Between the Lines: VM Assumptions

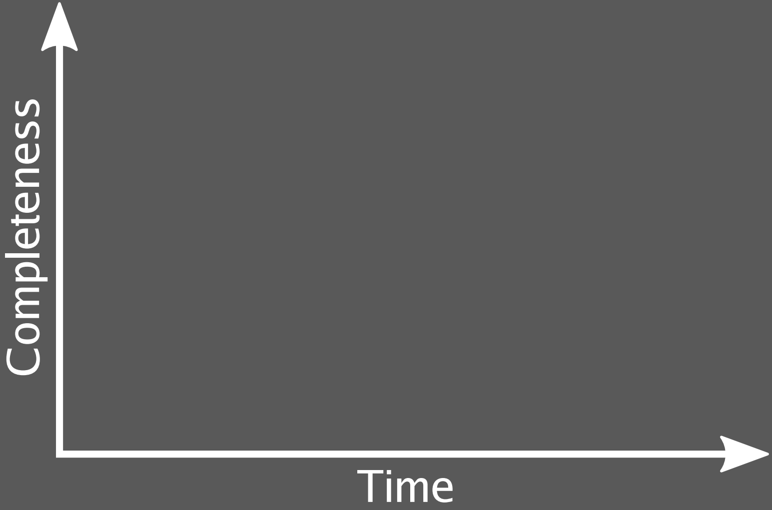
Laurence Tratt
https://tratt.net/laurie/

2020-11-05

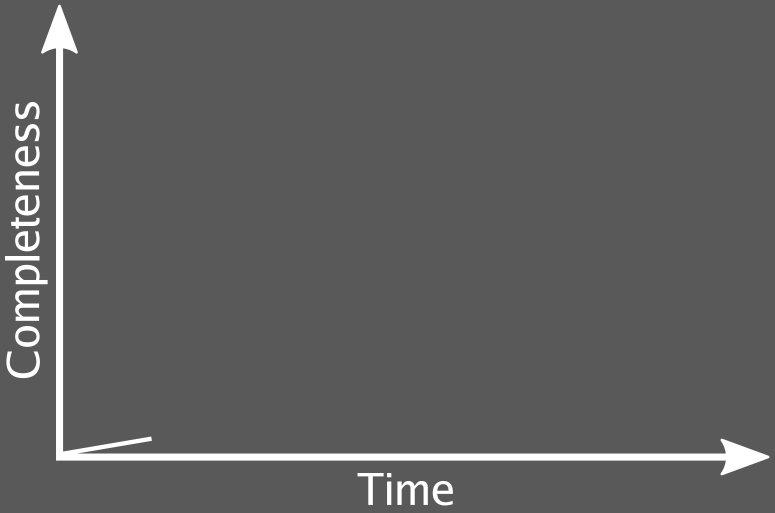
What happens if we’re wrong?

