Università della Svizzera italiana

Faculty of Informatics

CGO GraalVM Workshop
Washington DC, USA
16-02-2019

Efficient Dynamic Analysis for JavaScript / Node.js with NodeProf on GraalVM

Haiyang Sun
Advisor: Walter Binder

Università della Svizzera Italiana (USI)

JavaScript and Node.js

- Popular among developers
  - Easy to learn
  - Fast runtime
  - Huge ecosystem of libraries and frameworks
    - Node Package Manager (NPM)

Source: https://www.npmjs.com/

JavaScript / Node.js

- … though popular, easy to make mistakes
  - Dynamically typed
  - `eval`
  - Asynchronous event-driven programming
  - Easy to introduce bugs (performance / correctness / vulnerability)
    - Difficult to detect with static analysis
- Need for dynamic analysis tools
  - However, existing ones are limited

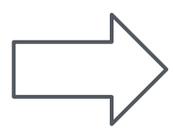
Dynamic Analysis Tools

- CPU / Memory profilers shipped with the runtime
  - **v8**: `--prof`
  - **GraalVM**: `--cpusampler --memtracer`
  - Strengths:
    - Performance diagnosis
    - Low-overhead
  - Weaknesses:
    - Not extensible

Dynamic Analysis Tools

- Jalangi [1]
  - Source-code instrumentation
    - Add hooks for dynamic analysis events
      - Function calls
      - Variable accesses
      - Field accesses
      - ...

```
function f(a,b,c) {
    return a > b[c];
}
```



```
J$.iids = {"9":[2,10,2,11], //...Source code mapping
function f(a, b, c) {
  try {
    // Arguments handling
    J$.Fe(57, arguments.callee, this, arguments);
    arguments = J$.N(65, 'arguments', arguments, 4);
    // Variable reads tracking
    a = J$.N(73, 'a', a, 4);
    b = J$.N(81, 'b', b, 4);
    c = J$.N(89, 'c', c, 4);
    // Binary operator tracking
    return J$.X1(49, J$.Rt(41, J$.B(10, '<',
      J$.R(9, 'a', a, 0), J$.G(33, J$.R(17, 'b', b, 0),
      J$.R(25, 'c', c, 0), 4), 0)));
  } catch (J$e) {
    J$.Ex(121, J$e);
  }
}
```

[1] https://github.com/Samsung/jalangi2

Dynamic Analysis Tools

- Jalangi
  - Strengths:
    - Flexible
  - Weaknesses:
    - High overhead
      - ~10^3 slowdown (peak performance)
    - Error-prone and difficult to maintain
      - Complex semantics and fast evolving
      - Any mistake in modification can crash the app
    - Limited coverage
      - Cannot instrument any built-in library

Our Solution

- **NodeProf**
  - A plugin tool GraalVM
    - *--jvm --jvm.Dtruffle.class.path.append=nodeprof.jar --nodeprof.Analysis=xxx*
  - Goal
    - As flexible as Jalangi
    - Less overhead
  - Joint paper [1]
    - USI
    - Oracle Labs (GraalVM Group, Dynamic Analysis Group)
  - Open source
    - Github: https://github.com/Haiyang-Sun/nodeprof.js

[1] Haiyang Sun (USI), Daniele Bonetta (Oracle Labs), Christian Humer (Oracle Labs), and Walter Binder(USI). *Efficient dynamic analysis for Node.js.* CC'18

NodeProf on GraalVM

- In the rest of my talk:
  - Instrumentation used by NodeProf
  - Case study: building a coverage tool in Java and JavaScript

Instrumentation

- Truffle instrumentation
  - Instrument at Abstract-syntax-tree (AST) level
  - Debugger [1]
- Compared to source-code instrumentation
  - More stable and easier to maintain
    - No source-code modification is required
  - More efficient
    - The instrumentation itself causes zero-overhead after compilation

