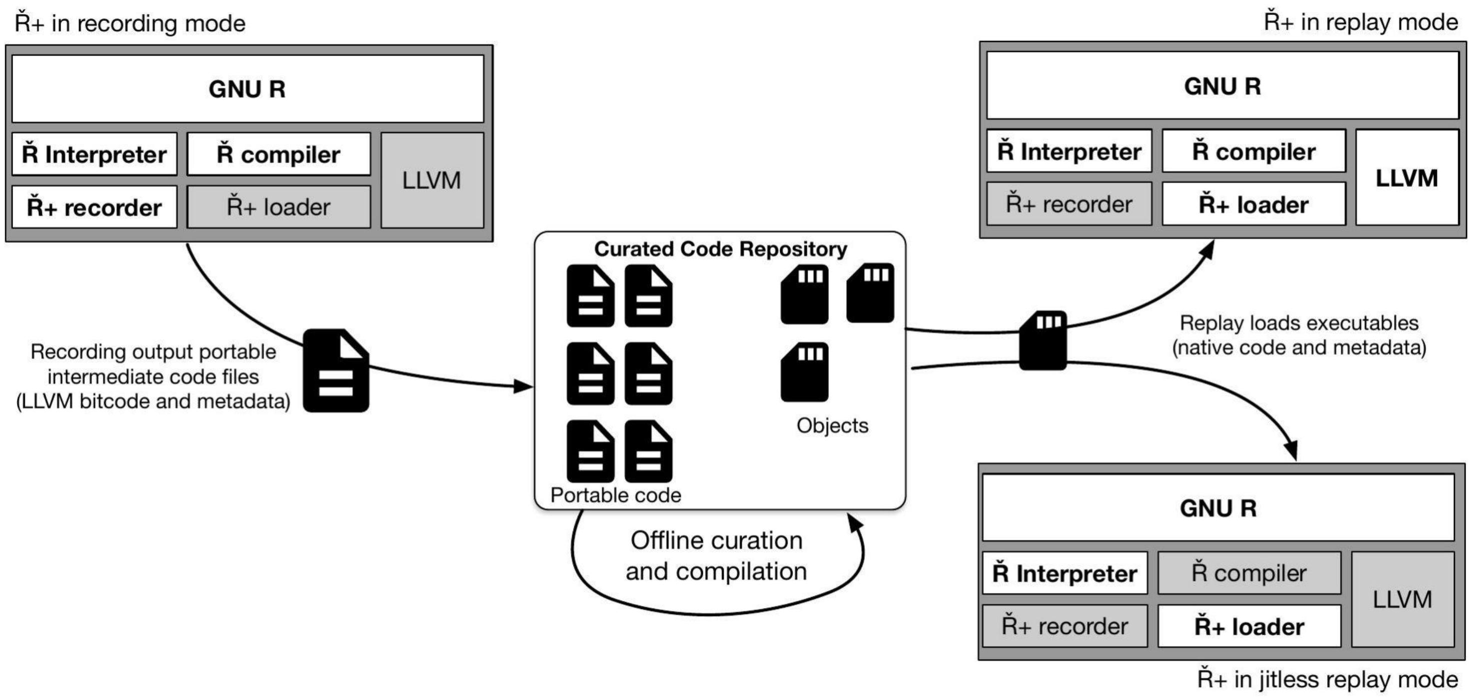
Multiple versions of functions are dynamically dispatched on the basis of **call-site based contexts [Flückiger Et al.]**.



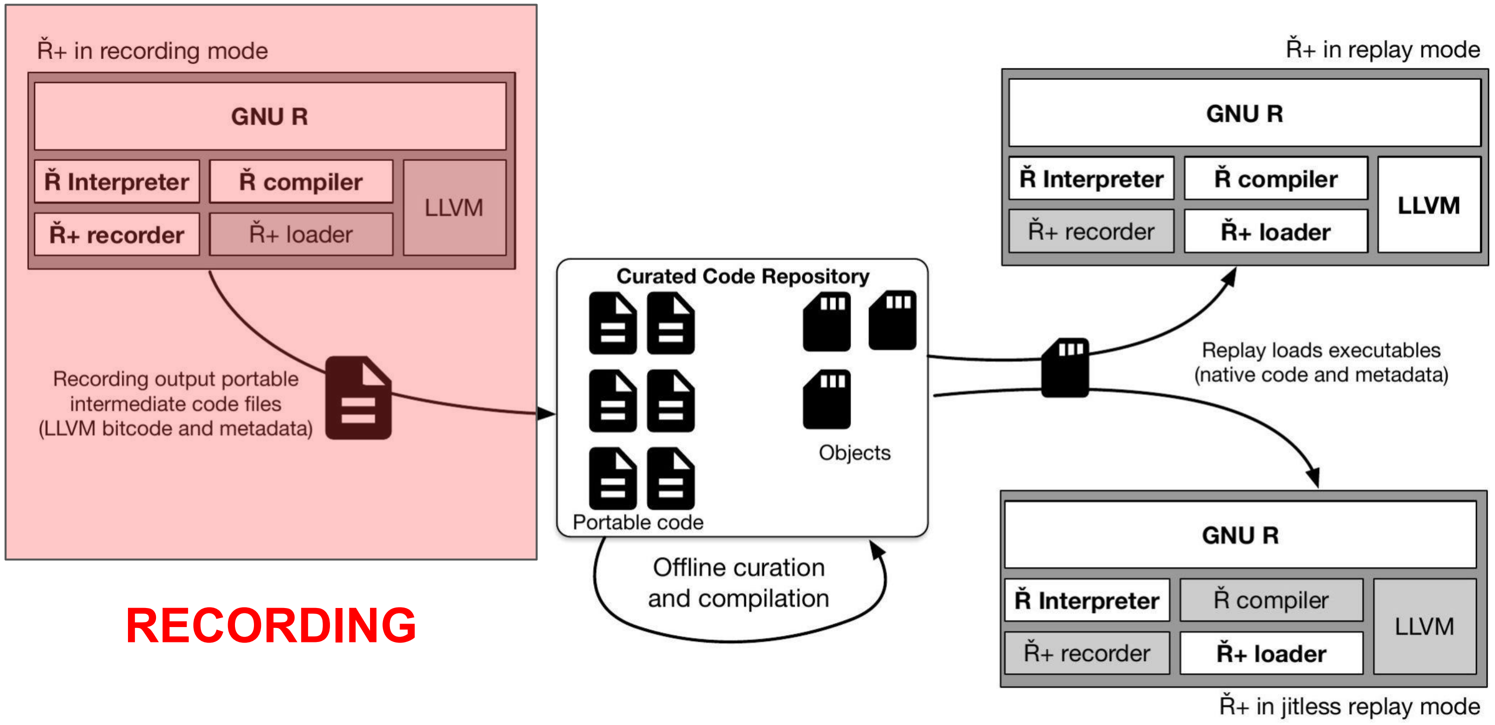
Challenges

1. **Recording** - *How to save the compiled code?*
2. **Duplicates** - *Identification and removal of duplicates*
3. **Reuse** - *Out of multiple options, which one to dispatch?*