VM development is a destination

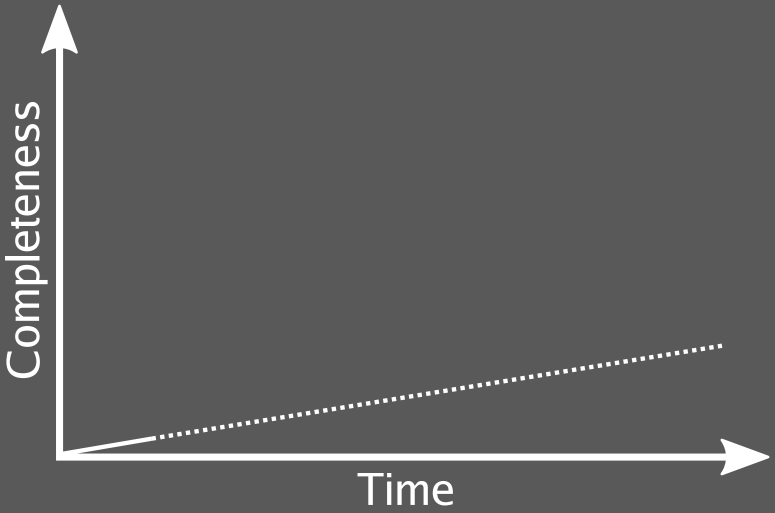
VM development is a destination



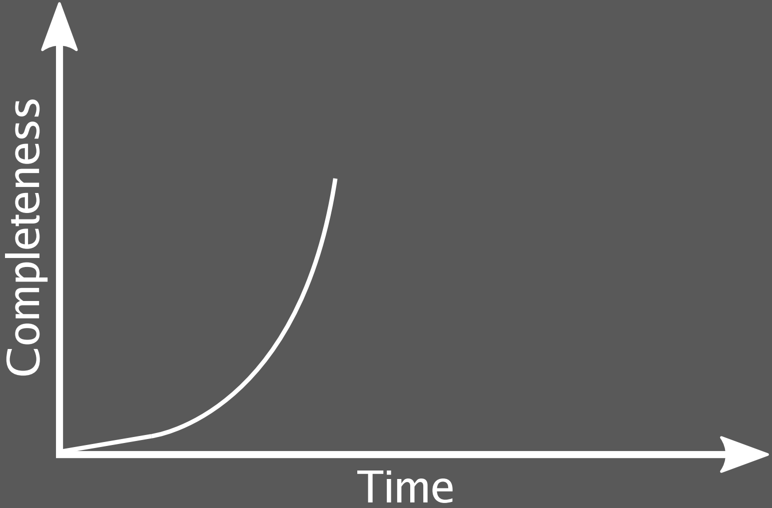
VM development is a destination



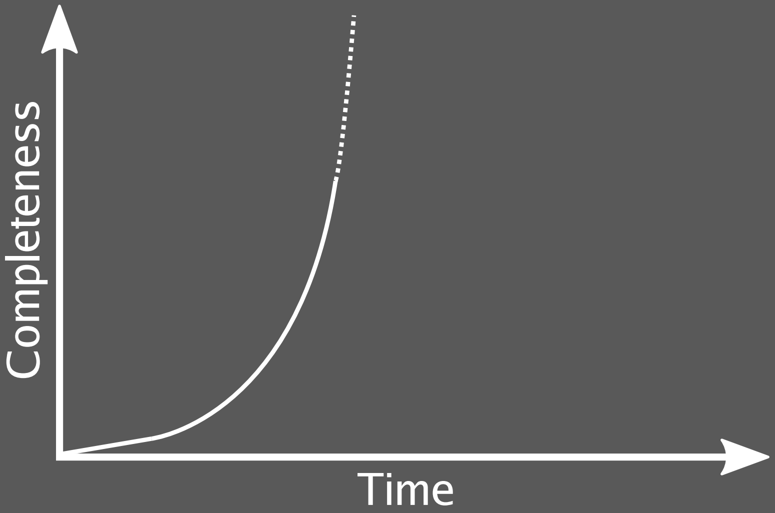
VM development is a destination



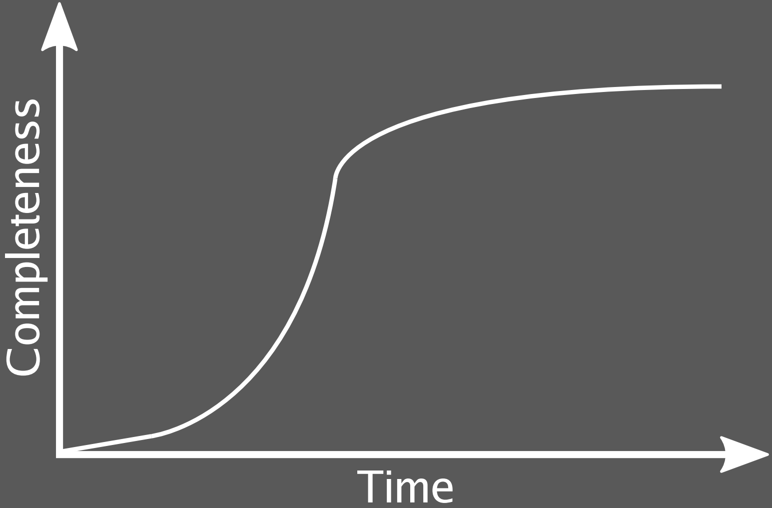
VM development is a destination



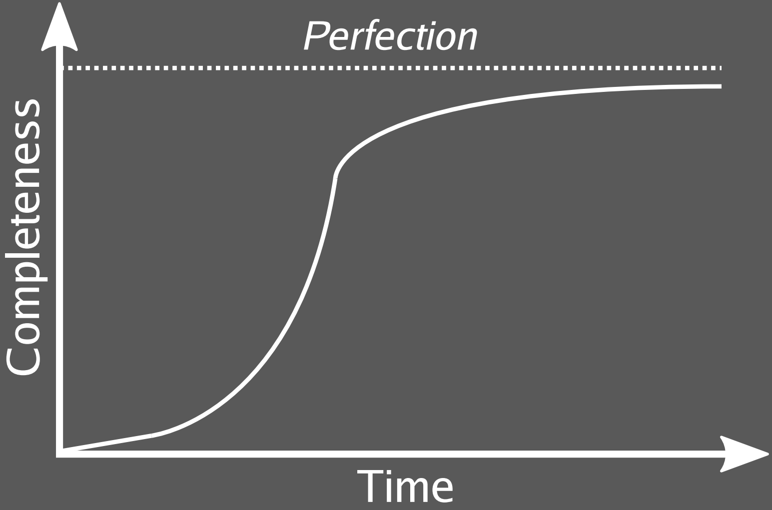
VM development is a destination



VM development is a destination



VM development is a destination



VM development is a destination

VM development is a destination

VM development is a never-ending process

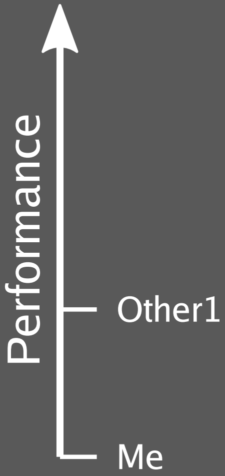
We know where the performance ceiling is

We know where the performance ceiling is

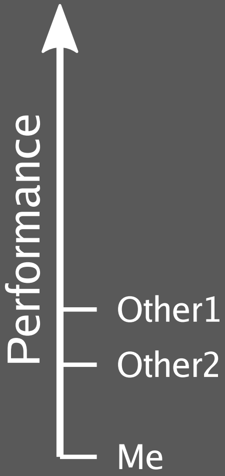
What is the best possible performance for an input P?

We know where the performance ceiling is

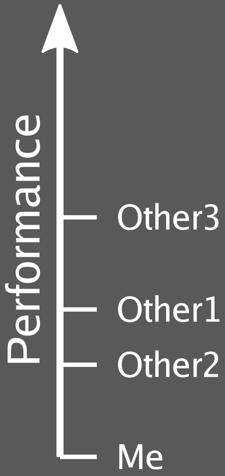
We know where the performance ceiling is



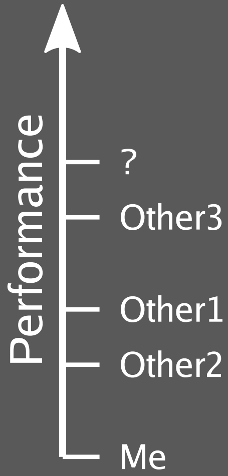
We know where the performance ceiling is



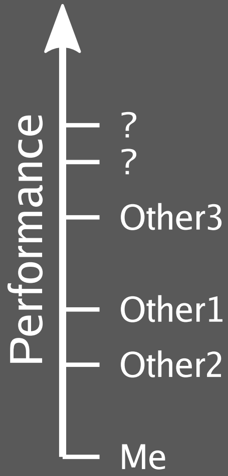
We know where the performance ceiling is