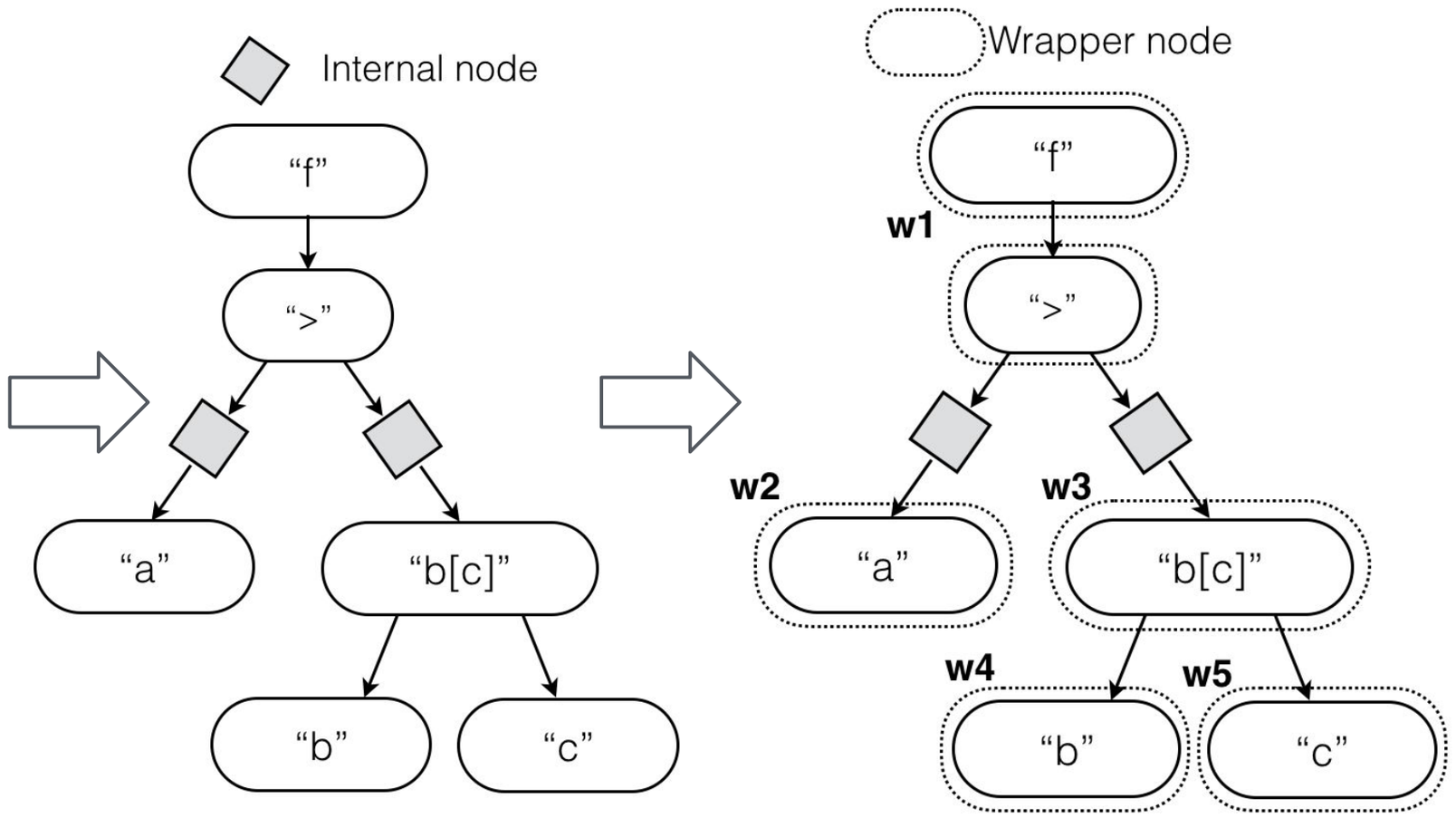
[1] Chris Seaton (Oracle Labs), Michael L. Van De Vanter (Oracle Labs), and Michael Haupt (Oracle Labs). *Debugging at Full Speed.* Dyla'14

Instrumentation

- Basics of Truffle instr. necessary to understand NodeProf
  - Node wrapping
    - `onEnter / onReturnValue / onReturnExceptional`
  - Node tagging
    - Select nodes for instrumentation

