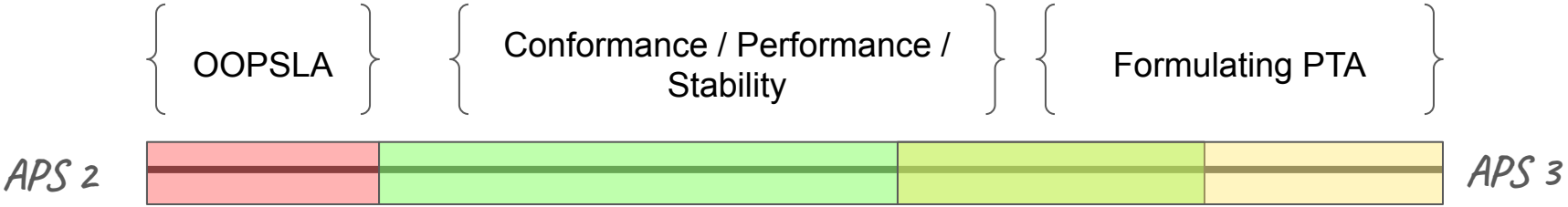
Static Analysis of **JavaScript** Programs

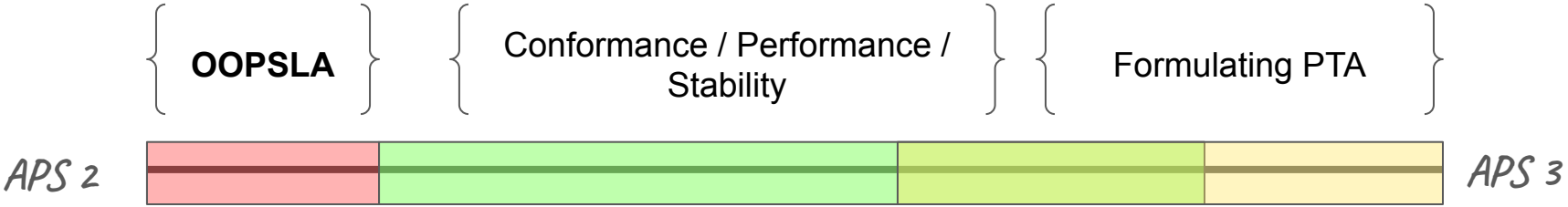
PLATO

Advisor ↠ **Dr Manas Thakur**
Presenter ↠ **Meetesh Kalpesh Mehta (23d0361)**, PhD3

Timeline Of Major Events

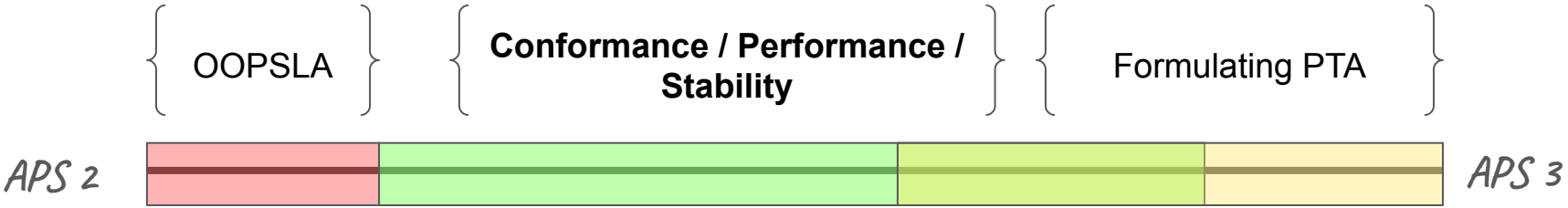


Timeline Of Major Events



[OOPSLA 2026] IRIDIUM: A Framework for Statically Optimizing JavaScript Programs

Timeline Of Major Events

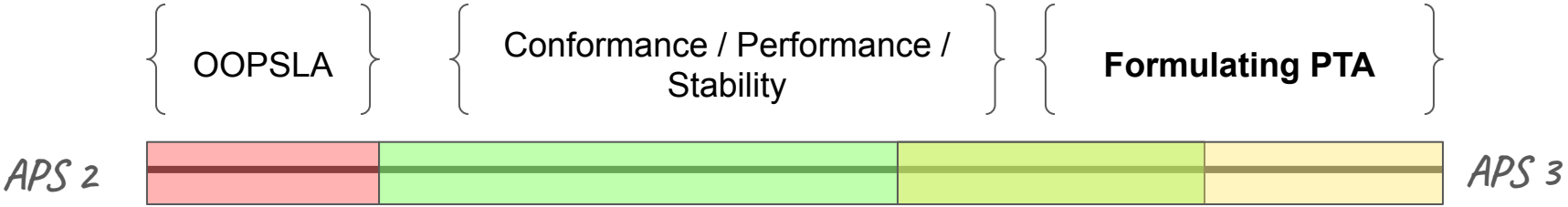


~60% (During APS2) **-> 99.78%** positive compliance against quickjs

Kraken **>5 mins -> 250ms**

Larger set of programs

Timeline Of Major Events

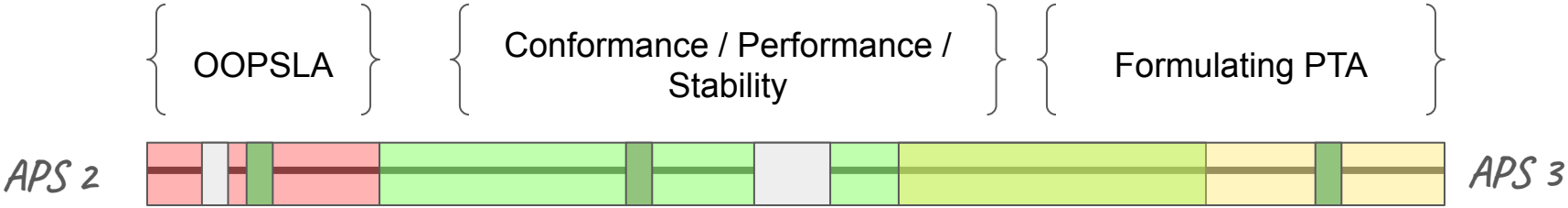


~ **Existing approaches + background survey**

Applications in security, repository of programs

~ **Prakriti: PTA engine for IRIDIUM**

Timeline Of Major Events



Coffre QEMU drivers / Linux Kernel / Security Architecture
Closed Source @ IITB

LMA utility: Ubuntu PPA, IRI Specification, Potential users of IRIDIUM
Open Source

Important Questions

1. **[LAST YEAR]** Why not derive an IR from **existing bytecode** implementations?
2. What is **runtime barrier**? **What is the semantics**? Why do we need them?
3. What's the **difference between 3JS and IRI**? Why are they needed?

JavaScript is easy!

test.mjs

```
function fun1() {
  fun1 = 12;
}

console.log(fun1());

let fun2 = function fun2() {
  fun2 = 12;
}

console.log(fun2());
```

Are you sure about that?

test.mjs

```
function fun1() {
  fun1 = 12;
}

console.log(fun1());

let fun2 = function fun2() {
  fun2 = 12;
}

console.log(fun2());
```

Output?

test.mjs

```
}

console.log(fun2());
```

ERROR

Output?