Our Approach: Ř+

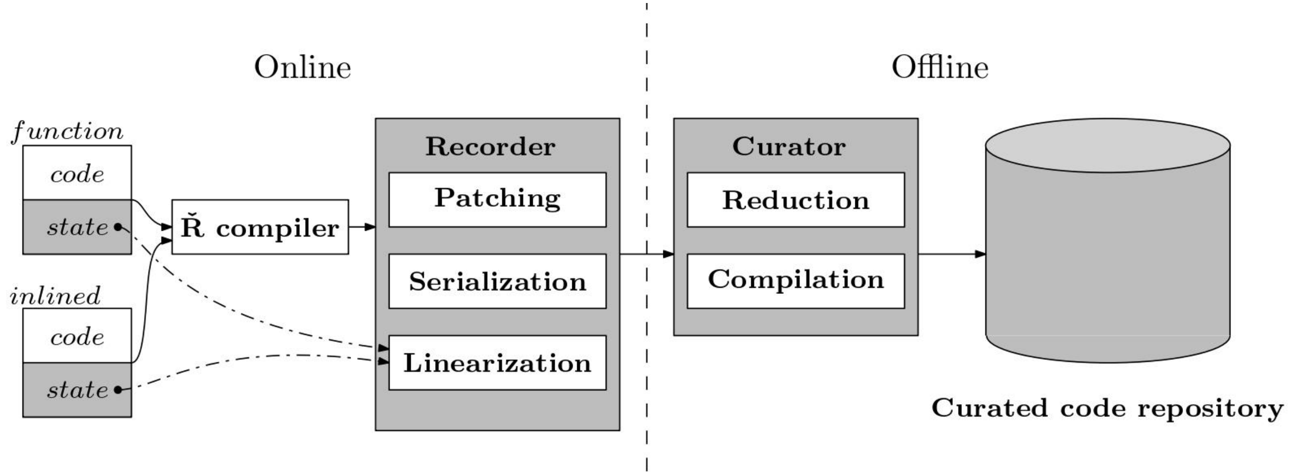


Our Approach: Ř+



RECORDING

System Overview



```
Compiled code:      *.bc   (LLVM bitcode   file)
Pool references:    *.pool (encoded binary file)
Function metadata:  *.meta (encoded binary file)
```

Speculative Contexts

```
Ř+ recording saves f' of a function f

        f' = compiler(f)
```

What it looks like

Speculative Contexts

`Ř+ recording saves f’ of a function f`

`f’ = compiler(⟨Code, ⟨C,F⟩⟩)`

What it is! :O

Speculative Contexts

Pair of ⟨**C**,**F**⟩

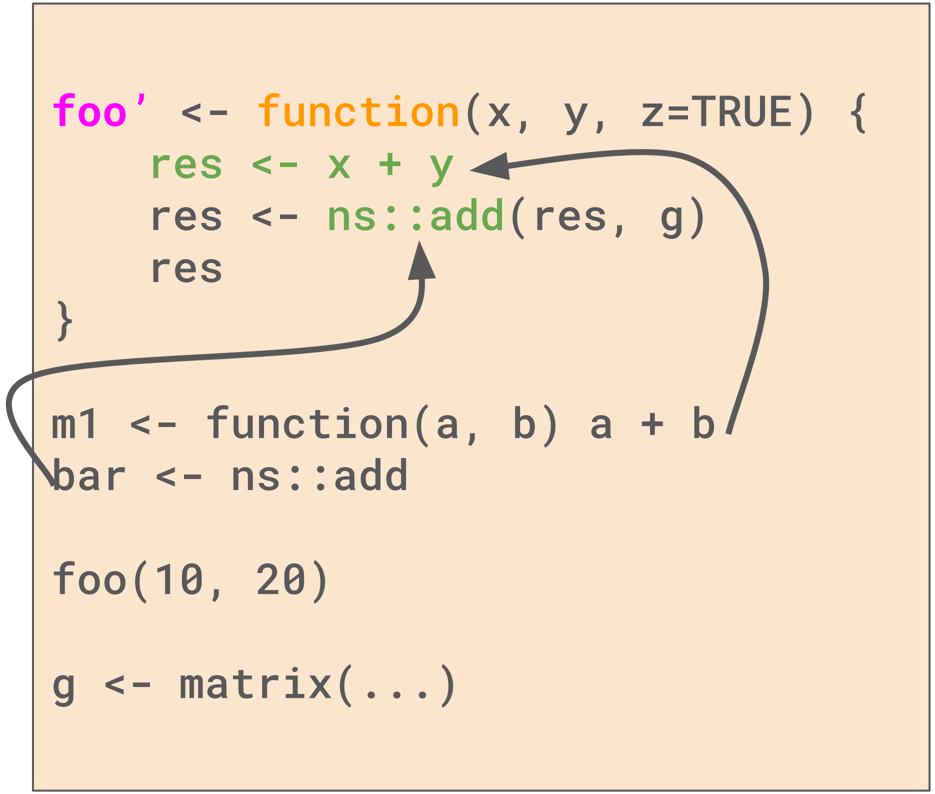
`f(□,□,□)`

Set of predicates on the call-site arguments

Speculative Contexts

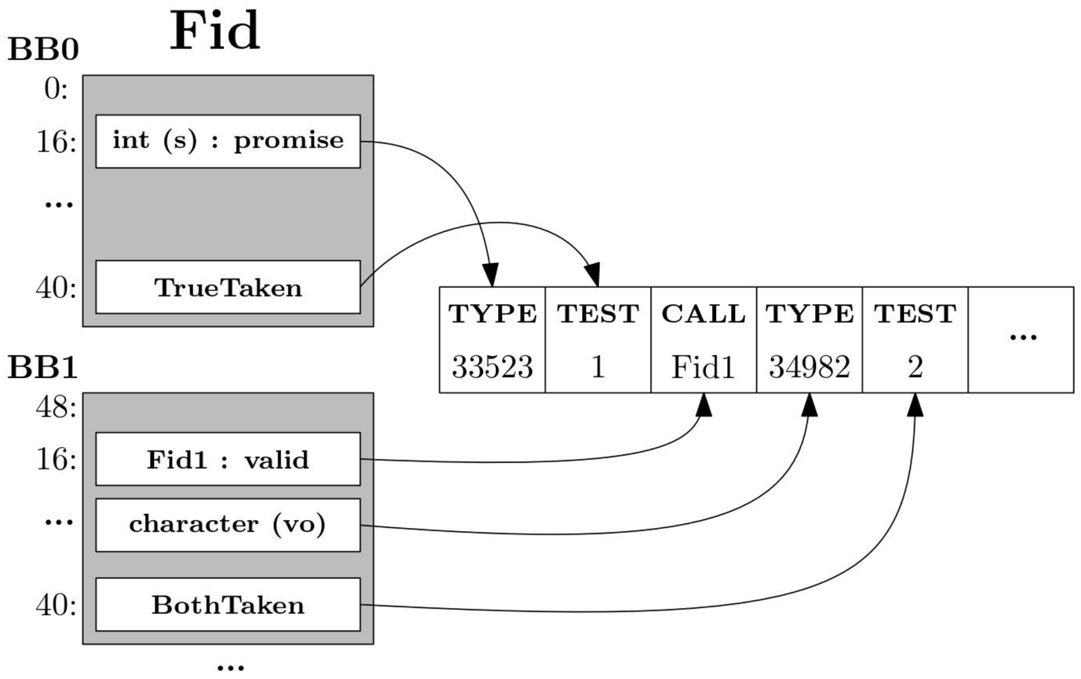
Pair of ⟨**C**,**F**⟩

F is a vector that holds feedback context.