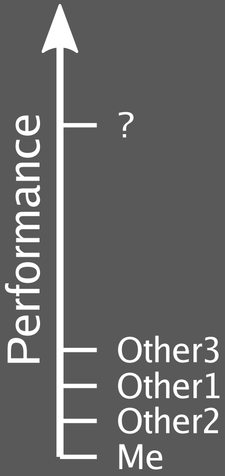
We know where the performance ceiling is



We know where the performance ceiling is



We know where the performance ceiling is



We know where the performance ceiling is

We don't know how well we're doing

We’re good at optimising abstractions

We’re good at optimising abstractions

Most optimisations are ad-hoc and/or unpredictable

We're good at optimising abstractions

Most optimisations are ad-hoc and/or unpredictable

e.g. mono $\rightarrow$ poly $\rightarrow$ megamorphic JS calls

We’re good at optimising abstractions

How to communicate optimisations to users?

We know what the impact of individual features is

We know what the impact of individual features is

What is the effect of e.g. pointer tagging?

We know what the impact of individual features is

What is the effect of e.g. pointer tagging?

GC and register allocation only
fairly deeply studied topics?

We know what the impact of individual features is

Hardware

We know what the impact of individual features is

Hardware: caches

We know what the impact of individual features is

Hardware: caches, predictors

We know what the impact of individual features is

Hardware: caches, predictors, temperature

We know what the impact of individual features is

Hardware: caches, predictors, temperature, etc.

We know what the impact of individual features is

Hardware: caches, predictors, temperature, etc.

OS

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches, etc.

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches, etc.
VM	

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches, etc.
VM:	compilation heuristics

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches, etc.
VM:	compilation heuristics, GC heuristics

We know what the impact of individual features is

Hardware:	caches, predictors, temperature, etc.
OS:	other processes, context switches, etc.
VM:	compilation heuristics, GC heuristics, etc.