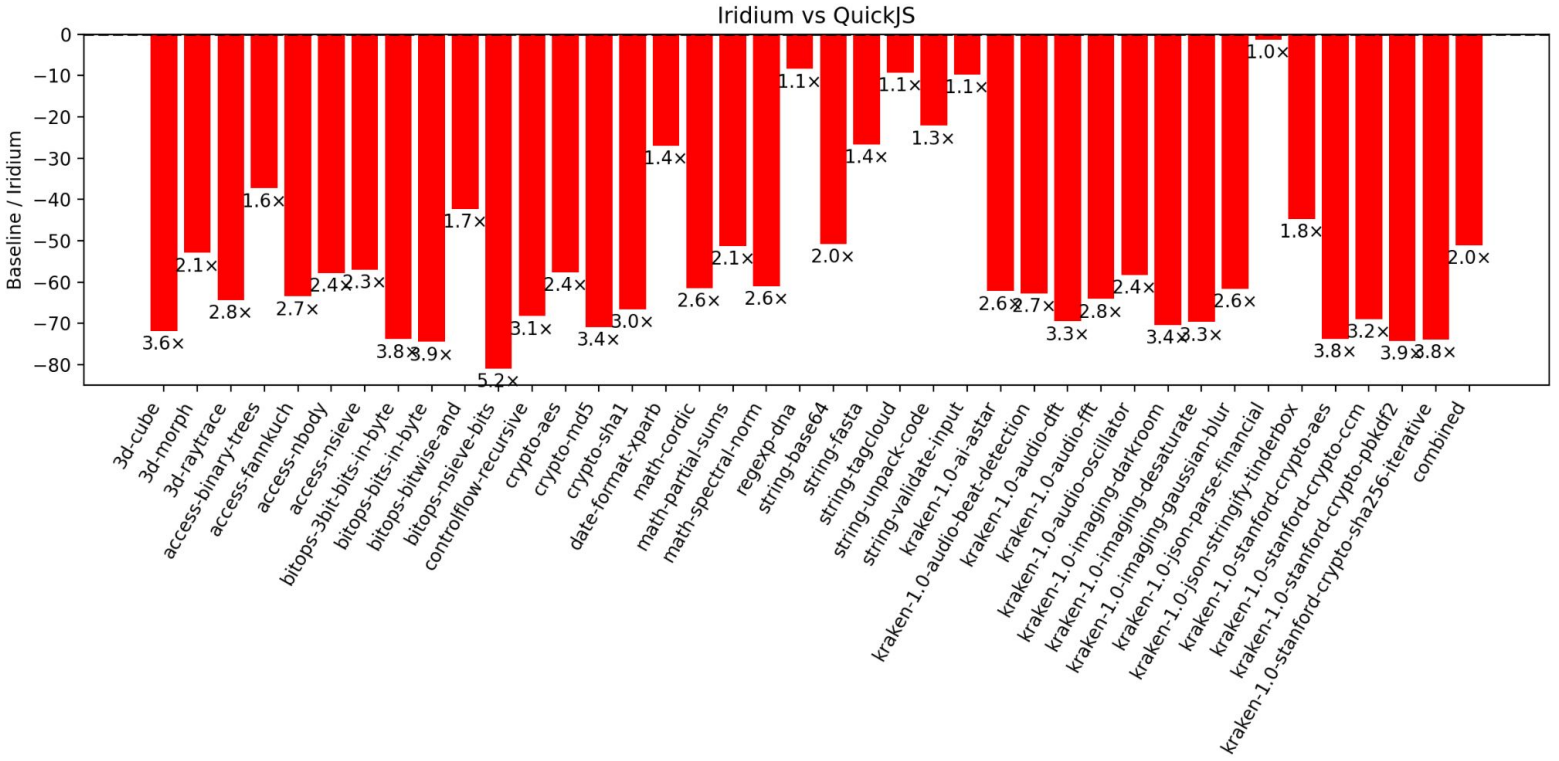
test.mjs

```
}

console.log(fun2());
```

TypeError: Assignment to constant variable.

OOPSLA

OOPSLA **[APS 2]**


Iridium vs QuickJS
Baseline / Iridium
0
−10
−20
−30
−40
−50
−60
−70
−80
3.6×
2.1×
2.8×
1.6×
2.7×
2.4×
2.3×
3.8×
3.9×
1.7×
5.2×
3.1×
2.4×
3.4×
3.0×
1.4×
2.6×
2.1×
2.6×
1.1×
2.0×
1.4×
1.1×
1.3×
1.1×
2.6×
2.7×
3.3×
2.8×
2.4×
3.4×
3.3×
2.6×
1.0×
1.8×
3.8×
3.2×
3.9×
3.8×
2.0×
3d-cube
3d-morph
3d-raytrace
access-binary-trees
access-fannkuch
access-nbody
access-nsieve
bitops-3bit-bits-in-byte
bitops-bits-in-byte
bitops-bitwise-and
bitops-nsieve-bits
controlflow-recursive
crypto-aes
crypto-md5
crypto-sha1
date-format-xparb
math-cordic
math-partial-sums
math-spectral-norm
regexp-dna
string-base64
string-fasta
string-tagcloud
string-unpack-code
string-validate-input
kraken-1.0-ai-astar
kraken-1.0-audio-beat-detection
kraken-1.0-audio-dft
kraken-1.0-audio-fft
kraken-1.0-audio-oscillator
kraken-1.0-imaging-darkroom
kraken-1.0-imaging-desaturate
kraken-1.0-imaging-gaussian-blur
kraken-1.0-json-parse-financial
kraken-1.0-json-stringify-tinderbox
kraken-1.0-stanford-crypto-aes
kraken-1.0-stanford-crypto-ccm
kraken-1.0-stanford-crypto-pbkdf2
kraken-1.0-stanford-crypto-sha256-iterative
combined

OOPSLA: **Optimizations**

Traditional optimizations are **possible** to implement, but **ineffective**.

- *‘Enabler’* passes are needed to make them useful.
- **?** → Liveness → DCE
-

OOPSLA: **O**

Traditional opti **neffective**.

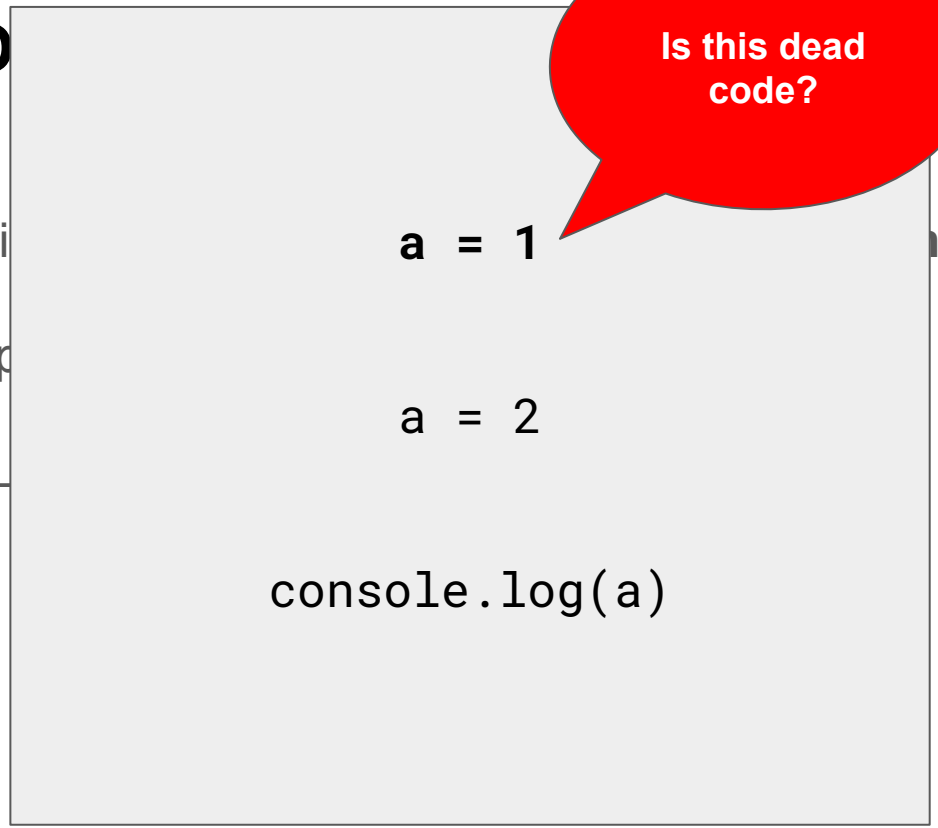
- *'Enabler'* p
- **?** → L
-

```
a = 1

a = 2

console.log(a)
```

OOPSLA: **O**

Traditional opti **neffective**.