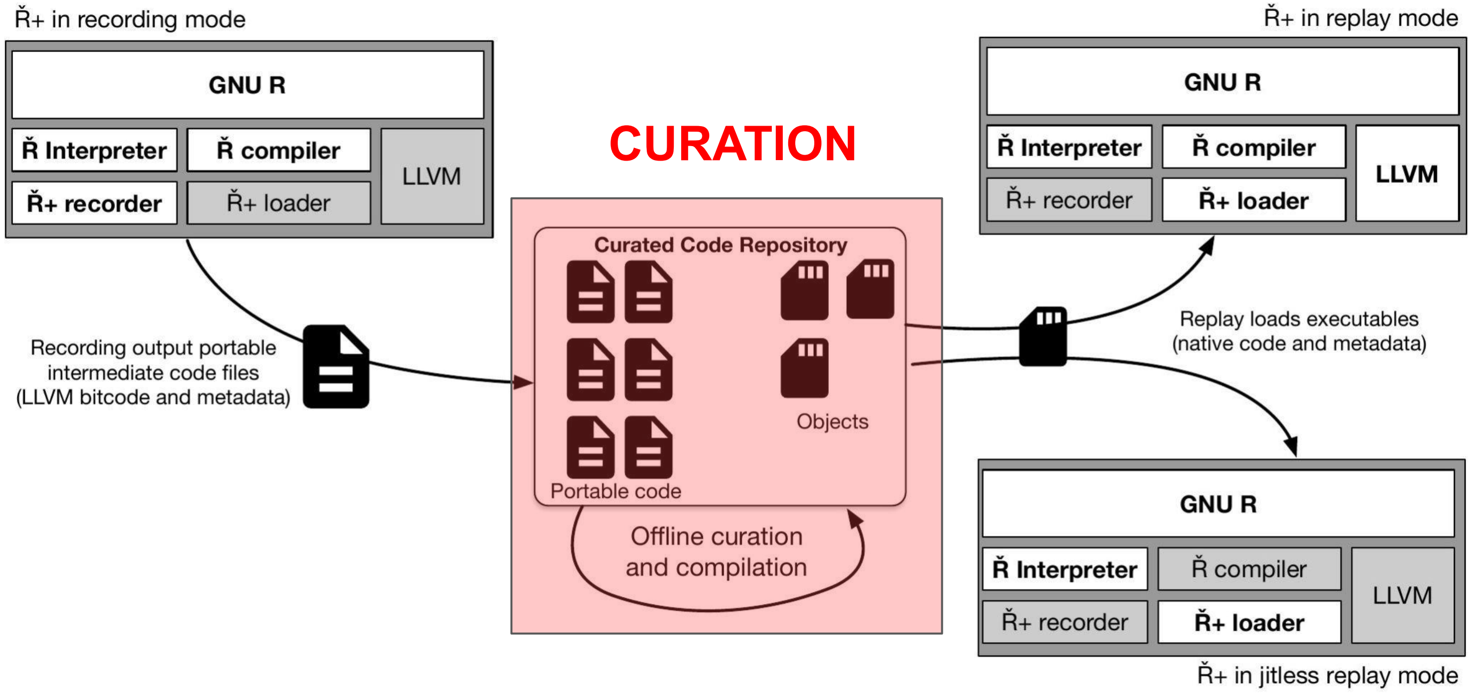


```
foo' <- function(x, y, z=TRUE) {
    res <- x + y
    res <- ns::add(res, g)
    res
}

m1 <- function(a, b) a + b
bar <- ns::add

foo(10, 20)

g <- matrix(...)
```

System Overview



Our Approach: Ř+



Curation

The complete compilation context contains:

Complete: ⟨(int, int), ⟨m1_inlined, ns::add_static, g_matrix⟩⟩

Curation

The complete compilation context contains:

Complete: ⟨(int, int), ⟨m1_inlined, ns::add_static, g_matrix⟩⟩

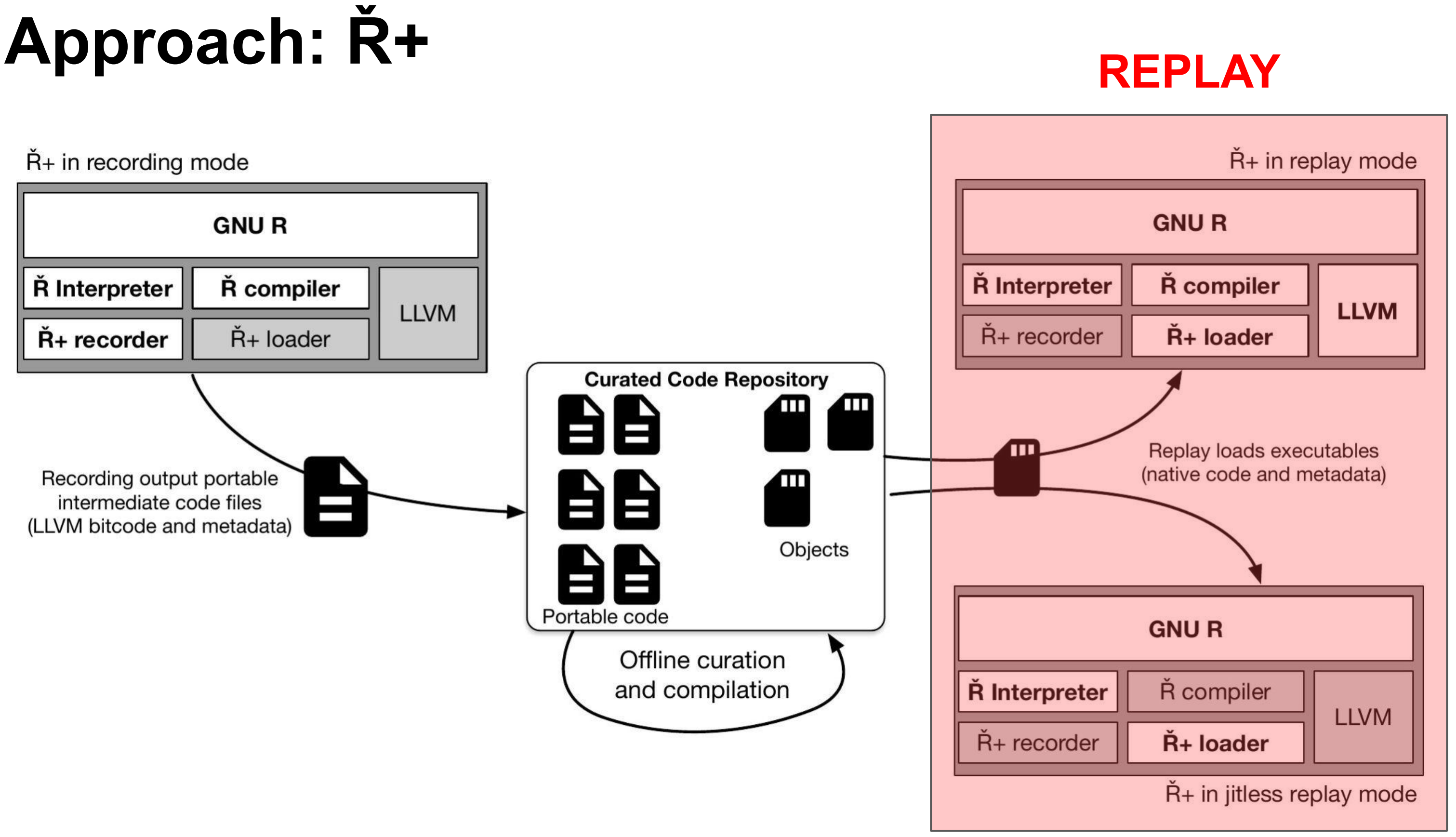
Useful(D): ⟨(int, int), ⟨m1_inlined, ns::add_static⟩⟩

We identify the useless part of the context using deopt points D.