Instrumentation

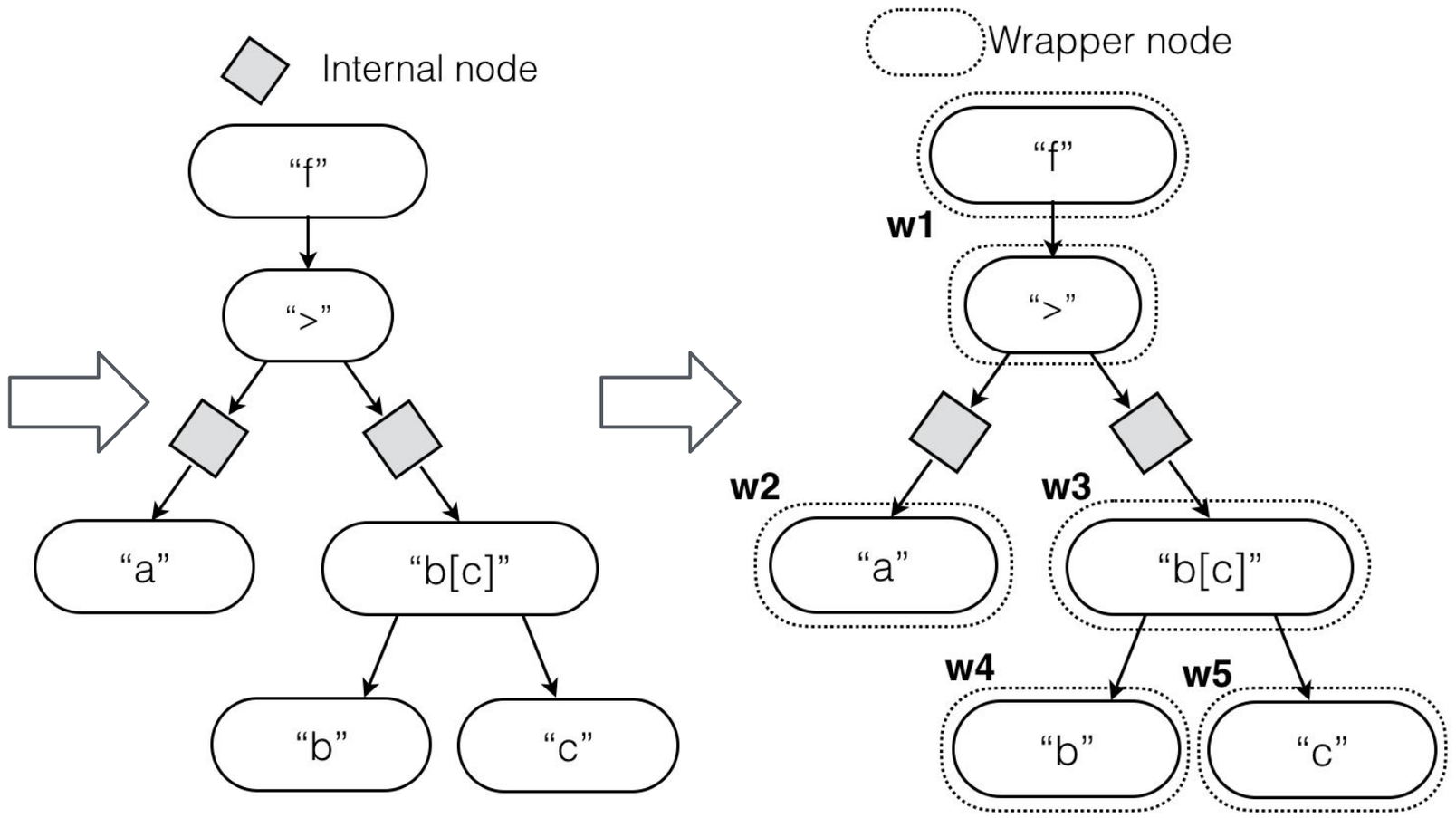
```
function f(a,b,c) {
    return a > b[c];
}
```



Instrumentation

- Not enough to build flexible dynamic analysis for JavaScript and Node.js

```
function f(a,b,c) {
    return a > b[c];
}
```