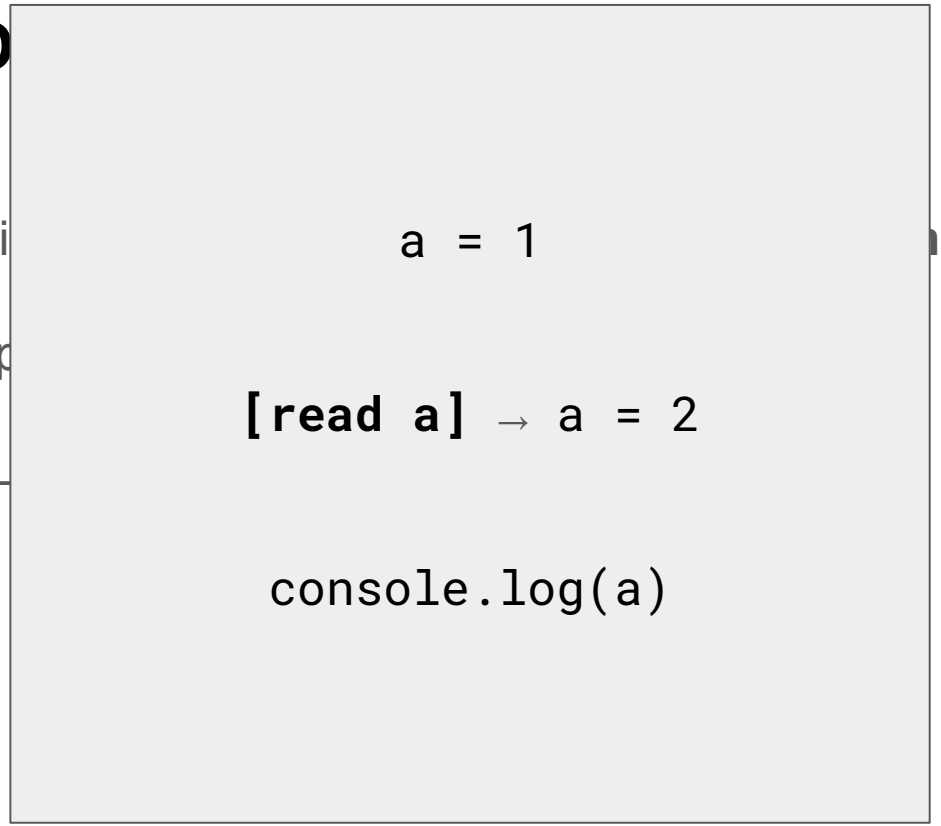
- *'Enabler'* p
- **?** → L
-



OOPSLA: **O**

Traditional opti **neffective**.

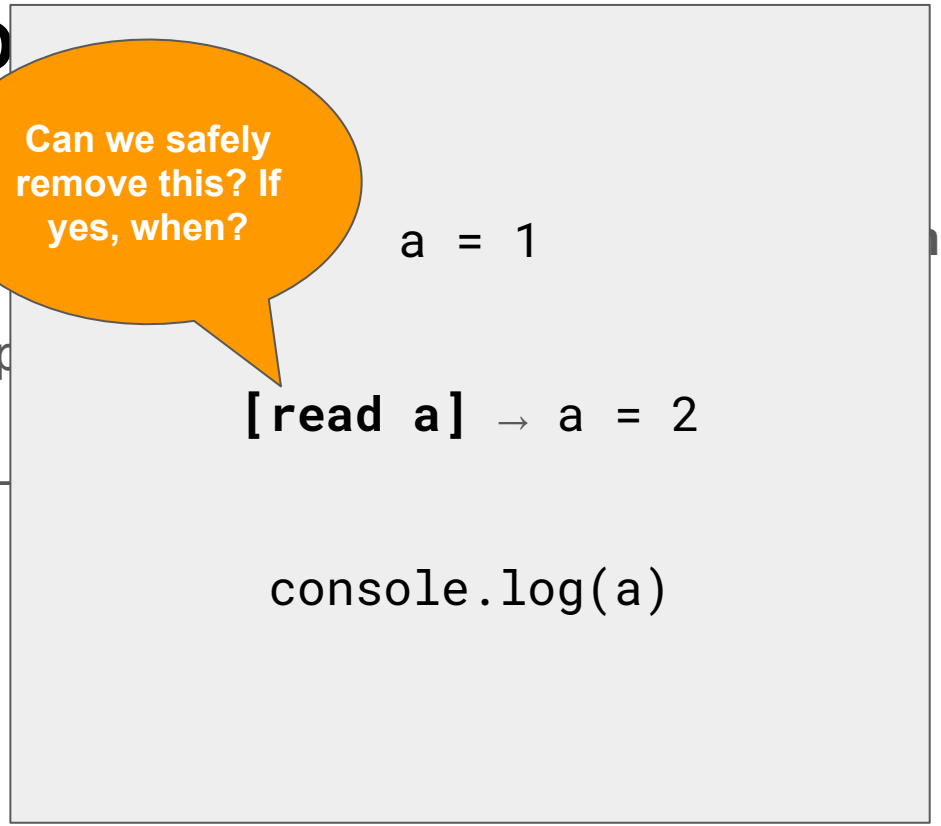
- *'Enabler'* p
- **?** → L
-



OOPSLA: **O**

Traditional o **ineffective**.

- *'Enabler'* p
- **?** → L
-



OOPSLA: **Optimizations**

Traditional optimizations are **possible** to implement, but **ineffective**.

- *'Enabler'* passes are needed to make them useful.
- **Barriers?** → Liveness → DCE (Analysis pipeline)
  - ?

OOPSLA: **Optimizations**

Traditional optimizations are **possible** to implement, but **ineffective**.

- *'Enabler'* passes are needed to make them useful.
- **TDZA** → **Barrier Elimination** → Liveness → DCE (Analysis pipeline)
  - Predictable stack behaviour → Reliable flow functions

OOPSLA: Enabler 1: **TDZA**

Source Code

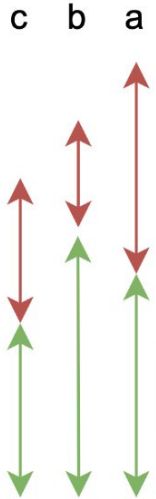
```
let b = () => a * 2;
let a = 12
let c = () => {
  a = 33;
}
c()
b()
a = 13
```

OOPSLA: Enabler 1: **TDZA**

Source Code

```
let b = () => a * 2;
let a = 12
let c = () => {
  a = 33;
}
c()
b()
a = 13
```

TDZ



Initial Code

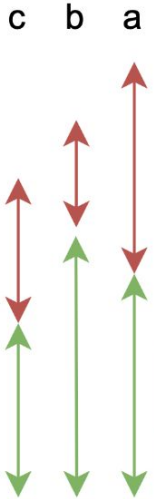
```
a = Λ
b = Λ
c = Λ
b = Lambda() => .a * 2
a = 12
c = Lambda() => .a = 33
.c()
.b()
a .= 13
```

OOPSLA: Enabler 1: **TDZA**

Source Code

```
let b = () => a * 2;
let a = 12
let c = () => {
  a = 33;
}
c()
b()
a = 13
```

TDZ



Initial Code

```
a = Λ
b = Λ
c = Λ
b = Lambda() => .a * 2
a = 12
c = Lambda() => .a = 33
.c()
.b()
a .= 13
```

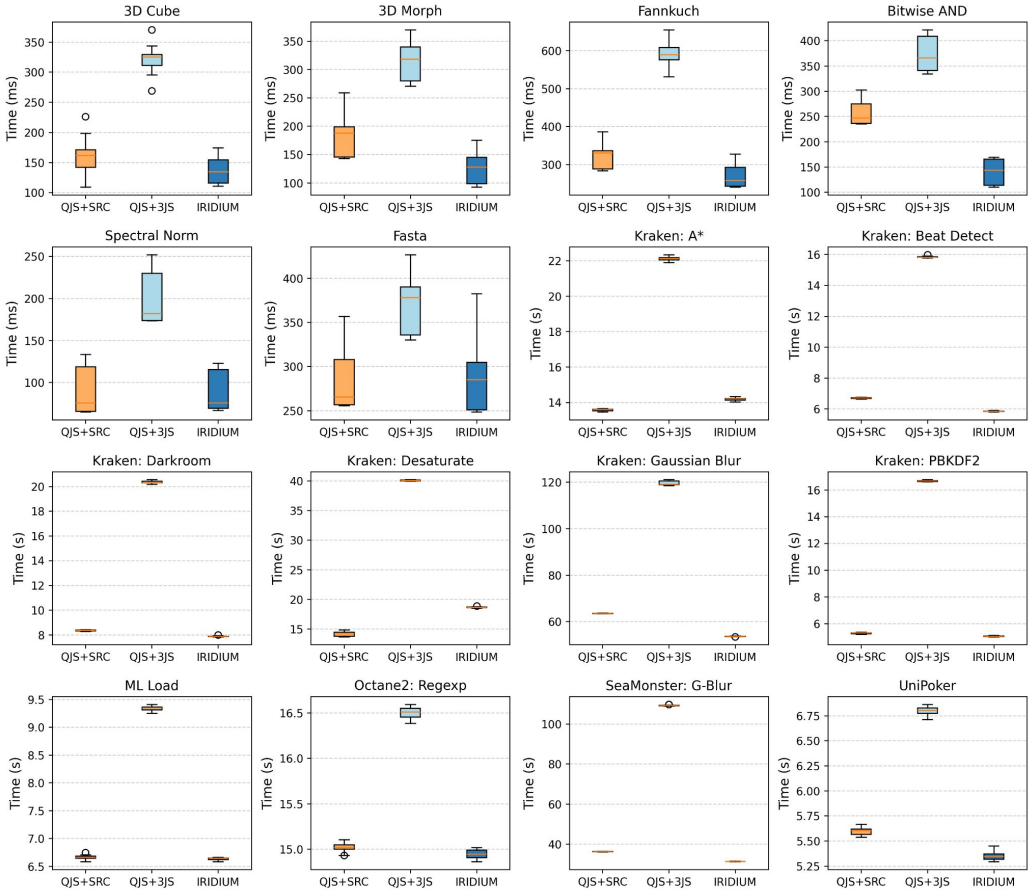
Barriers Removed

```
a = Λ
b = Λ
c = Λ
b = Lambda() => .a * 2
a = 12
c = Lambda() =>  a = 33
c()
b()
a = 13
```