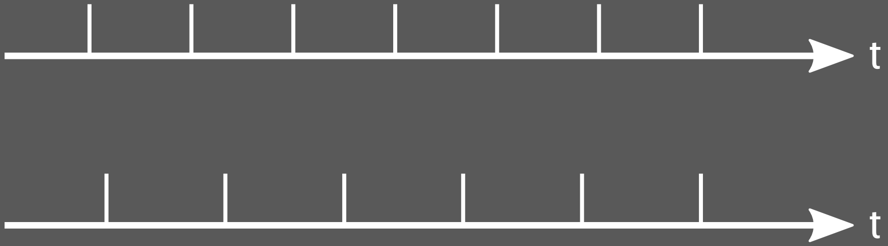
We know how features interact

Performance non-determinism is rife

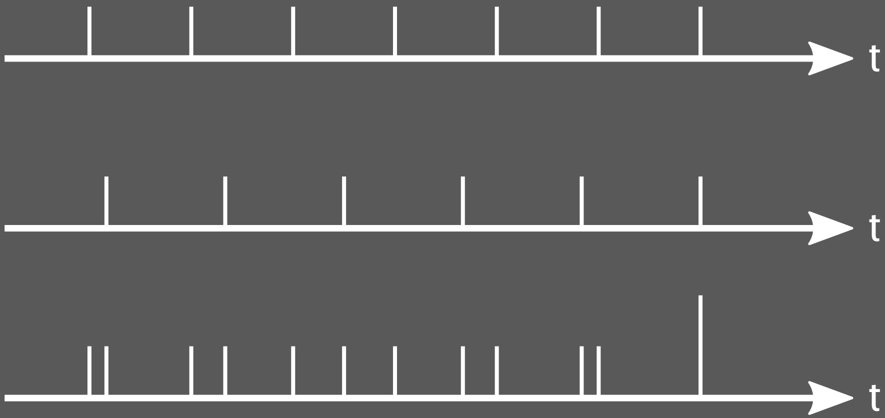
Performance non-determinism

Performance non-determinism

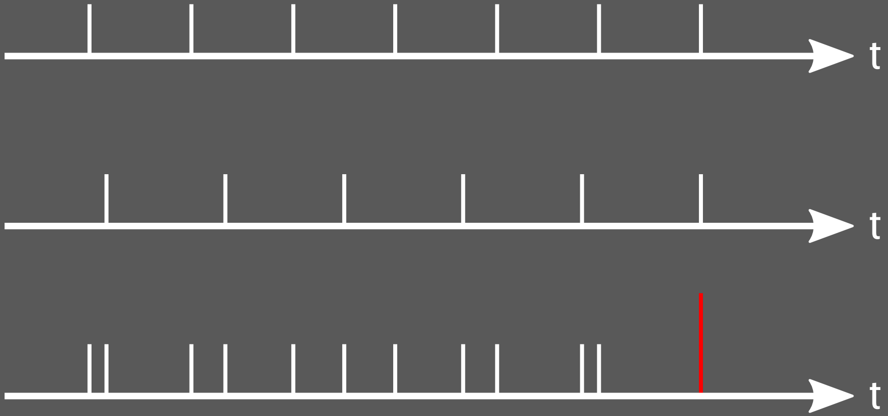
Performance non-determinism



Performance non-determinism



Performance non-determinism



Performance non-determinism

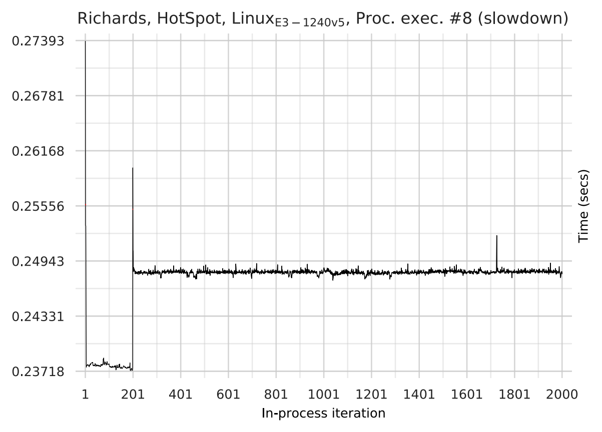
Solution to performance non-determinism

Performance non-determinism

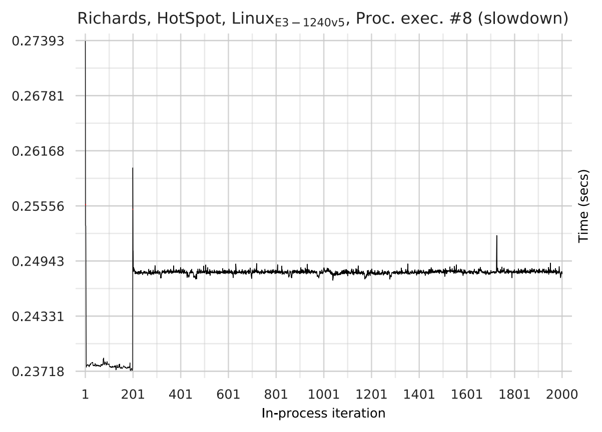
Solution to performance non-determinism:
non-determinism?

We understand how large systems perform

We understand how large systems perform



We understand how large systems perform



Microbenchmarks behave poorly

We understand how large systems perform

But that doesn’t affect real programs

We understand how large systems perform

How convenient!

We understand how large systems perform

What about compositionality?

Multi-language VMs

Multi-language VMs

VMs are expensive to create

Multi-language VMs

VMs are expensive to create

Why not reuse that hard work?

Multi-language VMs

VMs are expensive to create

Why not reuse that hard work?

CPython vs. Jython parable

Semantic mismatch

Semantic mismatch

Semantic mismatch

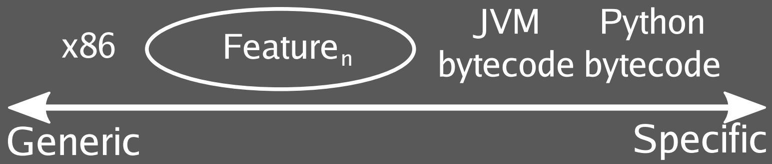
Semantic mismatch

Semantic mismatch

Semantic mismatch

Semantic mismatch

Semantic mismatch



Semantic mismatch

Semantic mismatch

Semantic mismatch

One solution: language design tweaks

Semantic mismatch

WASM will not solve the semantic mismatch

Semantic mismatch

WASM will not solve the semantic mismatch

Meta-VMs suffer much less

My benchmark suite is good, yours is bad

My benchmark suite is good, yours is bad

Fix memory leak in pdfjs.js. #42

ltratt wants to merge 2 commits into `chromium:master` from `ltratt:master`

Conversation 7 | Commits 2 | Checks 0 | Files changed 1

ltratt commented on Oct 2, 2016 • edited

A large amount of data is pushed into the global variable `canvas_logs` which isn't cleared in runPdfJS. On each iteration the list grows, eventually significantly so.

On a Linux machine with a recent-ish V8, it manages 2777 iterations before an allocation fails (at which point it's allocated over 2GiB of virtual memory, and used about 1.4Gib) and V8 crashes (`Fatal error in CALL_AND_RETRY_LAST`).

My benchmark suite is good, yours is bad

Fix memory leak in pdfjs.js. #42

ltratt wants to merge 2 commits into `chromium:master` from `ltratt:master`

Conversation 7 | Commits 2 | Checks 0 | Files changed 1

ltratt commented on Oct 2, 2016 • edited

A large amount of data is pushed into the global variable `canvas_logs` which isn't cleared in runPdfJS. On each iteration the list grows, eventually significantly so.

On a Linux machine with a recent-ish V8, it manages 2777 iterations before an allocation fails (at which point it's allocated over 2GiB of virtual memory, and used about 1.4Gib) and V8 crashes (`Fatal error in CALL_AND_RETRY_LAST`).

.

.

natorion commented on Feb 7, 2018 • edited **Member**

It won't be merged. Octane is retired and no longer maintained. Sorry for the long communication cycle.

natorion closed this on Feb 7, 2018

My benchmark suite is good, yours is bad

A year with Spectre: a V8 perspective

Published 23 April 2019 · tagged with security

On January 3, 2018, Google Project Zero and others disclosed the first three of a new class of vulnerabilities that affect CPUs that perform speculative execution, dubbed Spectre and Meltdown. Using the speculative execution mechanisms of CPUs, an attacker could temporarily bypass both implicit and explicit safety checks in code that prevent programs from reading unauthorized data in memory. While processor speculation was designed to be a microarchitectural detail, invisible at the architectural level, carefully crafted programs could read unauthorized information in speculation and disclose it through side channels such as the execution time of a program fragment.

.

.

.

We have experimented with (1) by inserting the recommended speculation barrier instructions, such as Intel's `LFENCE`, on every critical conditional branch, and by using retpolines for indirect branches. Unfortunately, such heavy-handed mitigations greatly reduce performance (2–3× slowdown on the Octane benchmark). Instead, we chose approach (2), inserting mitigation sequences that prevent reading secret data due to mis-speculation. Let us illustrate the technique on the following code snippet:

My benchmark suite is good, yours is bad

A year with Spectre: a V8 perspective

Published 23 April 2019 · tagged with security

On January 3, 2018, Google Project Zero and others disclosed the first three of a new class of vulnerabilities that affect CPUs that perform speculative execution, dubbed Spectre and Meltdown. Using the speculative execution mechanisms of CPUs, an attacker could temporarily bypass both implicit and explicit safety checks in code that prevent programs from reading unauthorized data in memory. While processor speculation was designed to be a microarchitectural detail, invisible at the architectural level, carefully crafted programs could read unauthorized information in speculation and disclose it through side channels such as the execution time of a program fragment.

.

.

.