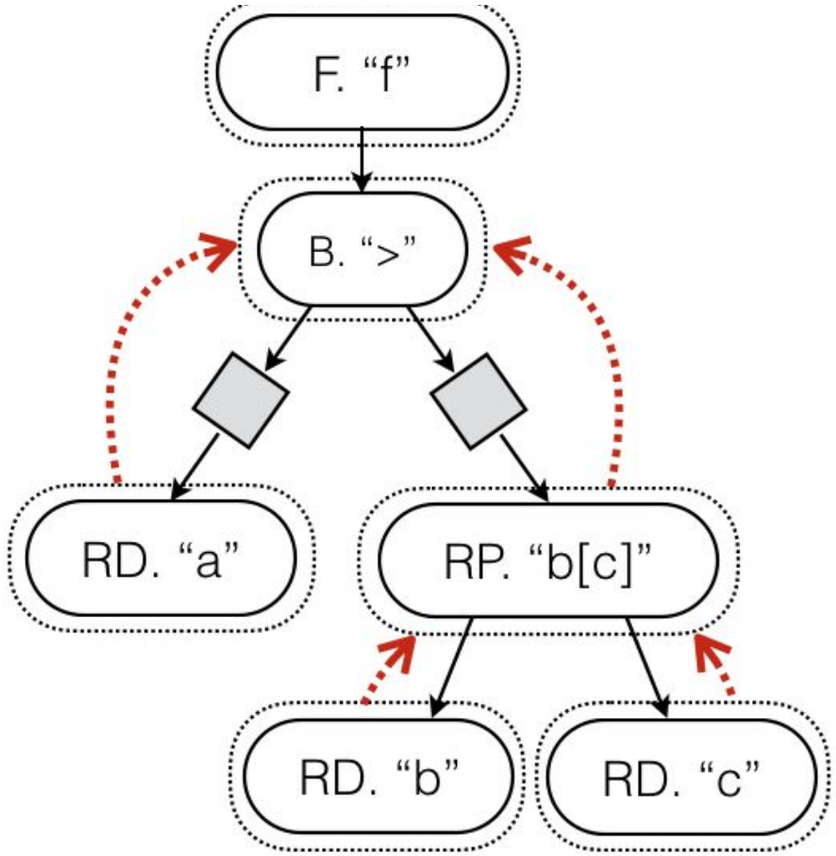


Extension: more JS tags

- Node tagging (similar to Jalangi hooks)
  - General
    - Expression
    - Statement
    - Root
  - JavaScript-specific
    - Invocation
    - Access
      - Variables
      - Object fields
    - Operators
      - Binary / Unary
    - Branch
    - Literals
    - ...

Extension: Inter-node Value Passing

- Input nodes
  - **Valid**
    - Avoid internal nodes
  - **Closest**
    - Avoid irrelevant nodes
- Receiving input values
  - `onEnter`
  - `onReturnValue`
  - `onReturnExceptional`
  - **`onInputValue`**



Use Case: Coverage Analysis

- Simple + Illustrative
- Coverage metrics:
  - Function
  - Statement
  - Branch
    - `if / while / for / switch / ?:`
    - Short-circuiting (binary operator)
      - `&& / ||`
- Performance comparison with Jalangi

Java API

final HashMap<SourceSection, Counter> stmts, funcs, falseBranches, trueBranches;

Wrapper

final Counter c;

onEnter:
c.inc()

StatementTag / RootTag

Wrapper

final Counter c1, c2;

onReturnValue (cond)*:*
if (isJSTrue(cond))
c1.inc()
else
c2.inc()

BranchTag

Wrapper

final Counter t, f;

onInputValue (index, value):
*if (**op**== "&&" || **op** == "||") {*
if (index == 0) { // left operand
c1.inc()
} else { // right operand
c2.inc()
c1.dec()
}
}

BinaryTag

Java API

final HashMap<SourceSection, Counter> stmts, funcs, falseBranches, trueBranches;

Wrapper

final Counter c;

onEnter:
c.inc()

StatementTag / RootTag

Wrapper

final Counter c1, c2;