OOPSLA **[After OPTS]**

18% faster

2.5X faster over transpilation



OOPSLA: Enabler 2: **Effect Propagation**

$$\text{LatticeVal} = \{\ \bot,\ \langle \text{store}, \text{effect} \rangle,\ \top\ \}$$

$$\text{merge}(v_1, v_2) = \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{if } v_2 = \bot \\ v_1 & \text{if } v_1 = v_2 \\ \top & \text{otherwise} \end{cases}$$

$$\text{In}(n) = \begin{cases} \bot & \text{if } n = \text{StartBB} \\ \text{merge}\big(\{\text{Out}(p)\}_{p \in \text{Preds}(n)}\big) & \text{otherwise} \end{cases}$$

$$\text{Out}(n) = \begin{cases} \langle \text{store}(n), \text{effect}(n) \rangle & \text{if } n \text{ is } \mathsf{EnvWrite} \wedge n \notin \mathit{Blacklist} \wedge \text{rhs}(n) \text{ may be effectful} \\ \top & \text{if } n \text{ is } \mathsf{EnvWrite} \vee n \text{ is effectful} \\ \text{In}(n) & \text{otherwise} \end{cases}$$

OOPSLA: Enabler 2: **Effect Propagation**

```
a = f1()
b = f2()
log(a*b)
log(a)
```

Iteration 1	
Liveness (IN)	**Effect Prop (OUT)**
{ }	{ a → f1() }
{ a }	{ b → f2() }
{ a , b }	{ }
{ a }	{ }

OOPSLA: Enabler 2: **Effect Propagation**

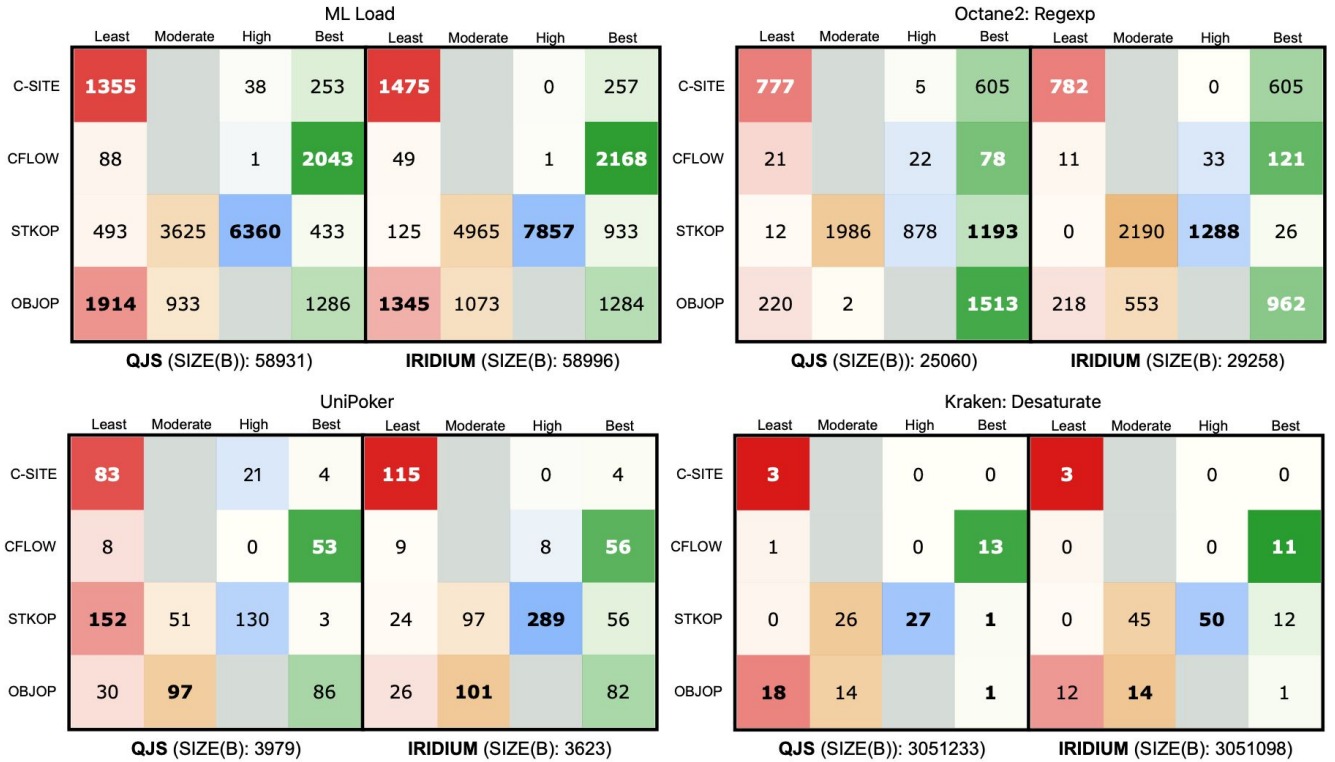
```
a = f1()
b = f2()
log(a*b)
log(a)
```

Iteration 1	
Liveness (IN)	**Effect Prop (OUT)**
{ }	{ a → f1() }
{ a }	{ b → f2() }
{ a , b }	{ }
{ a }	{ }

```
a = f1()
log(a*f2())
log(a)
```

Iteration 2	
Liveness (IN)	**Effect Prop (OUT)**
{ }	{ a → f1() }
{ a }	{ }
{ a }	{ }