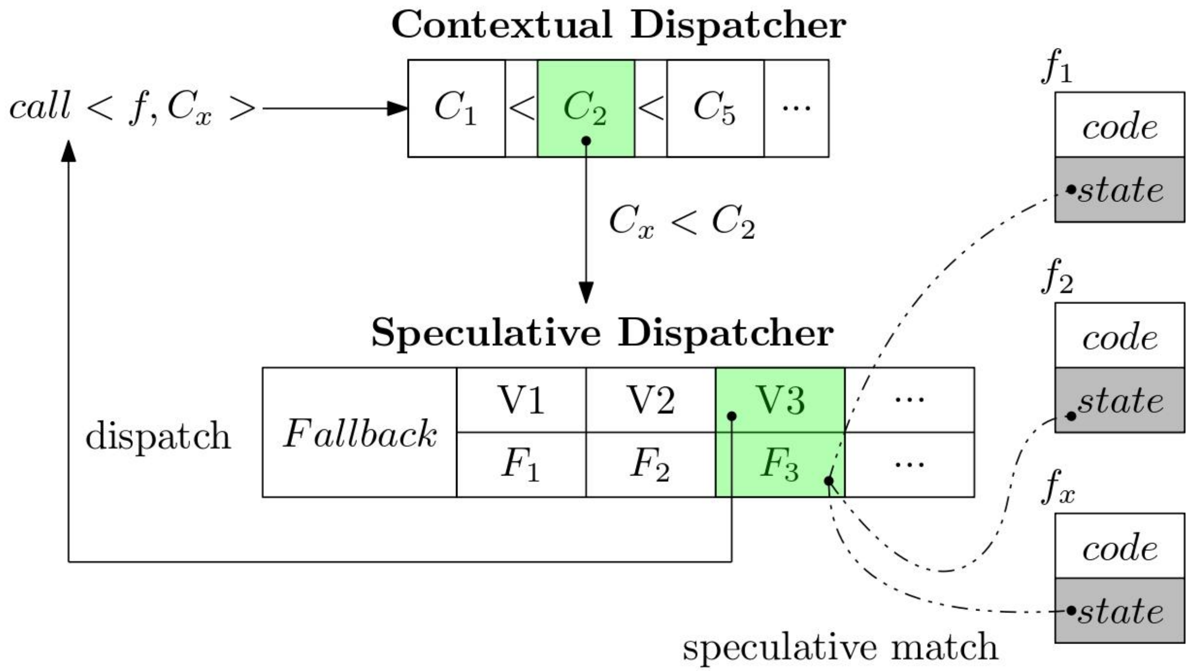
Curation

```
V’: ⟨C, D⟩

V”: ⟨C’, D’⟩



C == C' and D == D', both are same

V == V', keep version where C' < C
```

Our Approach: Ř+



The dispatcher



The dispatcher

```
Function* Dispatcher::dispatch() {
  if (cache != NULL) return cache;
  for (int i = length() - 1; i >= 0; i--) {
    auto f = getFunction(i);
    if (f->abled() && f->matchD())
      return cache = f;
  }
  return cache = getFallback();
}
```

Contextual dispatcher ensures correctness

Speculative dispatcher ensures precision

The dispatcher

```
Function* Dispatcher::dispatch() {
  if (cache != NULL) return cache;
  for (int i = length() - 1; i >= 0; i--) {
    auto f = getFunction(i);
    if (f->abled() && f->matchD())
      return cache = f;
  }
  return cache = getFallback();
}
```

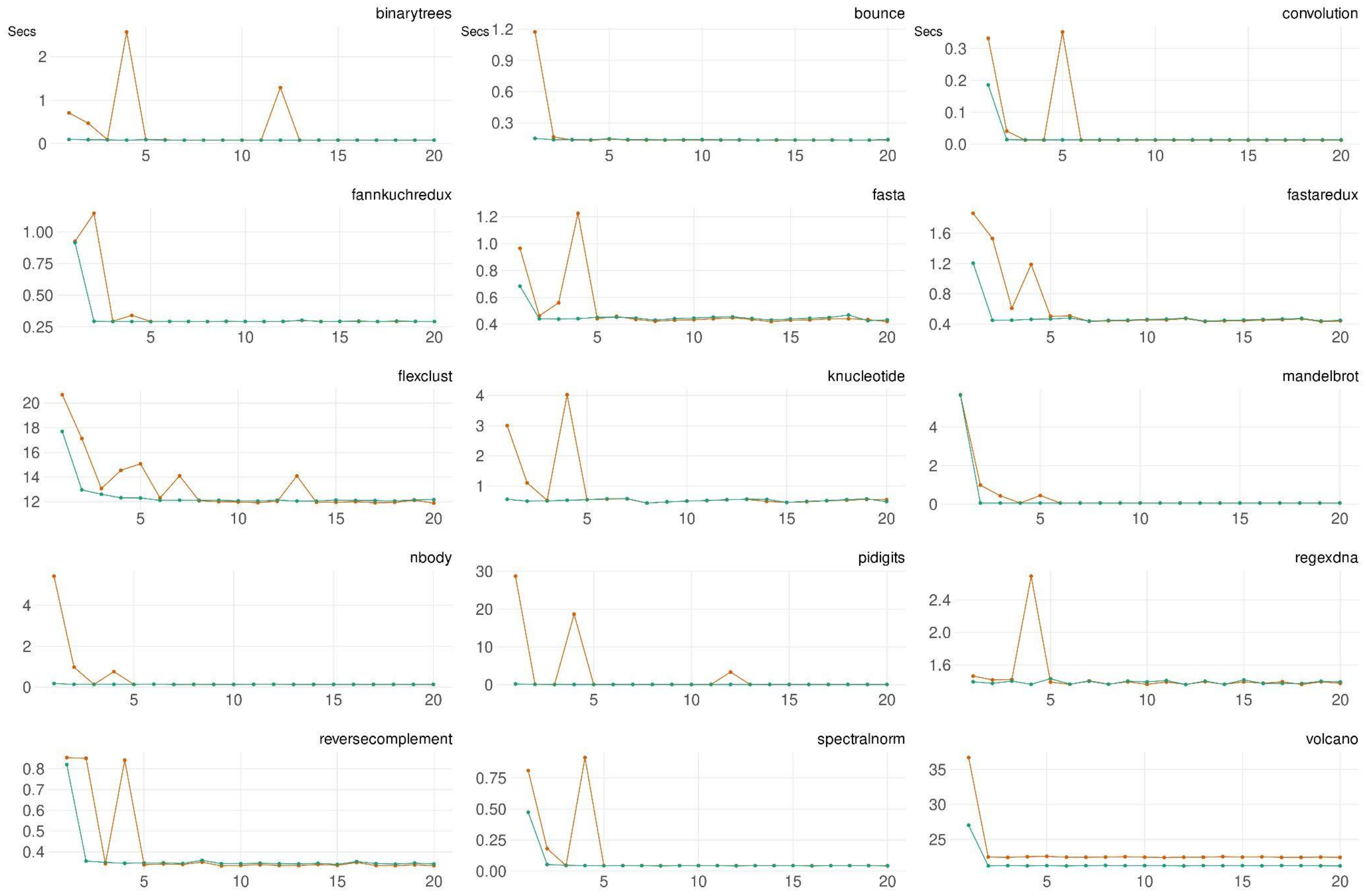
Expensive check needs to happen only when state changes :)

Otherwise we can keep a **fast cache**.