onReturnValue (cond)*:*
```
if ( isJSTrue(cond) )
  c1.inc()
else
  c2.inc()
```

BranchTag

Wrapper

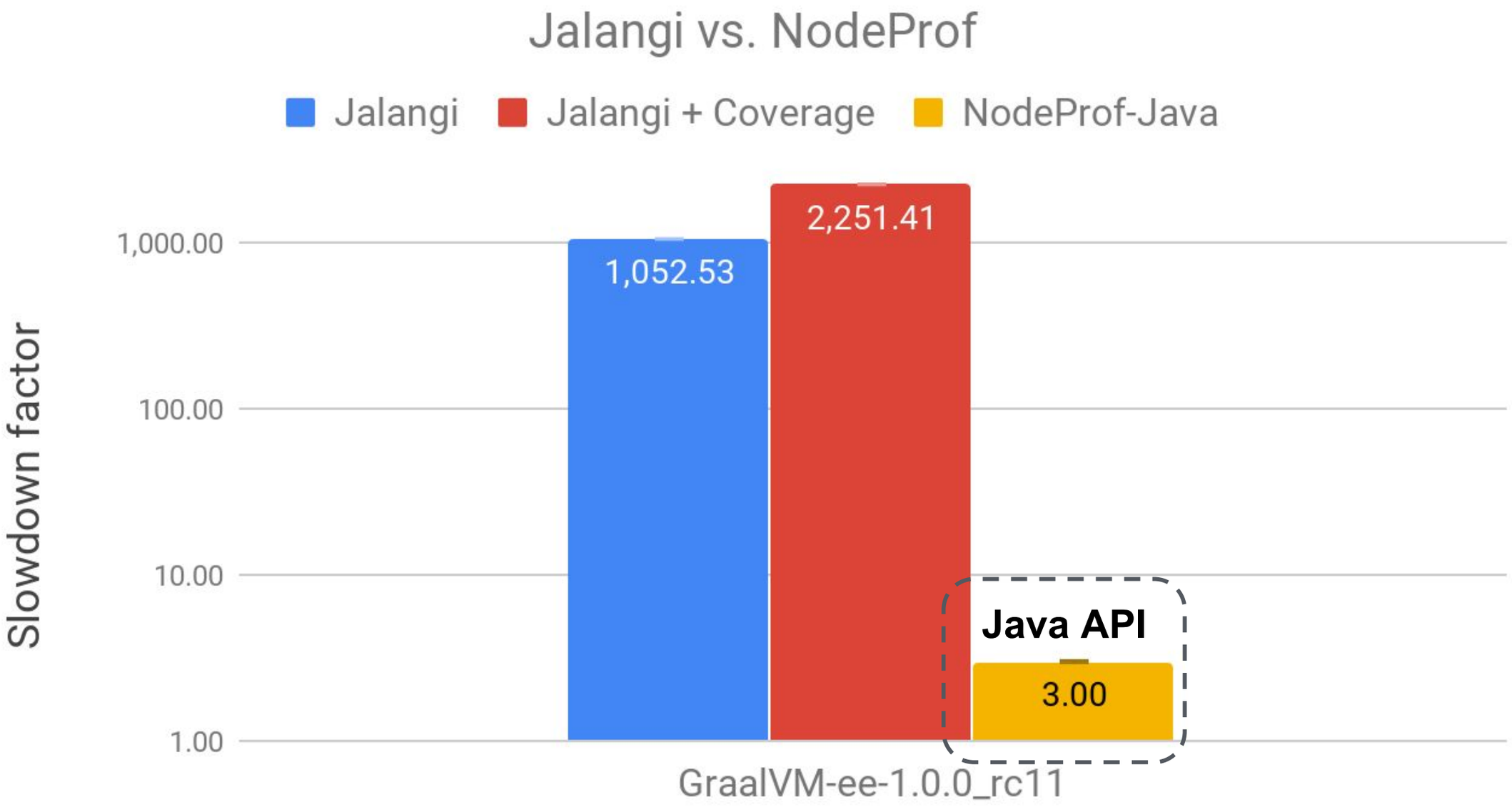
final Counter t, f;

onInputValue (index, value):
```
if ( op== "&&"  || op == "||" ) {
   if (index == 0) { // left operand
      c1.inc()
   } else { // right operand
      c2.inc()
      c1.dec()
   }
}
```

BinaryTag

Java API

- Performance:

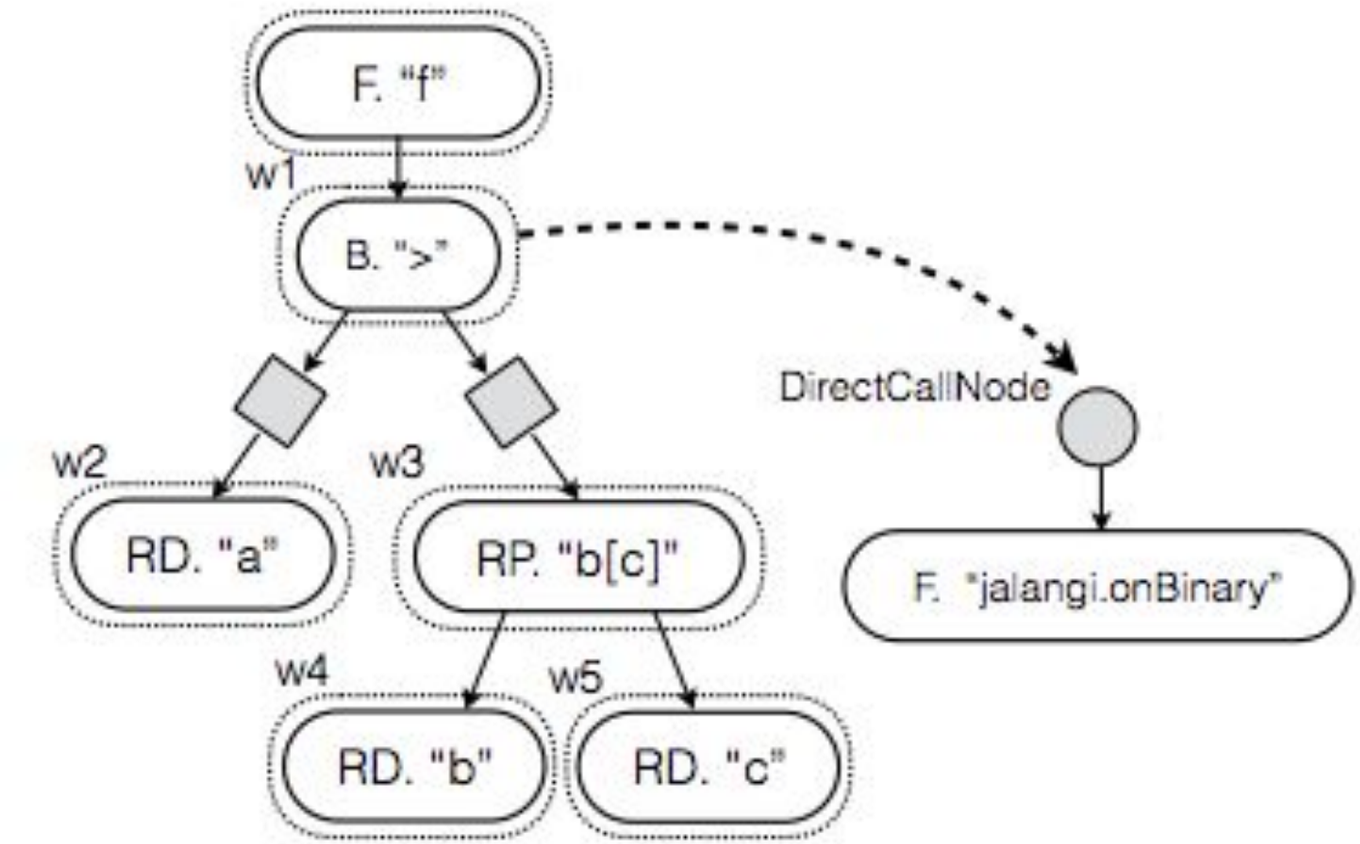


Java API

- Weakness:
  - Not friendly to JavaScript users
    - Requires familiarity with Java, Truffle and Graal.js API
  - Changes of the analysis always need to be re-compiled

JavaScript API

- Similar and compatible to Jalangi
  - Define hooks in JavaScript
  - Call from Java
    - Interoperability
    - Graal Polyglot API

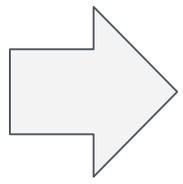


JavaScript API - Object

```
const stmts = {},  funcs = {}, trueBranches = {}, falseBranches = {};
function incMap (map, iid) => {
    let counter = map[iid] || 0;
    map[iid] = counter+1;
}
 this.statement = function(iid) {
    incMap(stmts, iid);
}
this.functionEnter = function(iid) {
    incMap(funcs, iid);
}
this.conditional = function(iid) {
    incMap(trueBranches, iid);
    incMap(falseBranches, iid);
}
```

JavaScript API - Map

```
// map: {}
function incMap (map, iid) => {
    let counter = map[iid] || 0;
    map[iid] = counter+1;
}
```



```
// map: new Map()
function incMap (map, iid) => {
    let counter = map.get(iid) || 0;
    map.set(iid, counter+1);
}
```

JavaScript API - Object' / Map'

```
// map: {}
function incMap (map, iid) => {
    let counter = map[iid] || 0;
    map[iid] = counter+1;
}
```

```
// map: new Map()
function incMap (map, iid) => {
    let counter = map.get(iid) || 0;
    map.set(iid, counter+1);
}
```