OOPSLA: Instruction Selection Quality

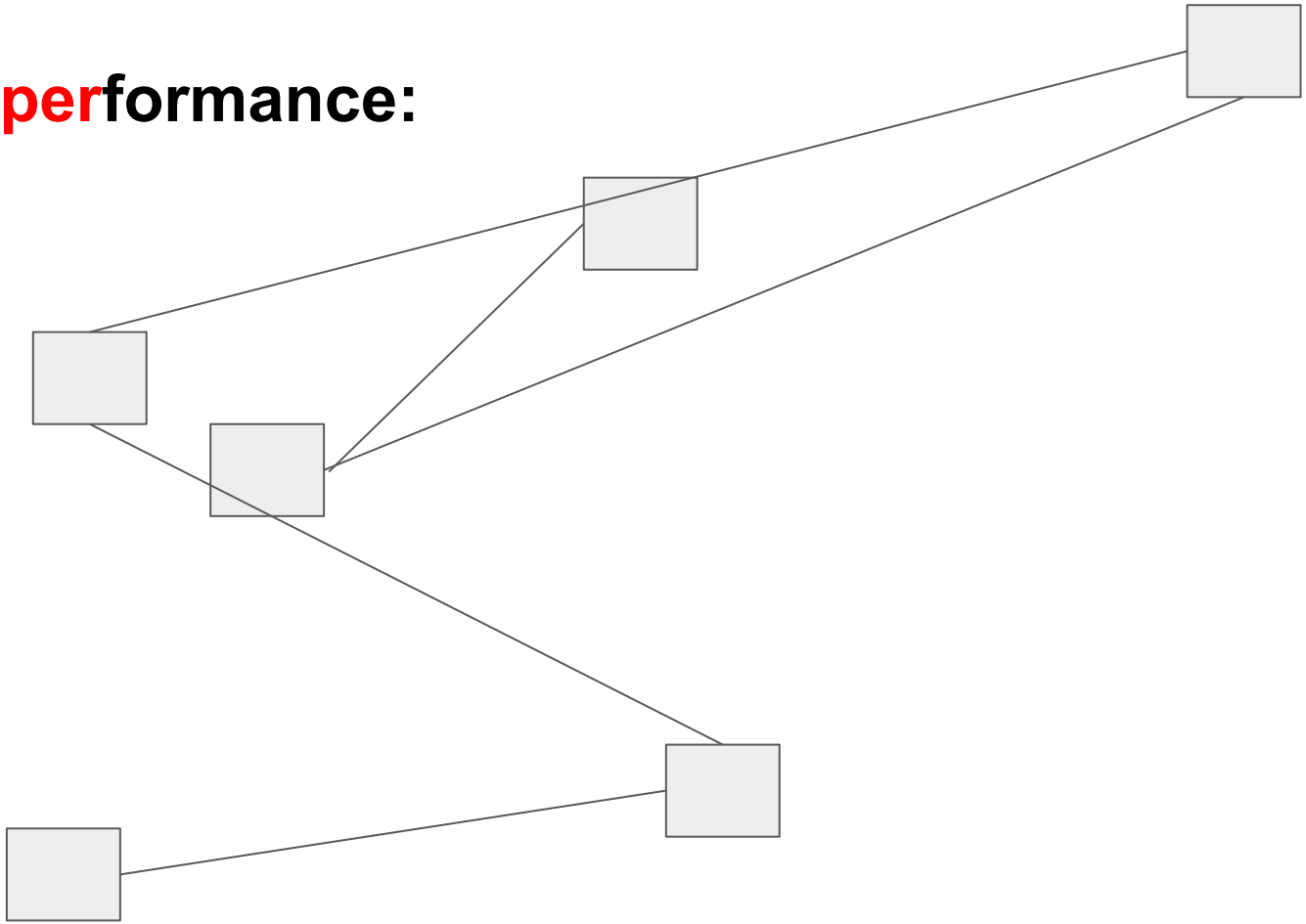


Per

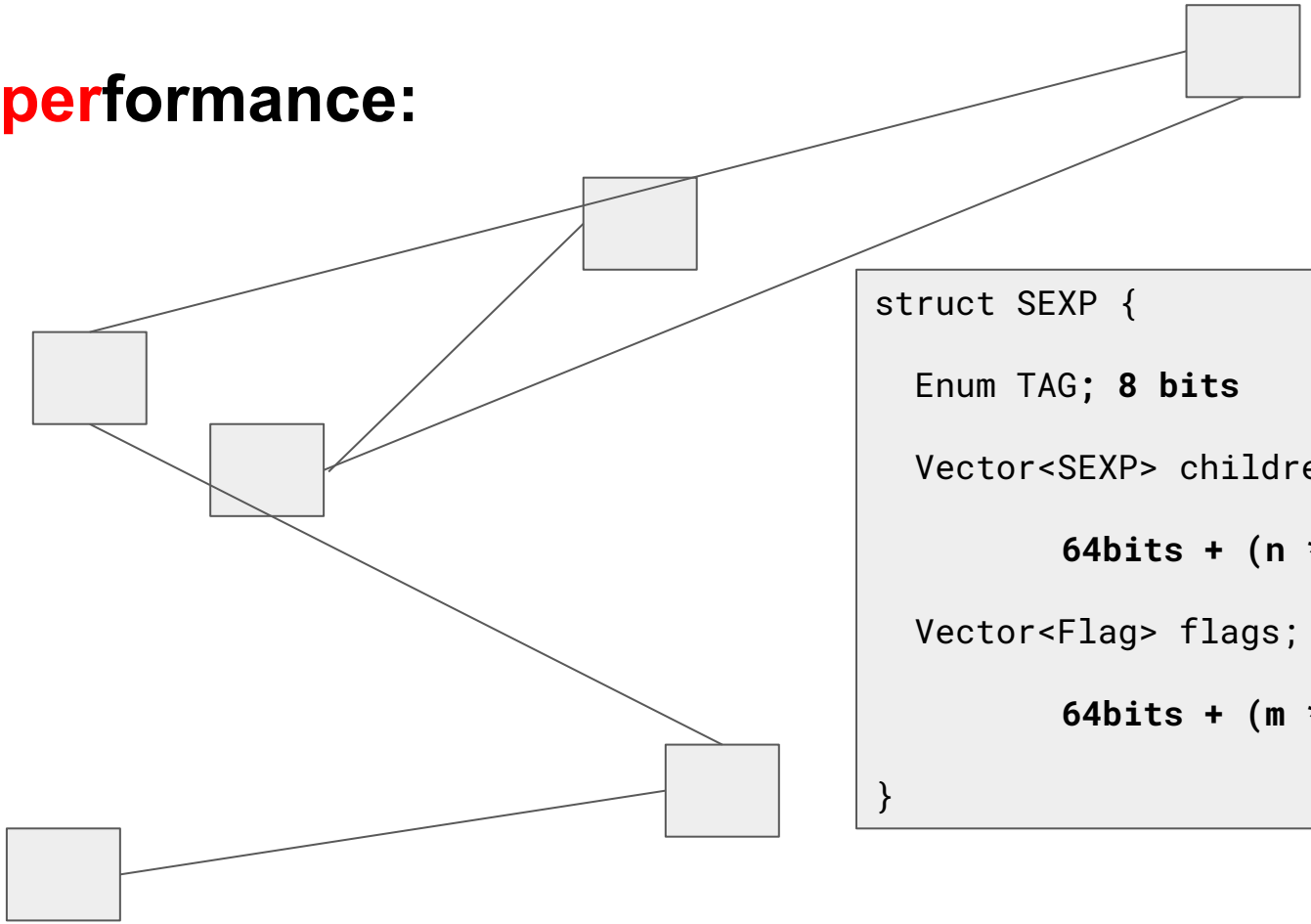
formance

Con

performance:

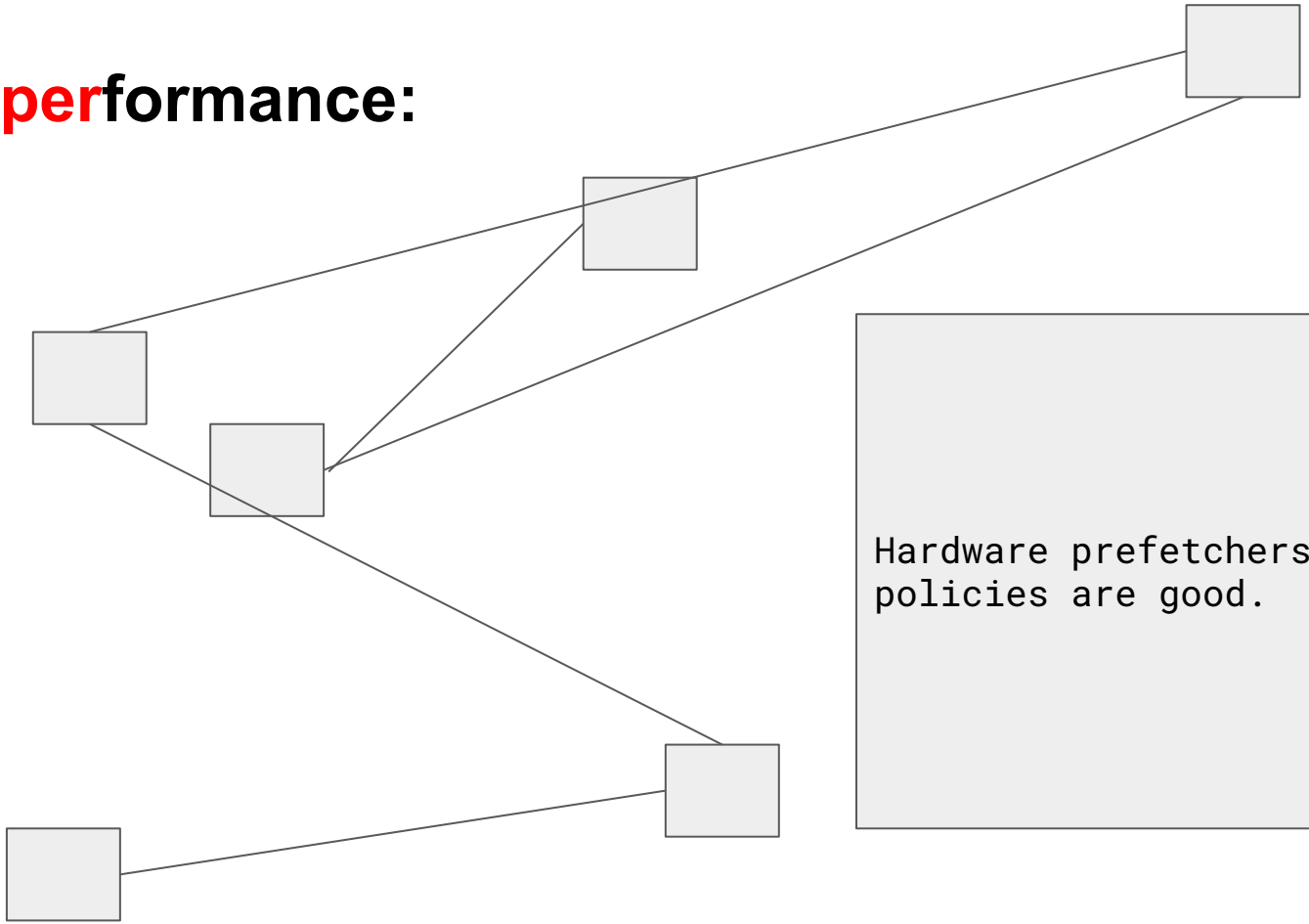


performance:



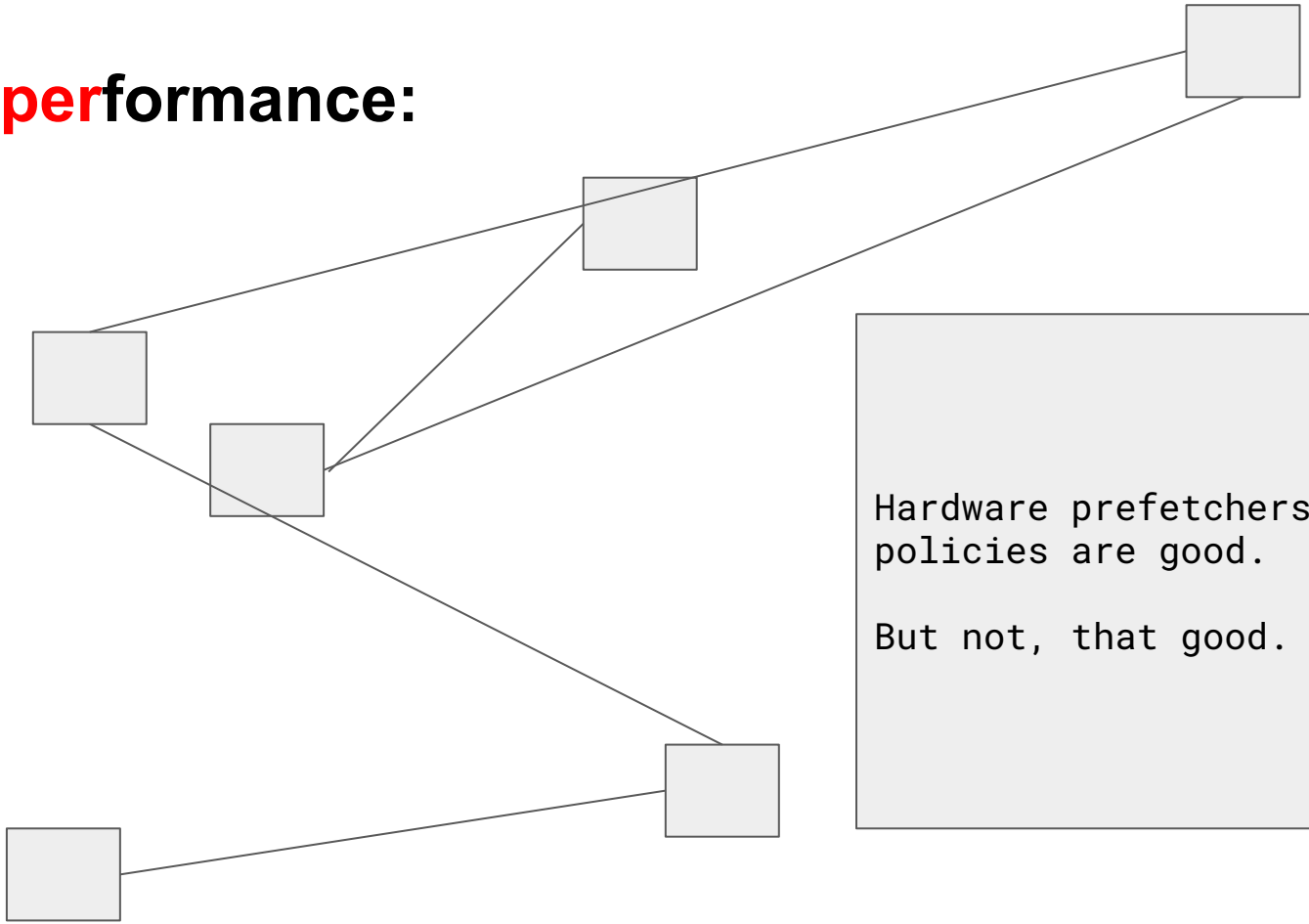
```
struct SEXP {

  Enum TAG; 8 bits

  Vector<SEXP> children;

          64bits + (n * 136 bits)

  Vector<Flag> flags;

          64bits + (m * 32 bits)

}
```

performance:



Hardware prefetchers and caching policies are good.

performance:



Hardware prefetchers and caching policies are good.

But not, that good.

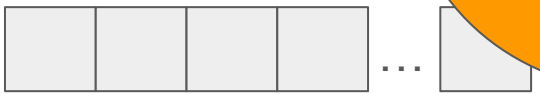
performance:

```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:

```
struct SEXP {
  Enum TAG; 8 bits
  NUM_CHILDREN; 32 bits
  OFF_CHILDREN; 32 bits
  NUM_FLAGS;    32 bits
  OFF_FLAGS;    32 bits
}
```

performance:



Temporal locality is good enough if the nodes queried are in the vicinity of ~30K nodes

```
truct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

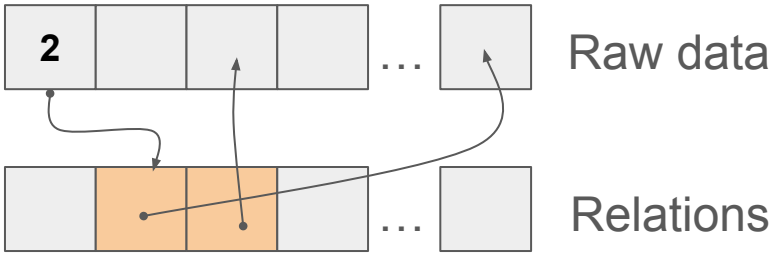
performance:

Raw data

Relations

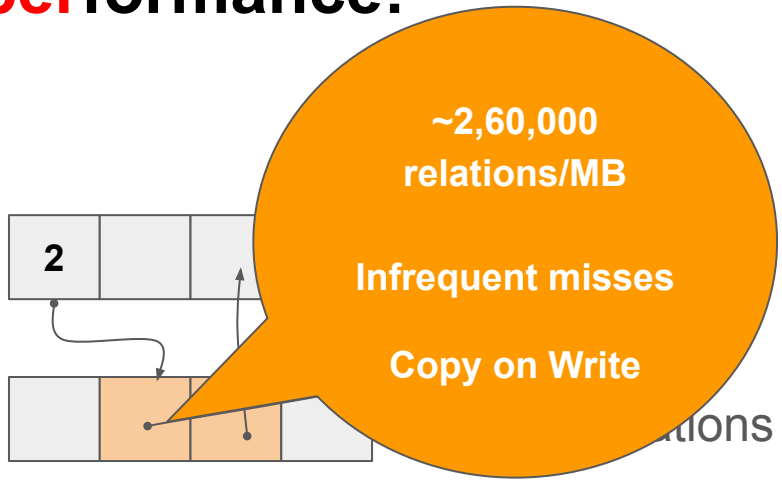
```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:



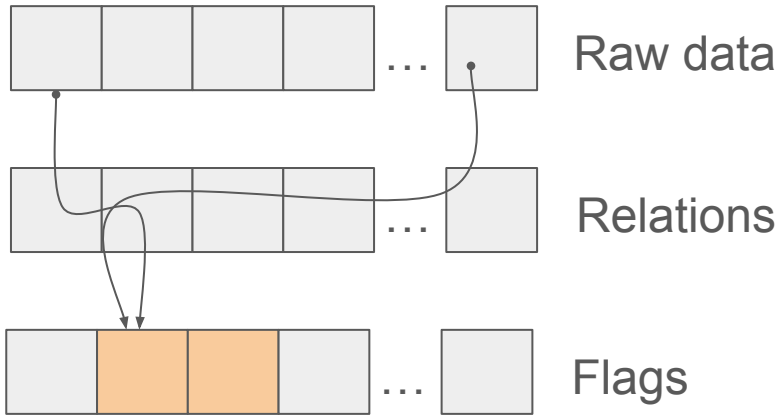