We have experimented with (1) by inserting the recommended speculation barrier instructions, such as Intel's `LFENCE`, on every critical conditional branch, and by using retpolines for indirect branches. Unfortunately, such heavy-handed mitigations greatly reduce performance (2–3× slowdown on the Octane benchmark). Instead, we chose approach (2), inserting mitigation sequences that prevent reading secret data due to mis-speculation. Let us illustrate the technique on the following code snippet:

My benchmark suite is good, yours is bad

My benchmark suite is good, yours is bad

Benchmark suites a finite representation of infinite behaviour

My benchmark suite is good, yours is bad

Benchmark suites a finite representation of infinite behaviour

All benchmark suites are imperfect

My benchmark suite is good, yours is bad

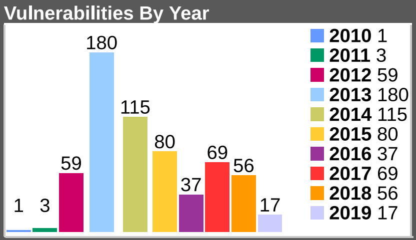
Benchmark suites a finite representation of infinite behaviour

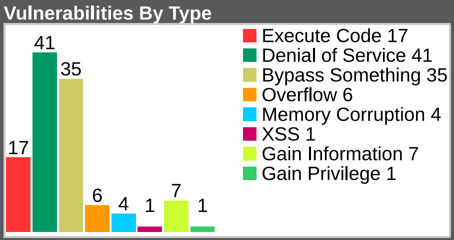
All benchmark suites are imperfect

We need more and more benchmarks!

Managed languages are safe

Managed languages are safe





Source: https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93

Managed languages are safe

C/C++ aren’t very safe

Managed languages are safe

C/C++ aren’t very safe

And what about JITted code?

Managed languages are safe

C/C++ aren’t very safe

And what about JITted code?

Prediction: VM security apocalypse is possible

We’re stuck with C/C++

We’re stuck with C/C++

What about Rust?

We’re stuck with C/C++

What about Rust?

Not an obvious fit for VMs

We’re stuck with C/C++

What about Rust?

Not an obvious fit for VMs

Can we make it so?

Thin pointers for dynamic dispatch

VMs use dynamic dispatch extensively

Thin pointers for dynamic dispatch

VMs use dynamic dispatch extensively

```
use std::mem::size_of;

trait T { }

fn main() {
  assert_eq!(size_of::<&bool>(), size_of::<&u128>());
  assert_eq!(size_of::<&bool>(), size_of::<usize>());
  assert_eq!(size_of::<&dyn T>(), size_of::<usize>() * 2);
}
```

Thin pointers for dynamic dispatch

```
let x: &dyn T = ...;
let (ptr, vtable) = unsafe {
  mem::transmute<_, (*mut u8, *mut u8)>(x)
};
```

Thin pointers for dynamic dispatch

```
#[repr(C)]
struct ThinPtr { objptr: *mut u8 }

impl ThinPtr {
  fn new<U>(o: U) -> ThinPtr
    where *const U: CoerceUnsized<*const (dyn T + 'static)>,
          U: T + 'static
  {
    let (dataptr, vtable) = unsafe { mem::transmute<_, (*mut u8, *mut u8)>(x) };
    let objptr = malloc(size_of::<*mut u8>() + size_of::<U>());
    unsafe { ptr::write(objptr, &vtable, size_of::<*mut u8>()) };
    unsafe { ptr::write(objptr + 1, ptr, size_of::<U>()) };
    ThinPtr { objptr }
  }
}

impl Deref for ThinPtr {
  type Target = dyn T;
  fn deref(&self) -> &(dyn T + 'static) {
    let vtable = unsafe { ptr::read(objptr, size_of::<*mut u8>()) };
    unsafe { transmute::<(*const _, *const _), _>((self.objptr + 1, vtable)) }
  }
}
```

Thin pointers for dynamic dispatch

```
#[repr(C)]
struct ThinPtr { objptr: *mut u8 }

impl ThinPtr {
  fn try_downcast<U: T>(&self) -> Option<&U> {
    let t_vtable = unsafe {
      transmute::<&dyn T, (*mut u8, *mut u8)>(ptr::null() as *const U) };
    let vtable = unsafe { ptr::read(objptr, size_of::<*mut u8>()) };
    if vtable == t_vtable {
      Some(unsafe { &* (self.objptr + 1) as *const U })
    } else {
      None
    }
  }
}
```

Thin pointers for dynamic dispatch

```
#[narrowable_abgc(ThinObj)]
trait Obj { }

struct VMInt { x: u64 }

impl Obj for VMInt { }

fn f(v: ThinObj) {
  if let Some(o) = v.try_downcast::<VMInt>() {
    println!(o.x);
  }
}
```

Can we use Rust for VMs?

Can we use Rust for VMs?

So far, so good

The security landscape is changing

The security landscape is changing

CHERI:

The security landscape is changing

CHERI: capabilities in 128-bit pointers

We’re good at explaining what we do

We’re good at explaining what we do

IronPython; Jython; Nuitka; Psyco; PyPy; Pyston; Shed Skin; Stackless; Starkiller; TrufflePython; Unladen Swallow; WPython; Zippy

We’re good at explaining what we do

IronPython; Jython; Nuitka; Psyco; PyPy; Pyston; Shed Skin; Stackless; Starkiller; TrufflePython; Unladen Swallow; WPython; Zippy

e.g. compiling to LLVM fails every time…

We’re good at explaining what we do

We often pretend trade-offs don’t exist

We’re good at explaining what we do

We often pretend trade-offs don’t exist

Huge burden for newcomers to the field

Where can we go next?

Where can we go next?

We understand less than we should

Where can we go next?

Clear problems

Where can we go next?

Opportunities!

Where can we go next?

Opportunities!