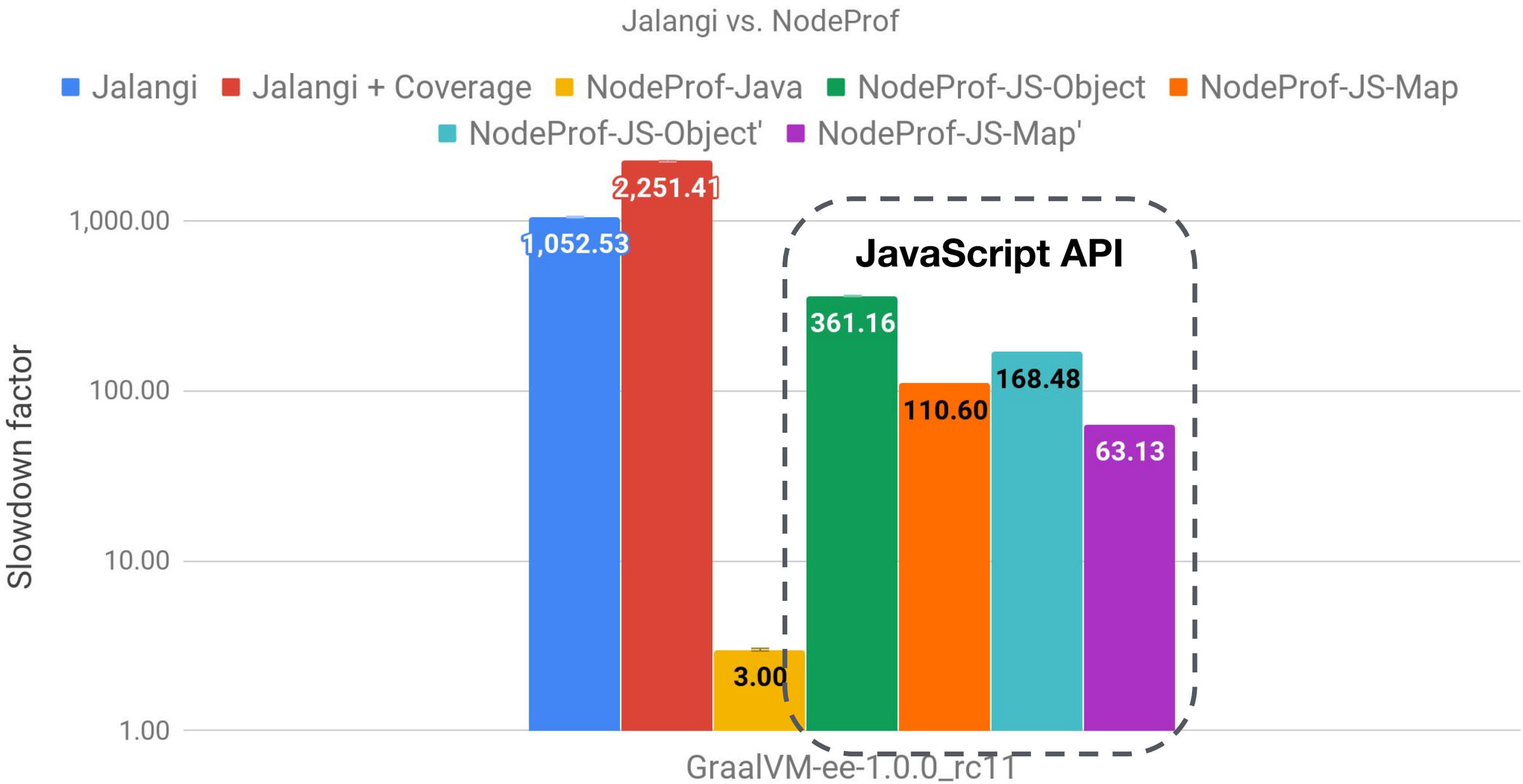
2 map operations (1 get + 1 set)

1 map operation (1 set)

```
// map: {}
function incMap (map, iid) => {
    map[iid] = true;
}
```

```
// map: new Map()
function incMap (map, iid) => {
    map.set(iid, true);
}
```