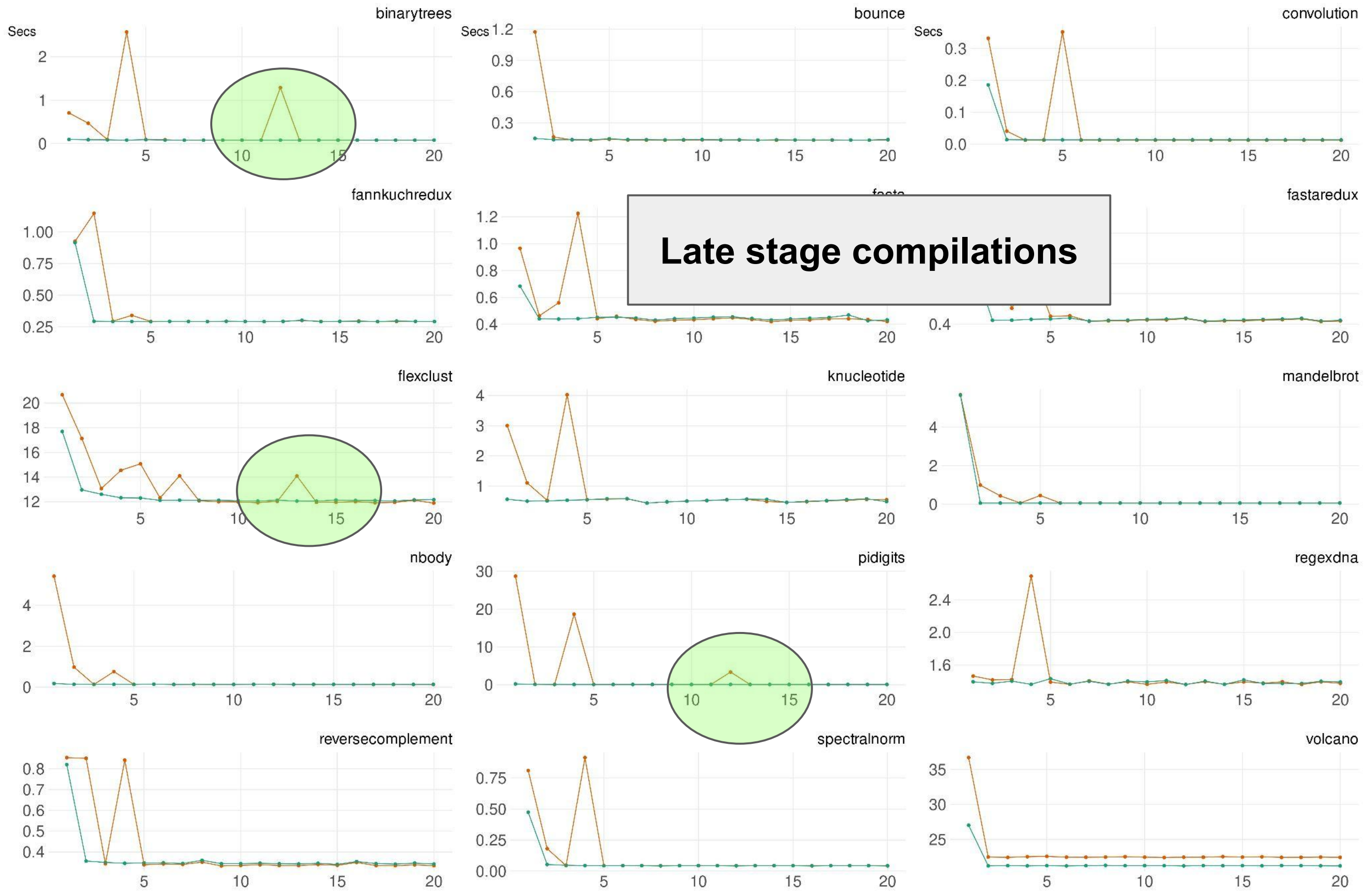
Contextual dispatcher ensures correctness