Hardware meta-tracing

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
  elif instr == INSTR_IF:
    result = stack.pop()
    if result == True:
      program_counter += 1
    else:
      program_counter +=
        read_jump_if_instruction()
  elif instr == INSTR_ADD:
    lhs = stack.pop()
    rhs = stack.pop()
    if isinstance(lhs, int)
     and isinstance(rhs, int):
      stack.push(lhs + rhs)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter	User program (lang *FL*)

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

Meta-tracing JITs

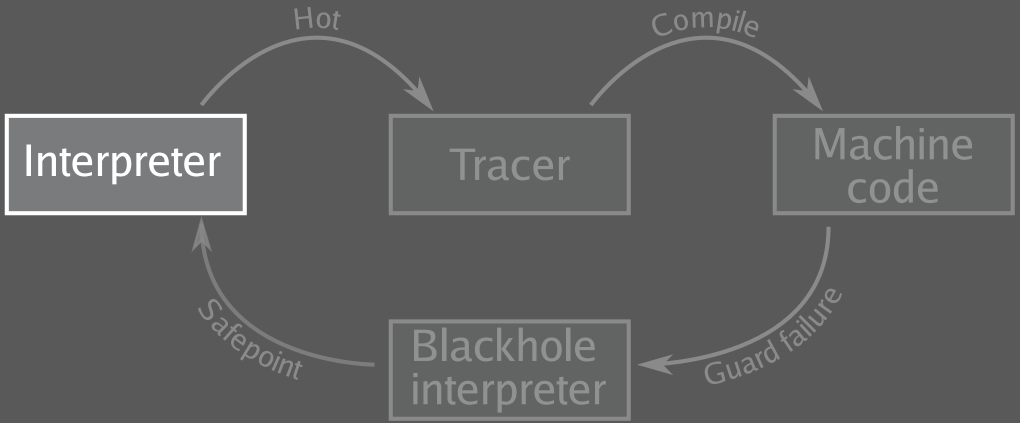
FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

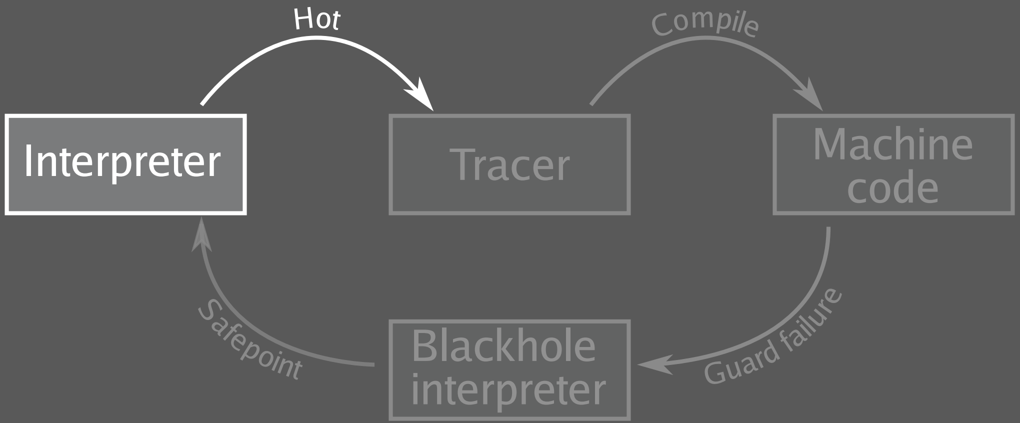
Initial trace

```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
```