```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:



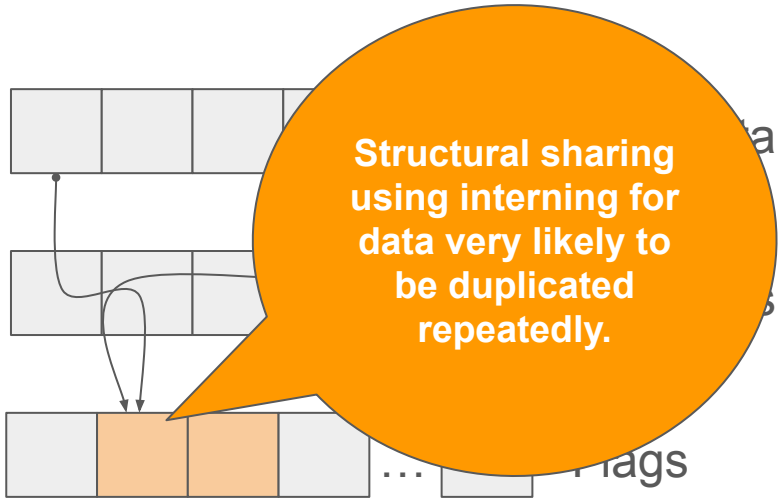

```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:



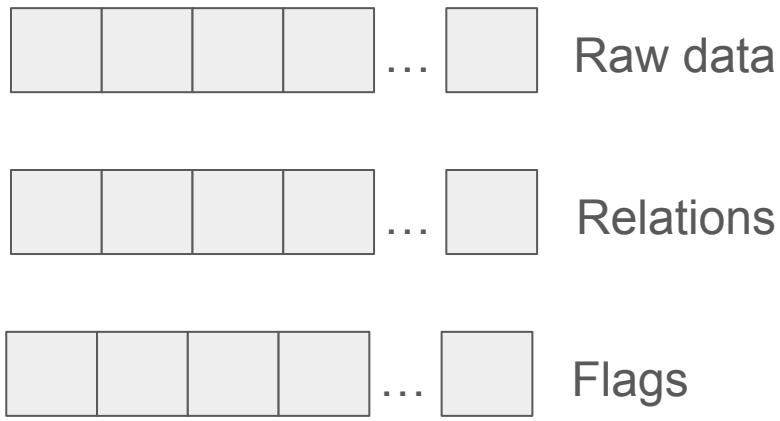

```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:




```
struct SEXP {

  Enum TAG; 8 bits

  NUM_CHILDREN; 32 bits

  OFF_CHILDREN; 32 bits

  NUM_FLAGS;    32 bits

  OFF_FLAGS;    32 bits

}
```

performance:



We do employ more levels of interning for things like tracking binding metadata, strings and well known SEXP nodes.

performance:

Not always the worst case, **average case matters**.

Improve locality == **Make each instance very cheap**

De duplicate smartly.

- Sometimes it’s okay to **let some memory leak**.

Memory saving != Performance Improvement

conformance:

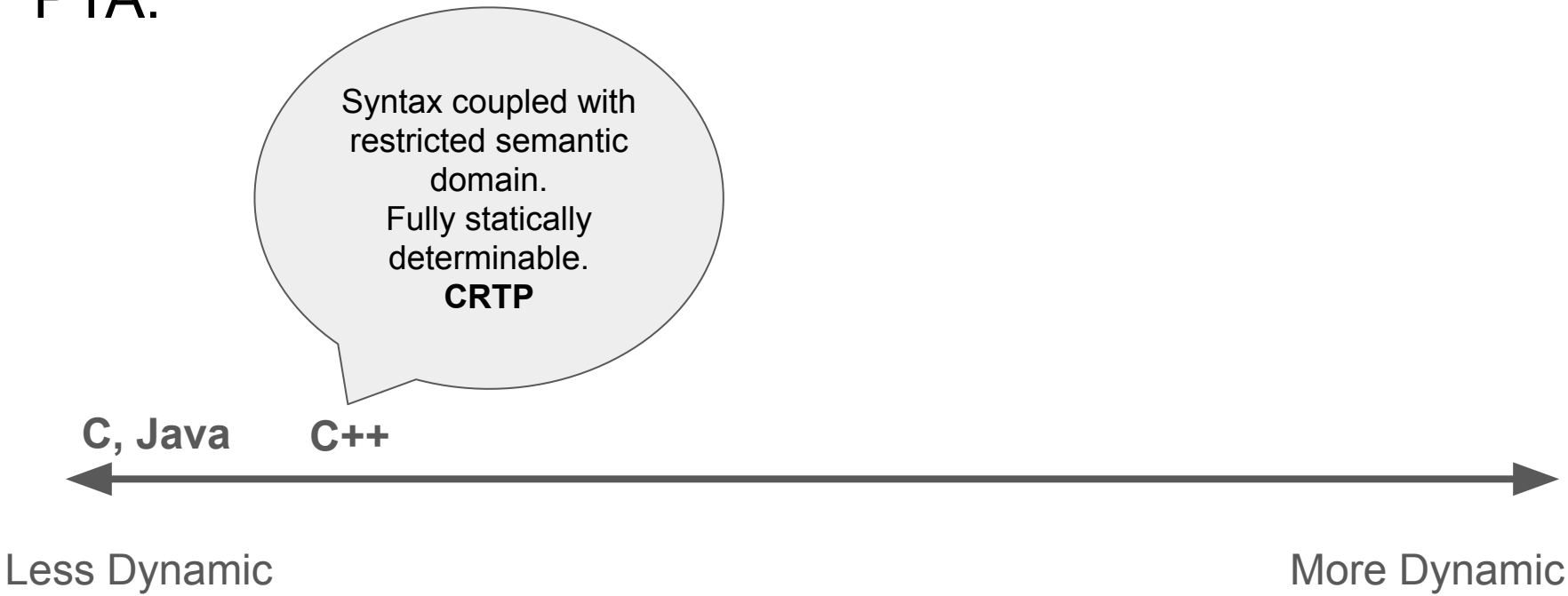
```
🚀 Iridium Test-262 Runner (--iri)
Threads: 64 | Target: test/language
--- Final Report ---
Total:                          32803
PassCount:                      32732
✔ True Passes (Pass Both):      26442
➖ Failed Both (Err Match):       1040
➖ Failed Both (Err Mismatch):      14
★ Over Compliant (Fixes):       26
✖ True Regressions:             57
✖ Timeout Regressions:          0
⊘ Ignored (Compile Errors):     5224
📊 Overall Bar Rate:            99.78%
```

PT.|.

PTA:

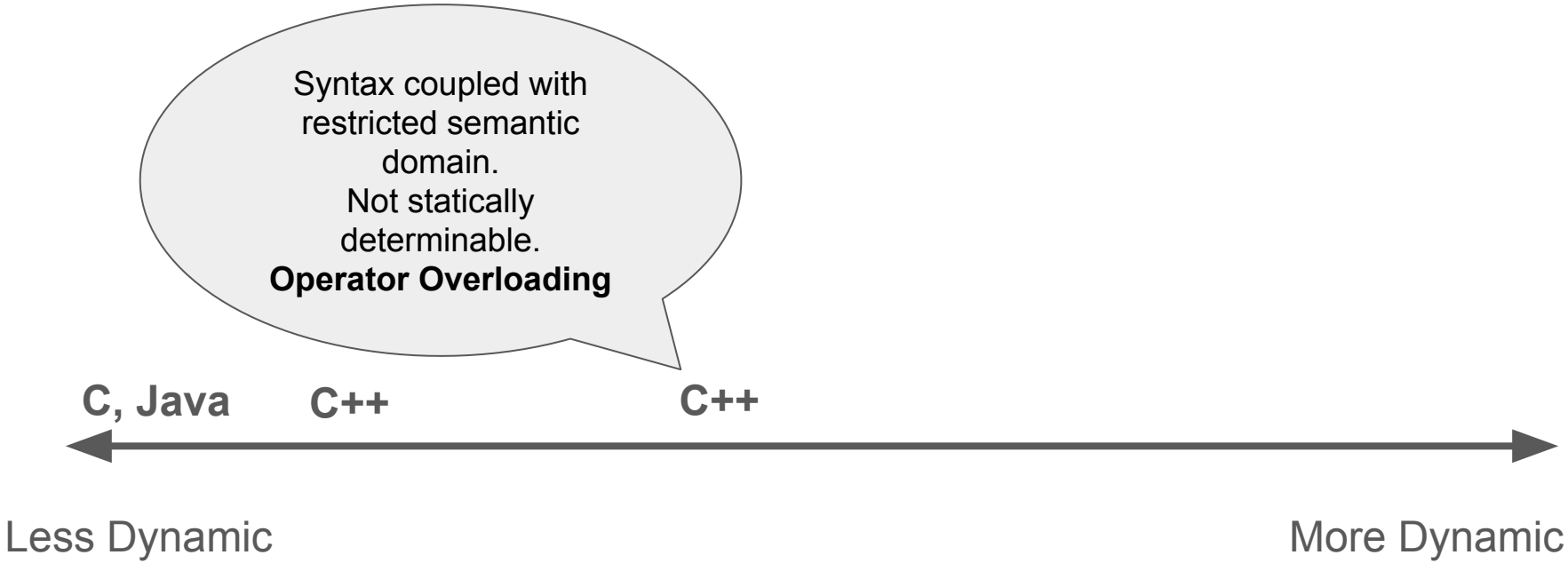
~Meetesh's rough classification of dynamic language features that determine static analyzability.