Speculative dispatcher ensures precision

R-benchmarking suite

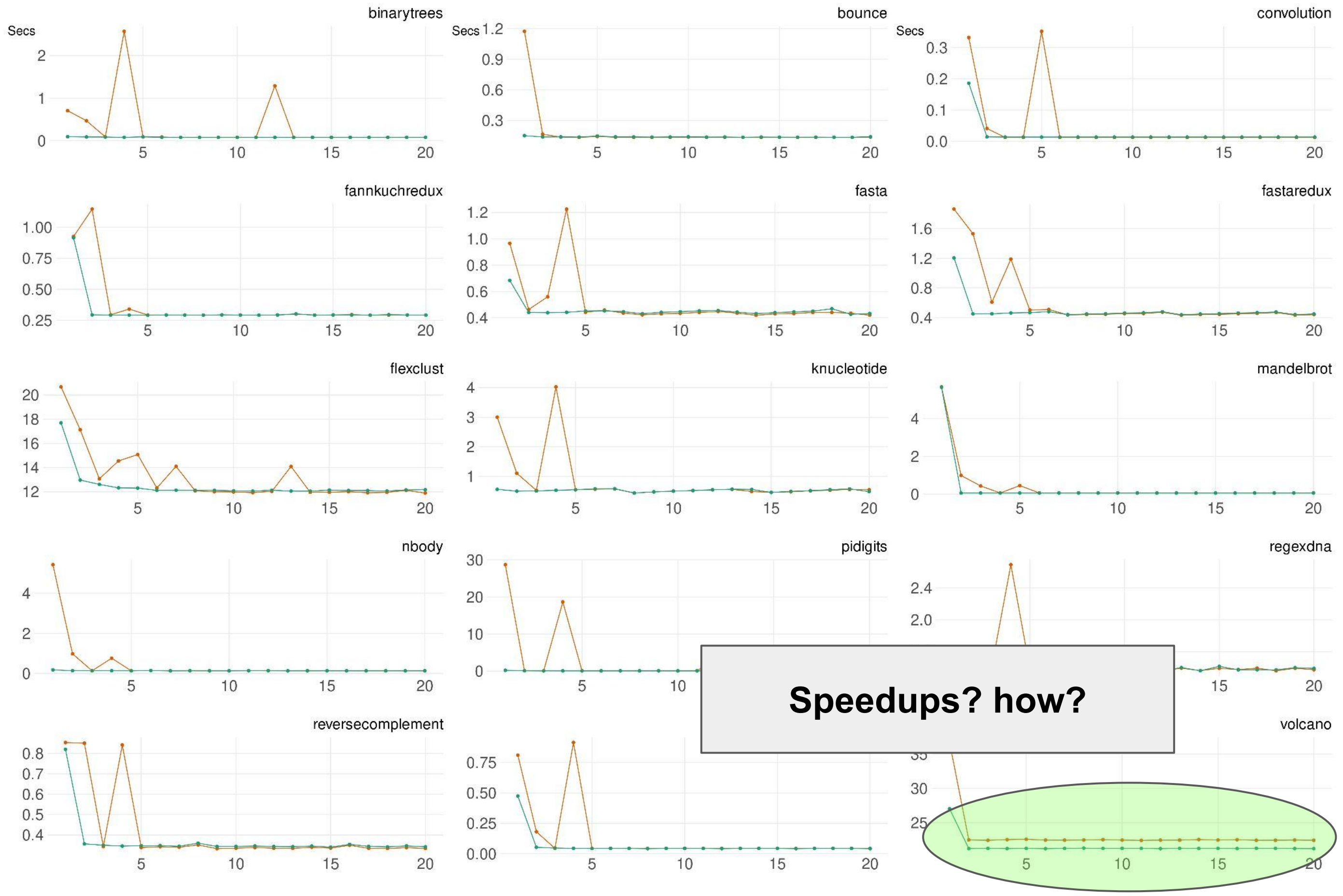


R-benchmarking suite



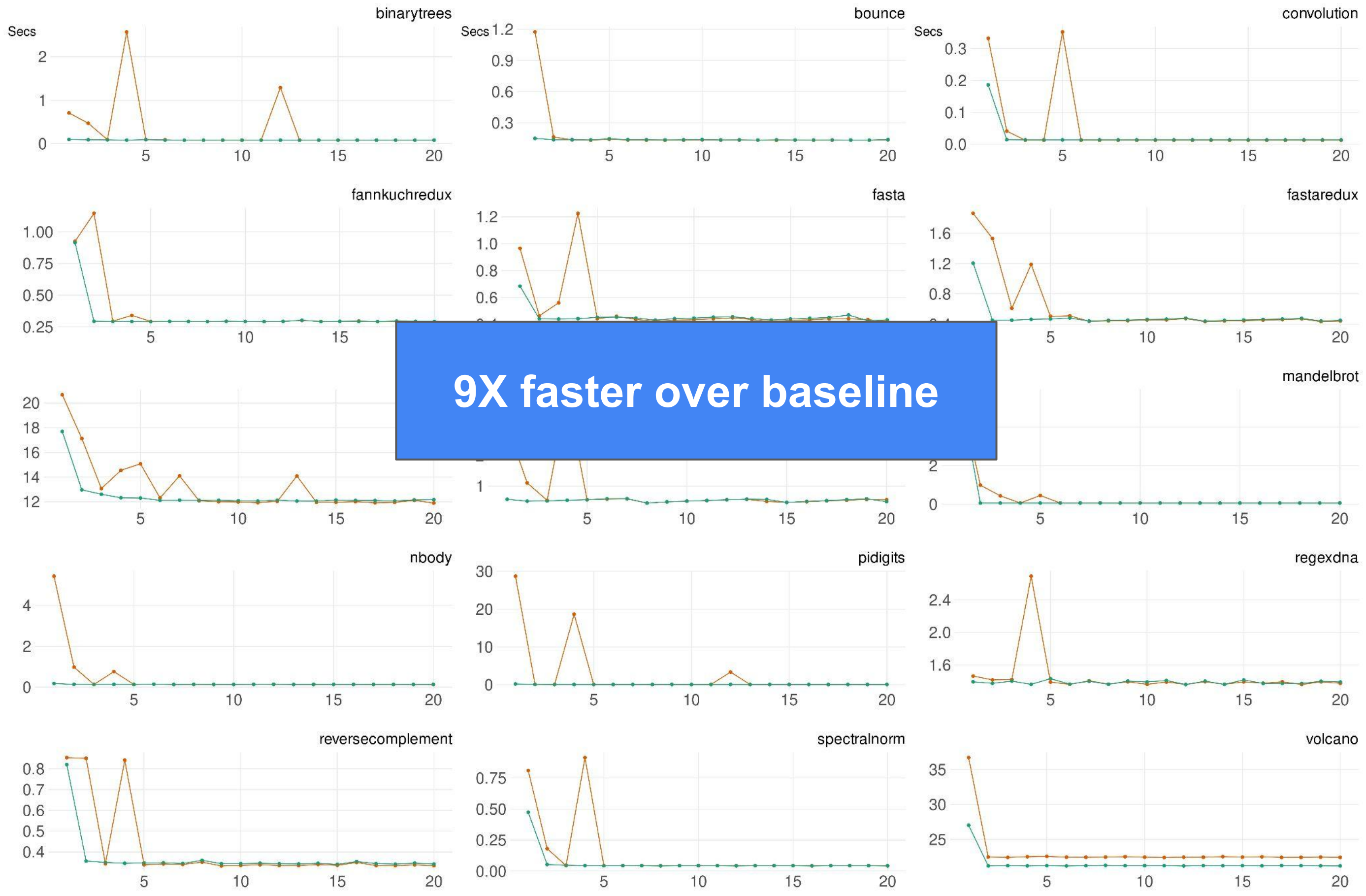
Late stage compilations

R-benchmarking suite



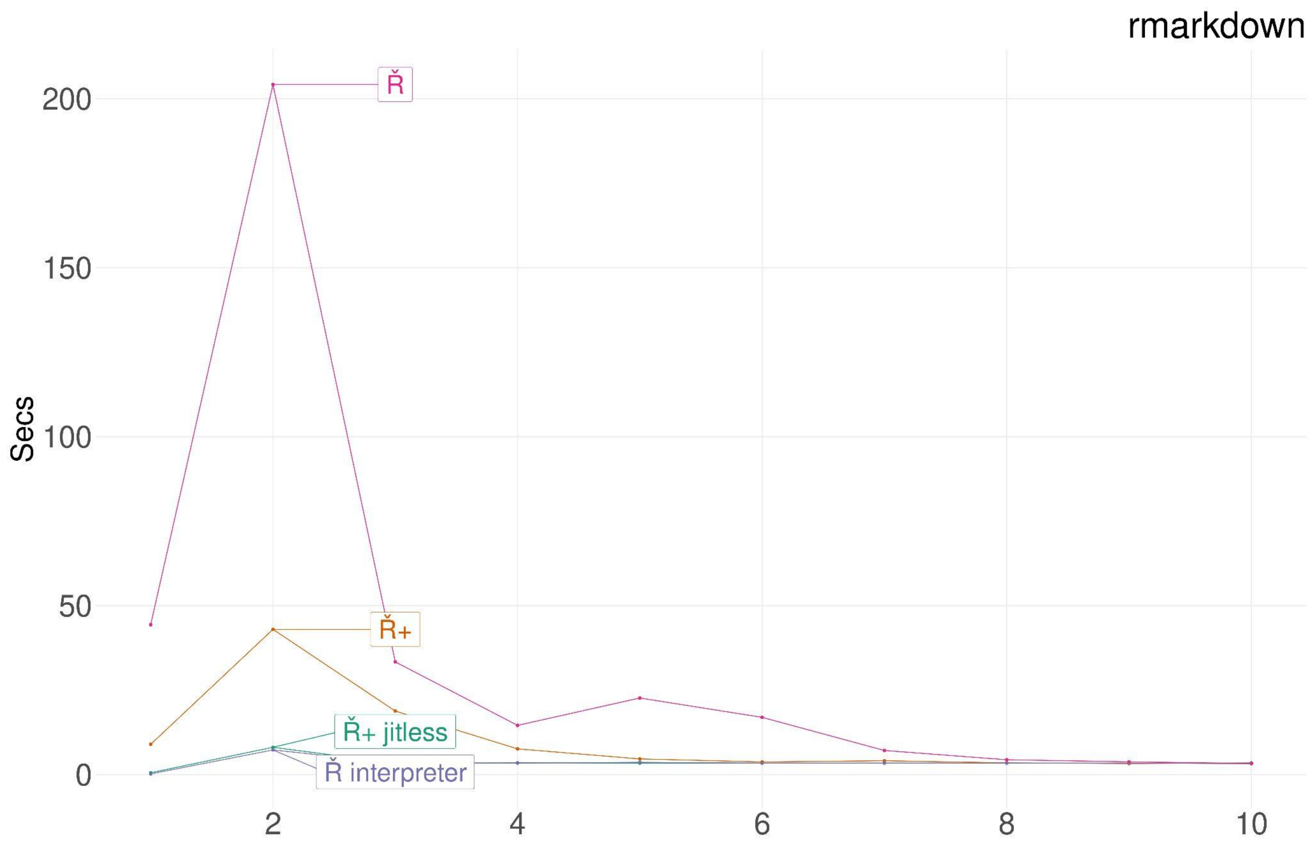
Speedups? how?