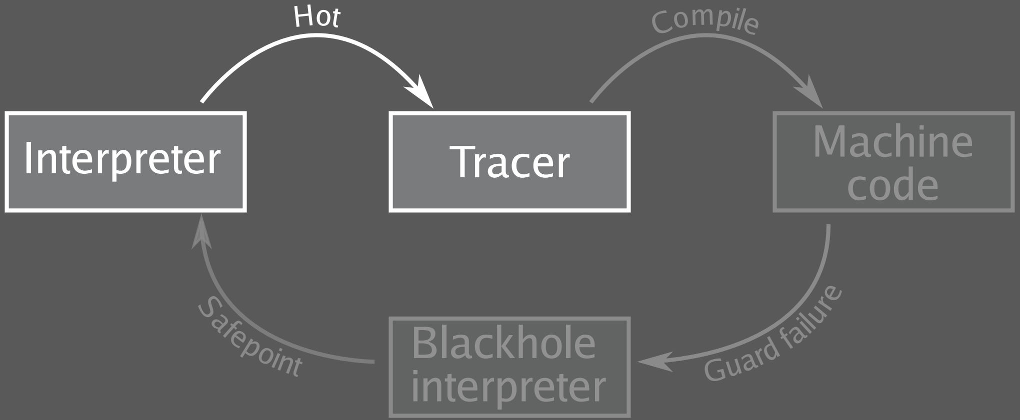
Meta-tracer states



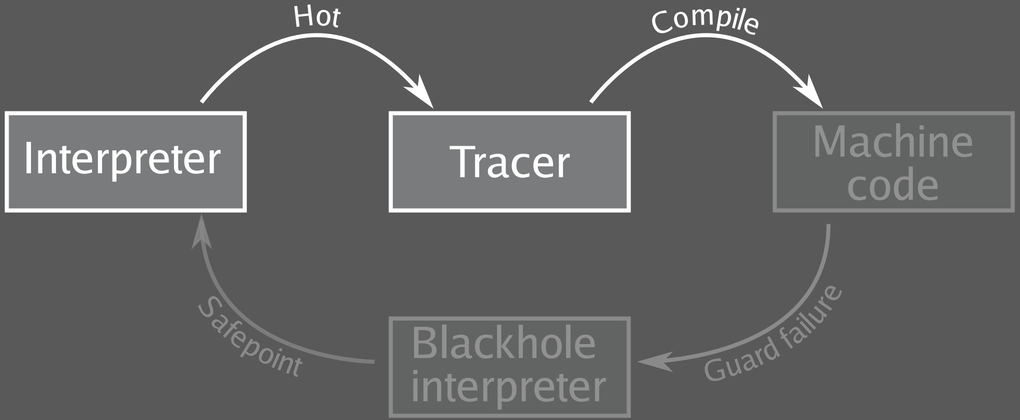
Meta-tracer states



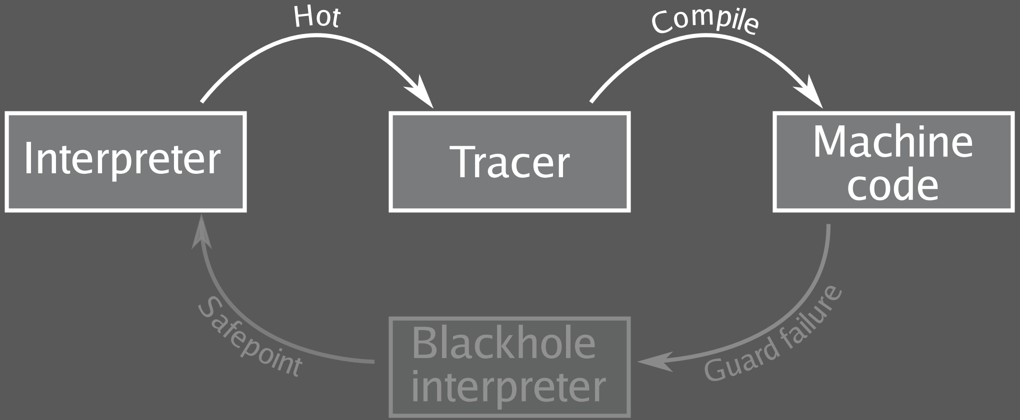
Meta-tracer states



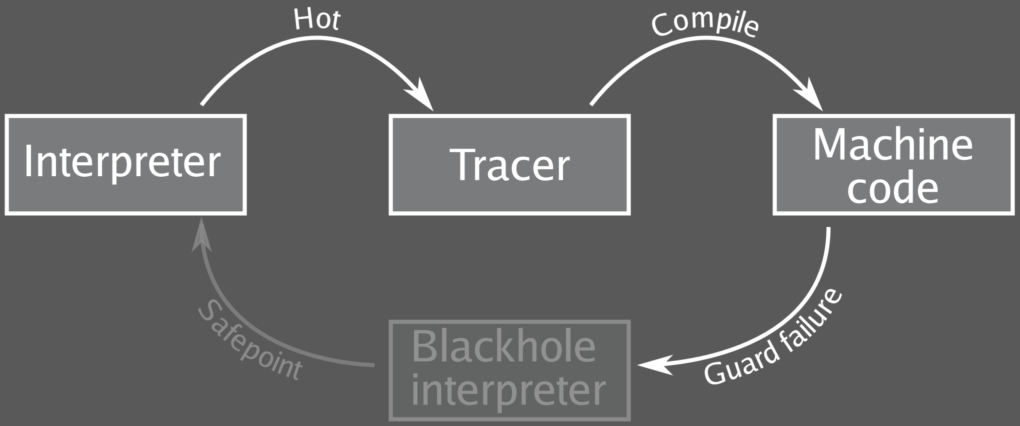
Meta-tracer states



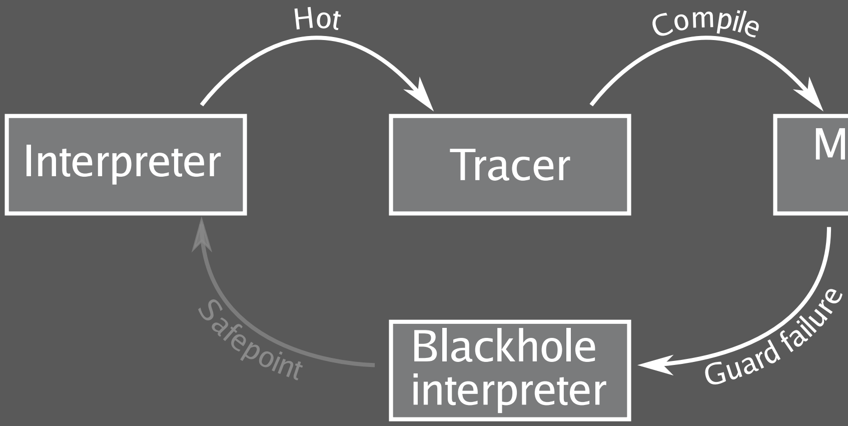
Meta-tracer states



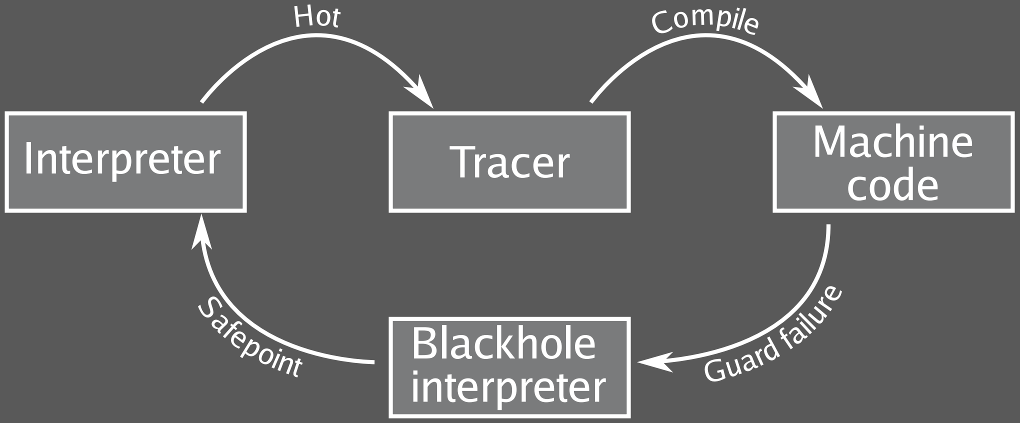
Meta-tracer states



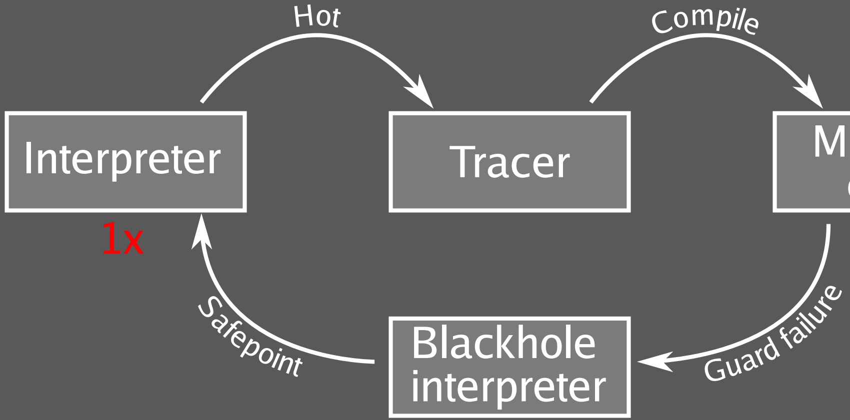