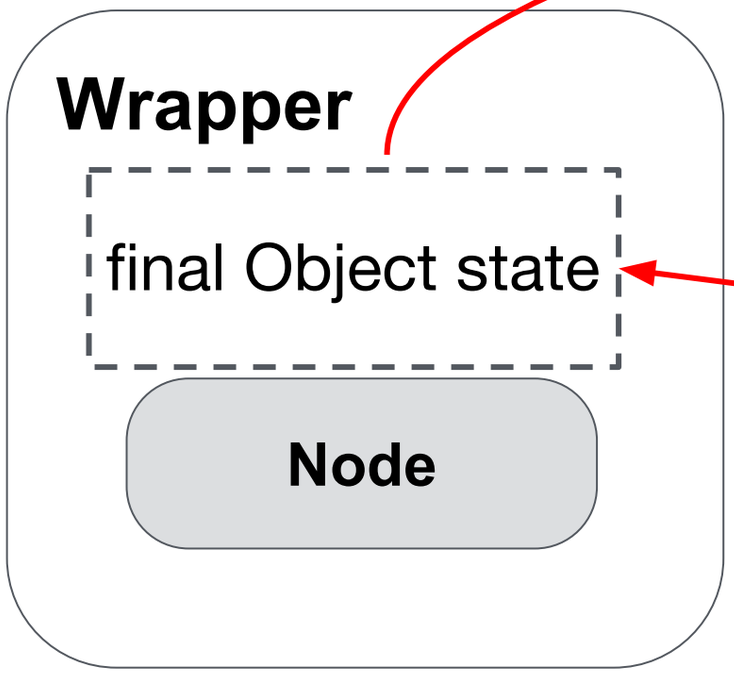
JavaScript API



JavaScript API - Interop

State cached in Java

Logic remains in JavaScript

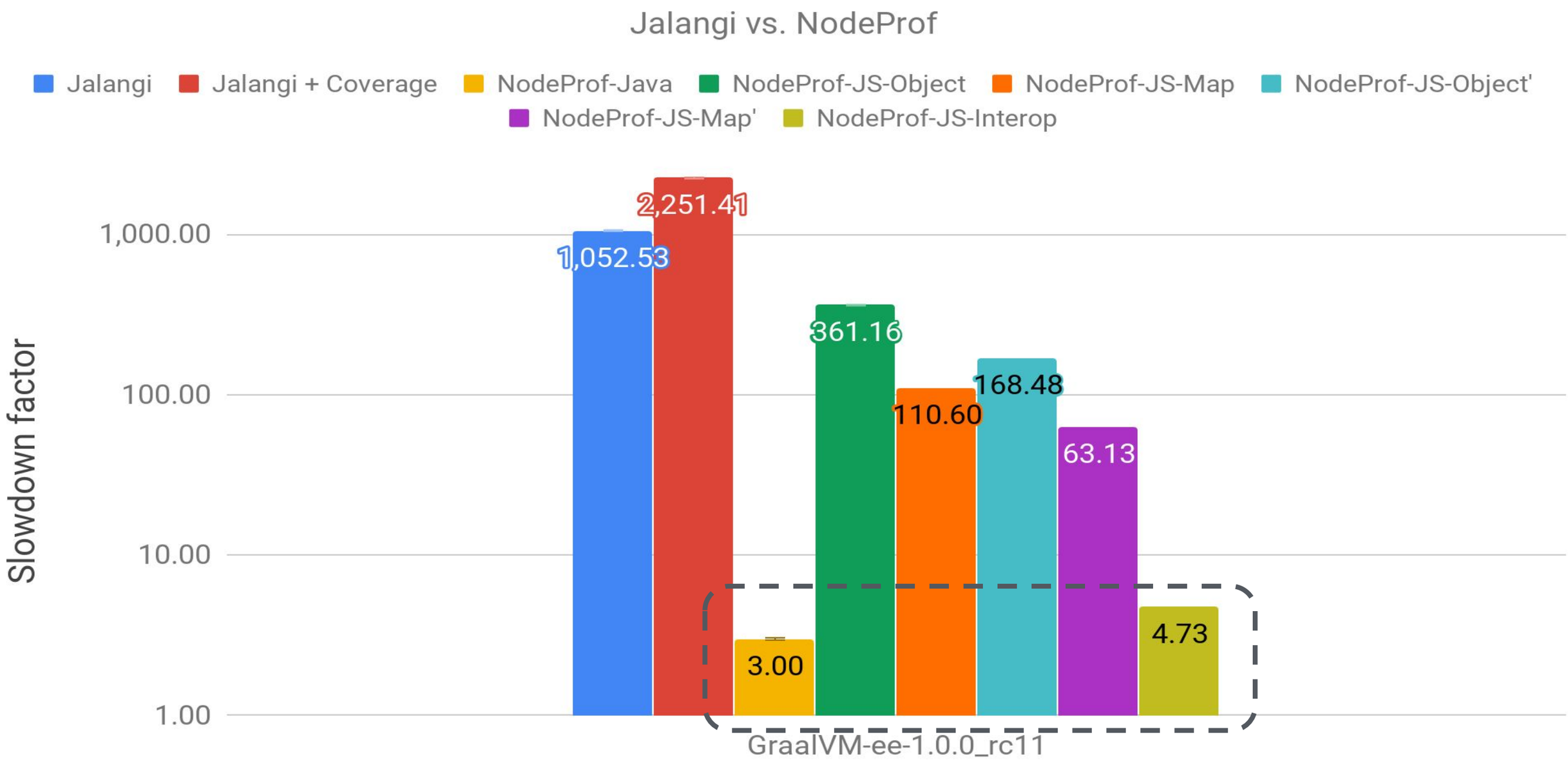


interop

```
this.statement = function(iid, state){
    state.cnt++;
}
// cb to create the state in Java
this.statement.createState = function(iid){
    let res = map.get(iid);
    if(!res) {
        res = {cnt: 0};
        map.set(iid, res);
    }
    return res;
}
```

JavaScript API - Interop

- Comparable performance after optimization
- Space for improvement



Conclusions

- **NodeProf**: dynamic analysis framework for Node.js
  - Based on Truffle instrumentation
  - As flexible and much lower overhead wrt. Jalangi
- Open questions:
  - Interference of instrumentation with dynamic compiler optimizations
- Github: https://github.com/Haiyang-Sun/nodeprof.js
- CGO talk on Monday
  - Reasoning about the Node.js Event Loop using Async Graphs