PTA:



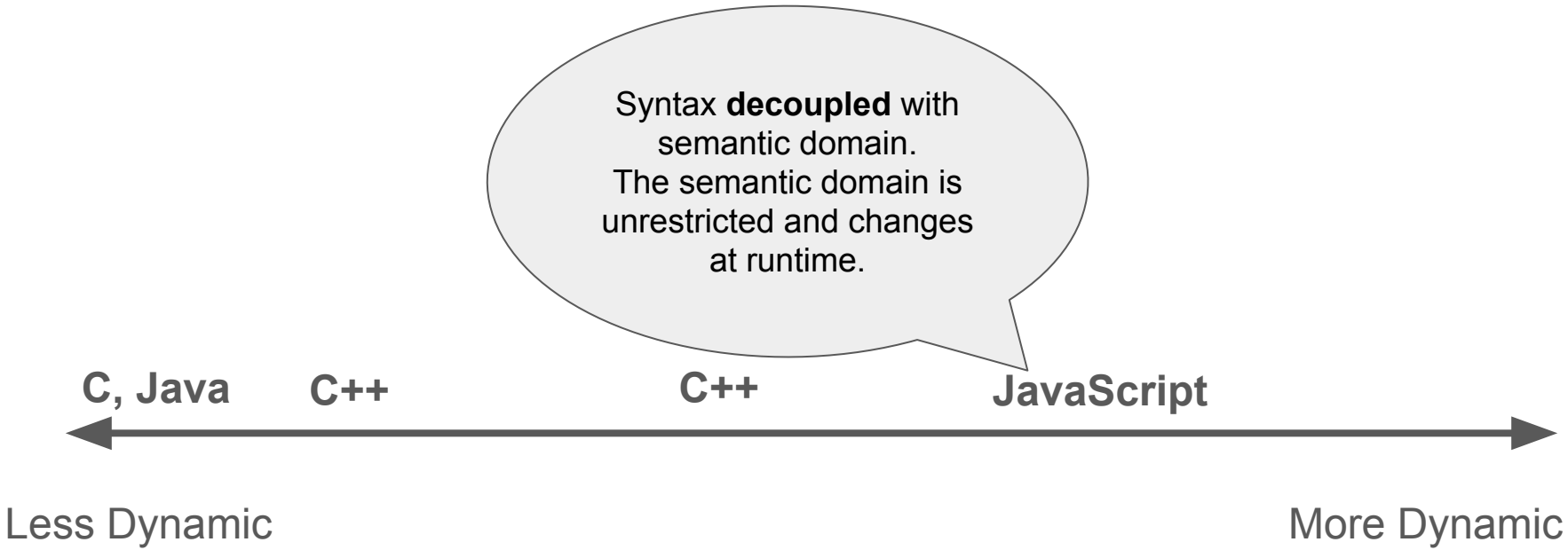
~Meetesh's rough classification of dynamic language features that determine static analyzability.

PTA:



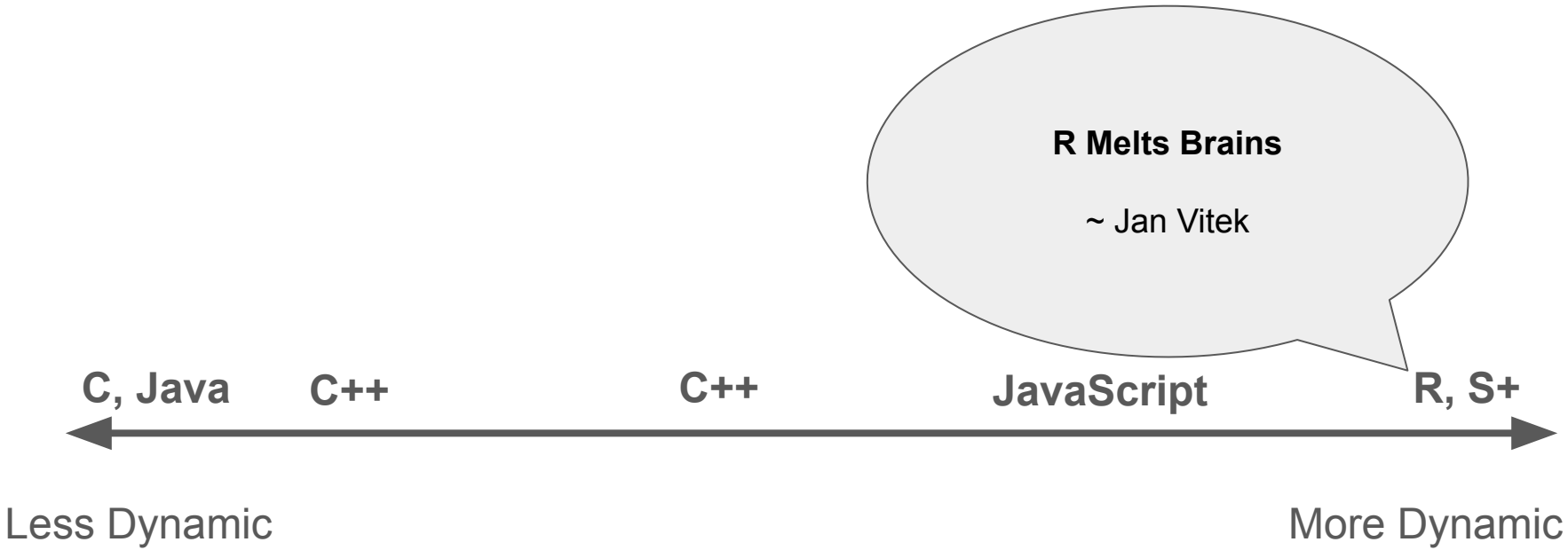
~Meetesh's rough classification of dynamic language features that determine static analyzability.

PTA:



~Meetesh's rough classification of dynamic language features that determine static analyzability.

PTA:



~Meetesh's rough classification of dynamic language features that determine static analyzability.

PTA: **Define Calculus, Project Semantics**

IRI in some sense was designed and developed, keeping in mind prior knowledge of existing well known languages.

The semantics which were finally left, managed to model a large chunk of the language. This was mainly achieved using **brute force**.

This is **“borrowed knowledge”**, JavaScript does not define itself as a loose function of existing language semantics.

It is **self contained**, **well defined** and **fully documented**.

PTA: **Define Calculus, Project Semantics**

IRI, in some sense, broke down semantics which **belonged to the runtime** and defined them as **new explicit syntax**.

- But what exactly did we really do? As an instance consider:
  - **ECMA Abstract Operation**
    - **10.2.11 ~ 29** :: was simplified ::
- We did this even more, without realizing.
- Designing an IR is more of an art, things are sometimes done just because they appear to fit better.

PTA: **Define Calculus, Project Semantics**

ECMAScript is a description of a programming system with **cooperating nodes**.

All meta information, including the semantics needed to **bind syntax with semantic points** is well defined.

- The fundamental idea in Prakriti is to project these semantics “as-is” into a graph based calculus.
  - Large amount of manual effort, but very interesting.

- *The currently presented specification is a work in progress and likely to change in the coming months.*

PTA: **Define Calculus, Project Semantics**

Some interesting observations up until now:

- JavaScript await continuations may be defined as a general case of the problem of “recomputing PTA information when the **source code** is changed”

- Closure captured bindings may be described as “**Global aggregates in C like languages** where they have special lifetimes in which strong updates are safe”.

PLATO

PLATO @ IITB