Meta-tracer states



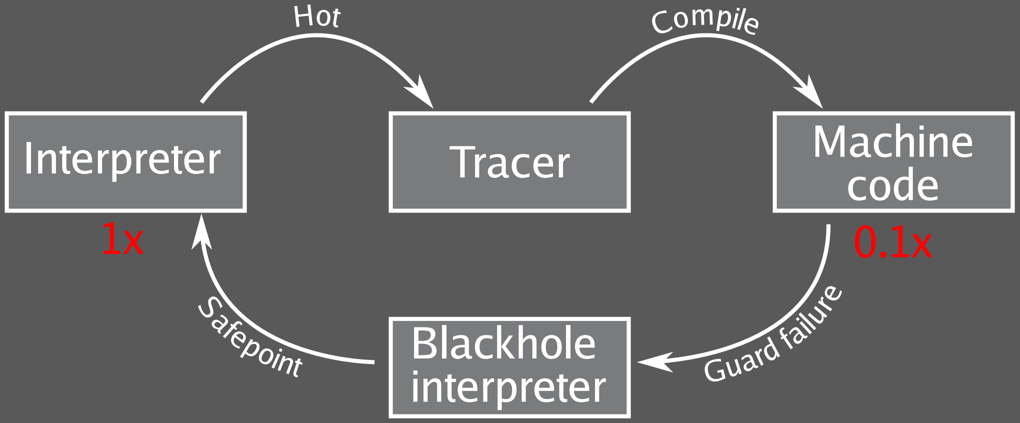
Meta-tracer states



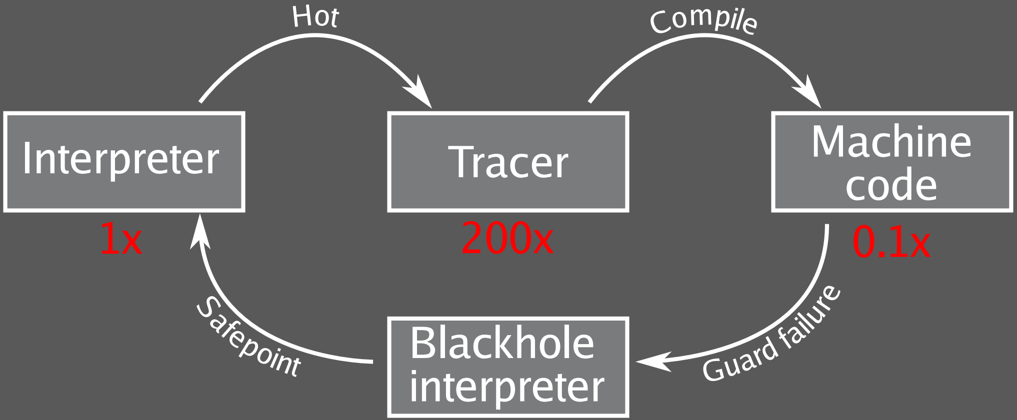
Meta-tracer performance (now)



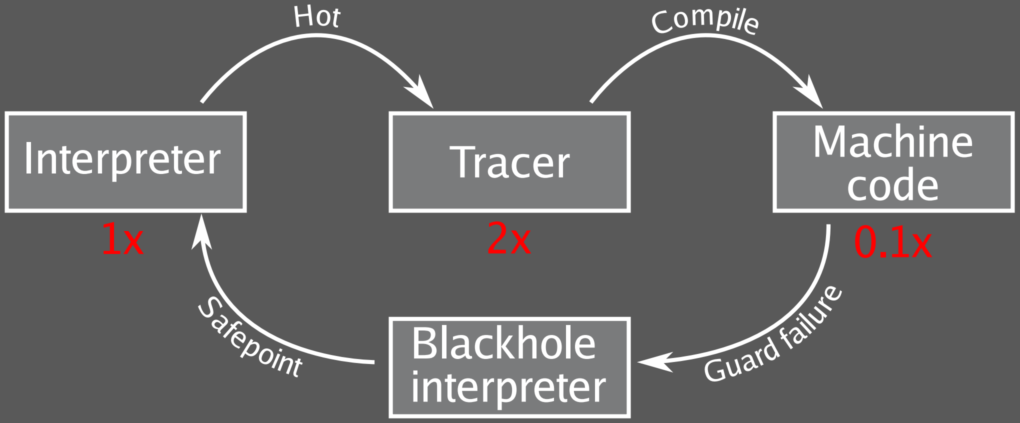
Meta-tracer performance (now)



Meta-tracer performance (now)



Meta-tracer performance (Rust + PT)



Status:

Status: hello world

Thanks

- EPSRC: *COOLER* and *Lecture*.
- Oracle.
- Cloudflare.

Thanks for listening