R-benchmarking suite

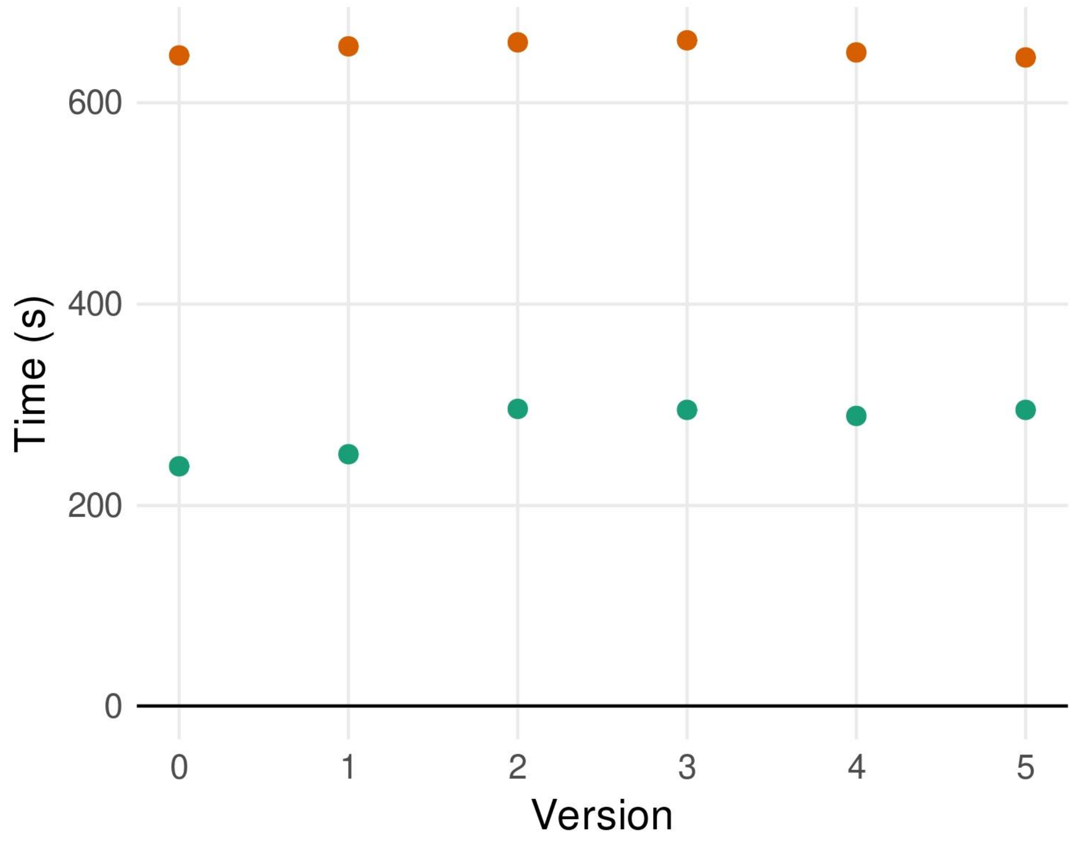


9X faster over baseline

Real-World Use Case

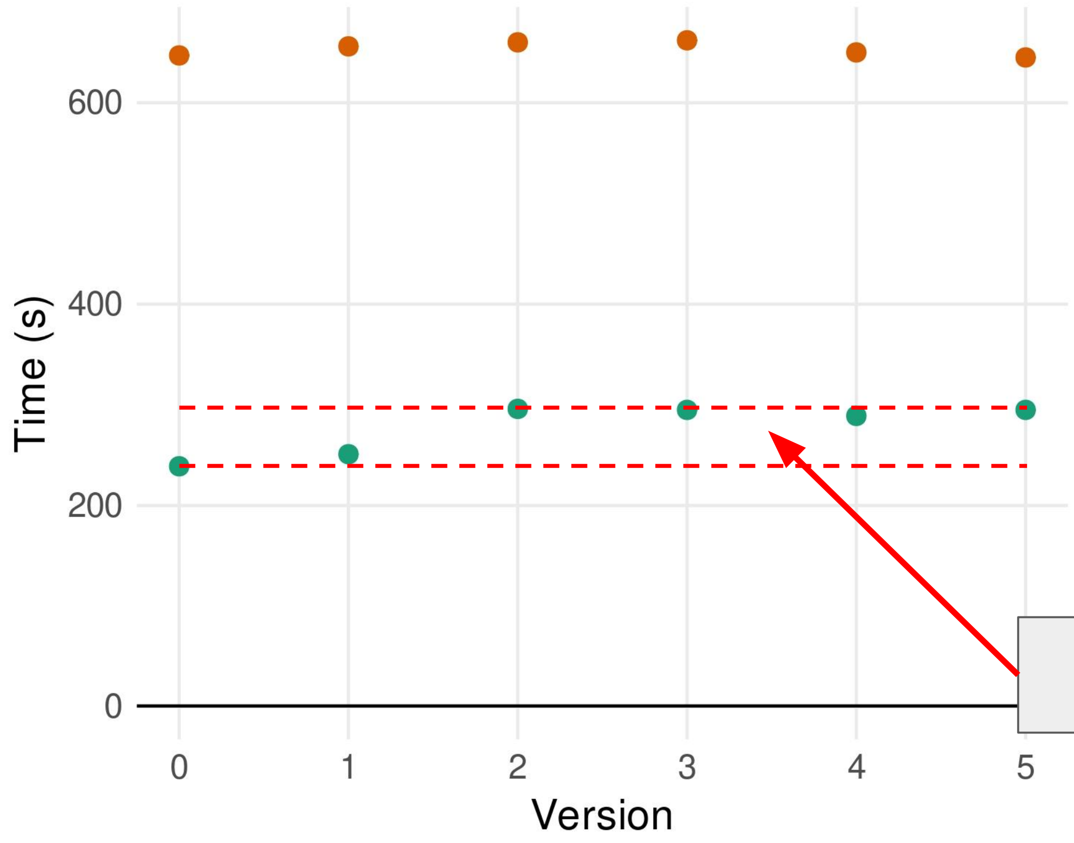


End-to-End Performance



Code changes over time

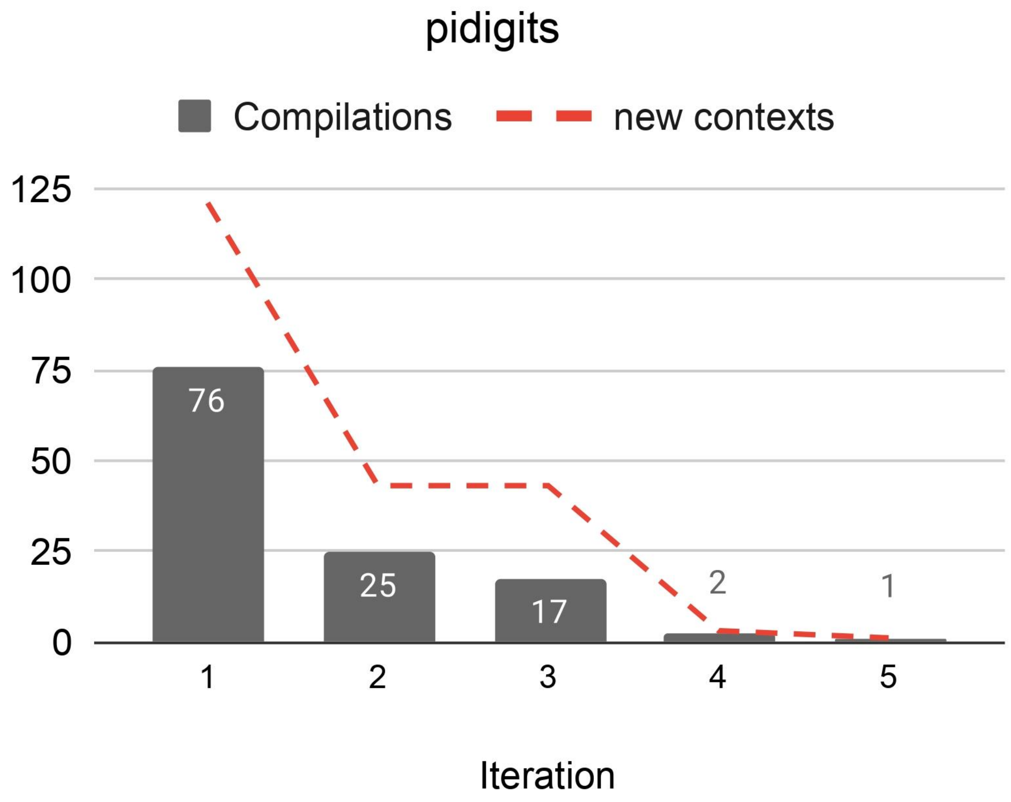
End-to-End Performance



Code changes over time

New code over time

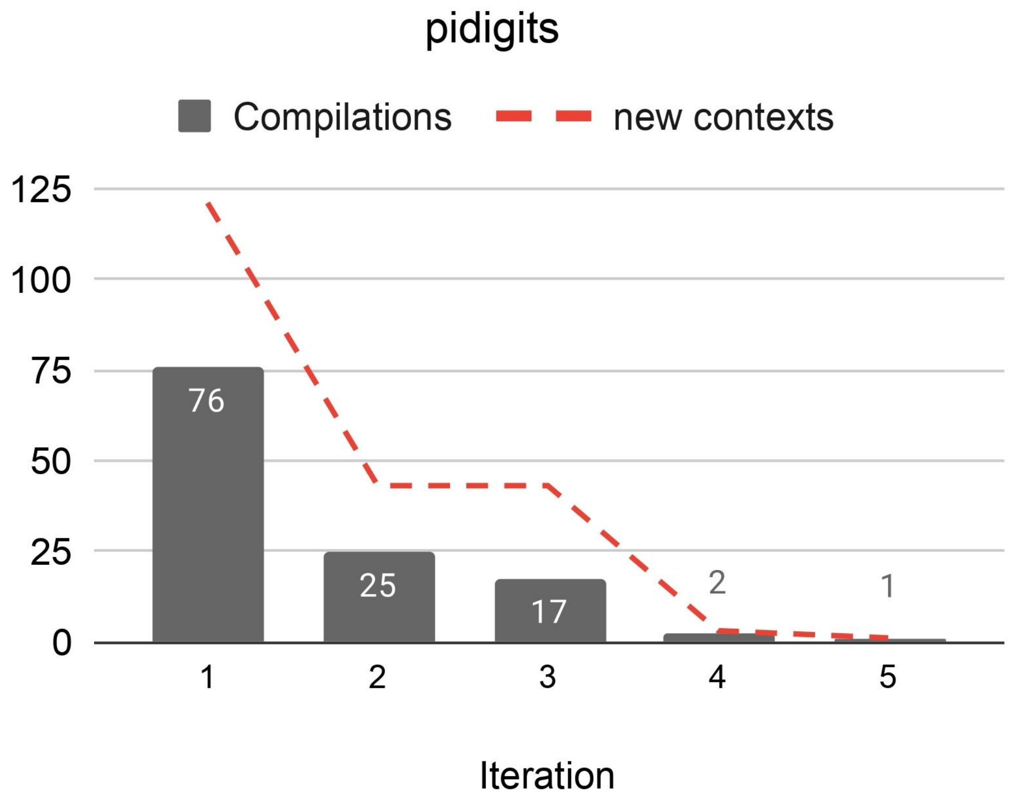
Repository Construction



Same program,
Same inputs,

Iterative **loop**

Repository Construction

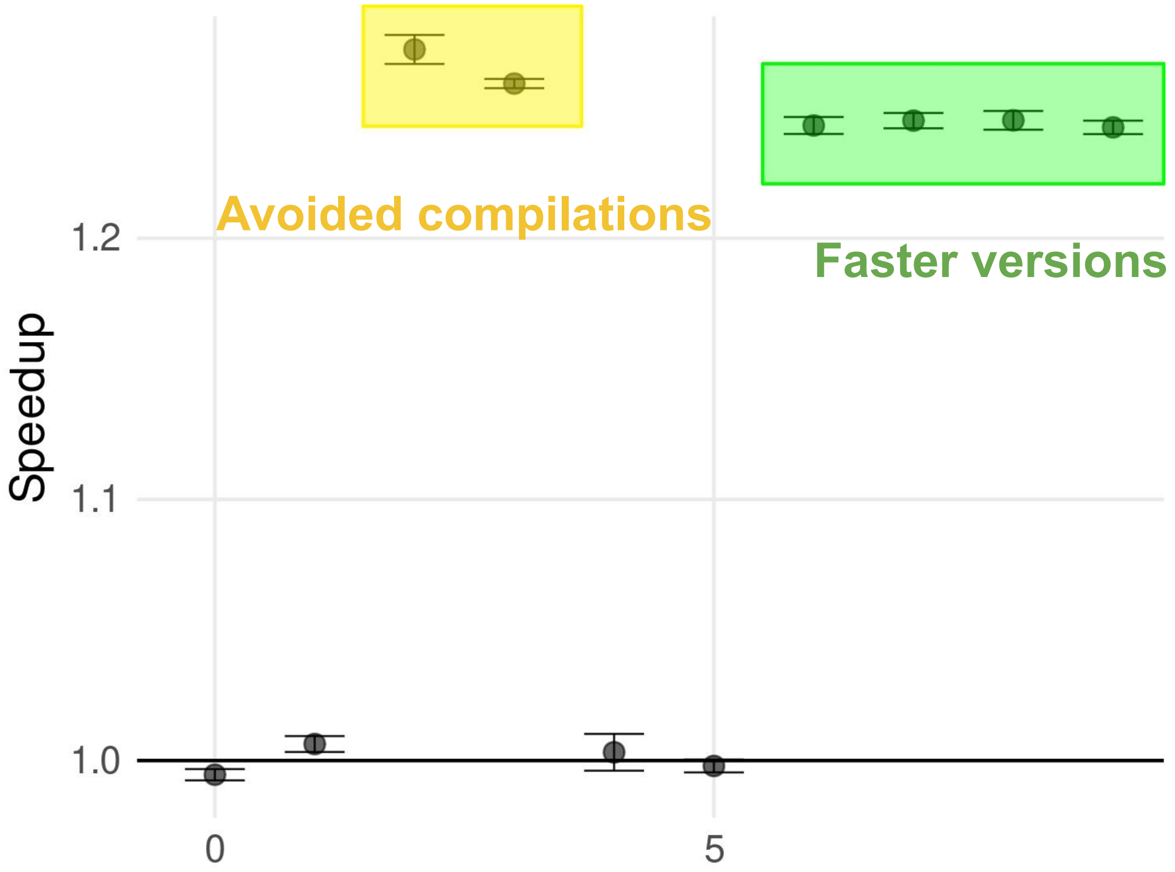


Same program,
Same inputs,

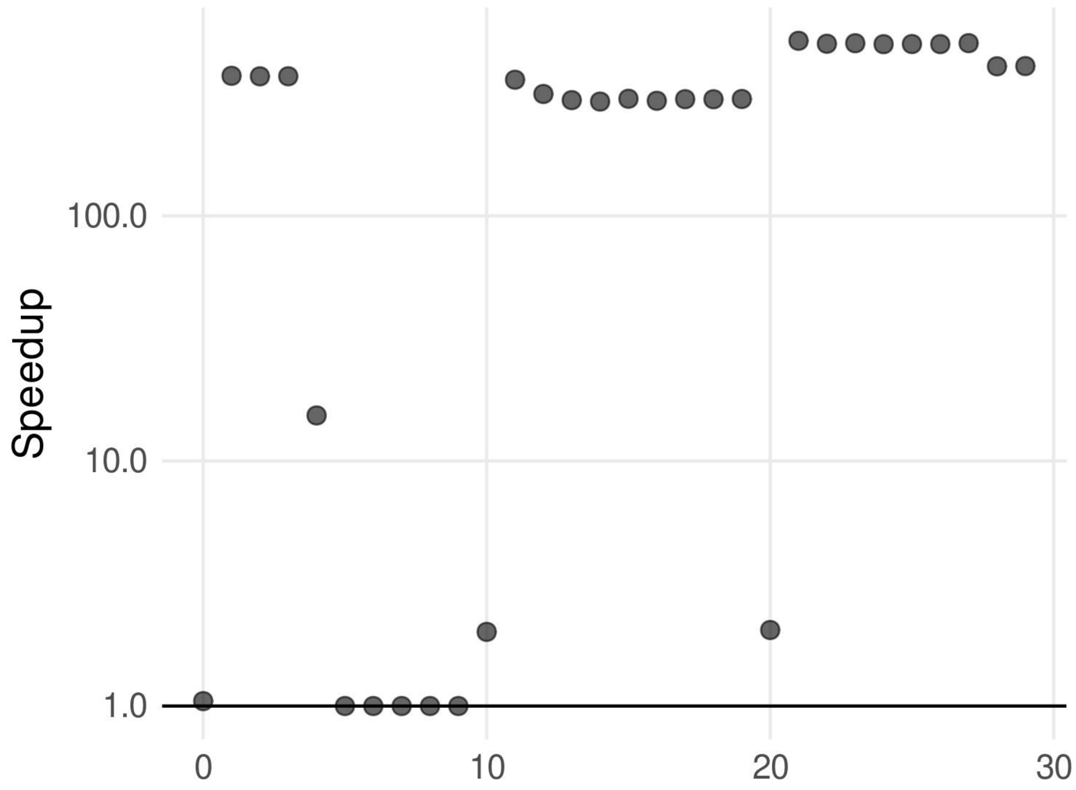
Iterative **loop**

Why still compilations?

Phase Change Behaviour



Phase Change Behaviour



Conclusion

Less **compilations**, fancy **dispatcher**, runs **faster**

- **Speculative Dispatcher ->** *more complex operations?*
- **Global optimization of serialized code?**
- **More de-optimizations are good???**
- **Current work only focuses on maintaining JIT performance.**
- **But we show cases where we exceed it, this requires more careful exploration.**

Thank you :)