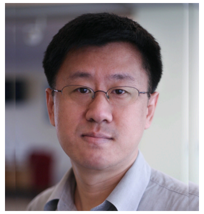
Validating JIT Compilers via Compilation Space Exploration



Cong Li[+] Yanyan Jiang[+] Chang Xu[+] Zhendong Su[++]

[+] Nanjing University, [++] ETH Zurich

The work was done when Cong was visiting Zhendong's AST Lab

Compilers: Critical System Software

- Almost all critical system software require a compiler
  - E.g., kernel, virtual machines and emulators, compilers etc.

GCC

LLVM

COMPCERT

Javac

Graal Compiler

ahead-of-time

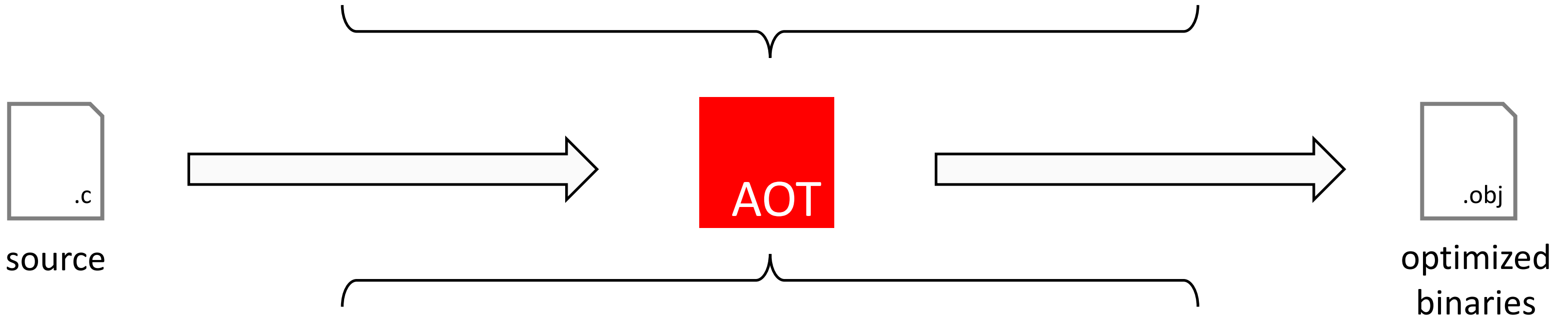
just-in-time

AOT (Ahead-Of-Time) Compilers

- Compile a program before running the program

Analysis Passes

- Liveness analysis, Invariant variable analysis, class hierarchy analysis …



Transformation Passes

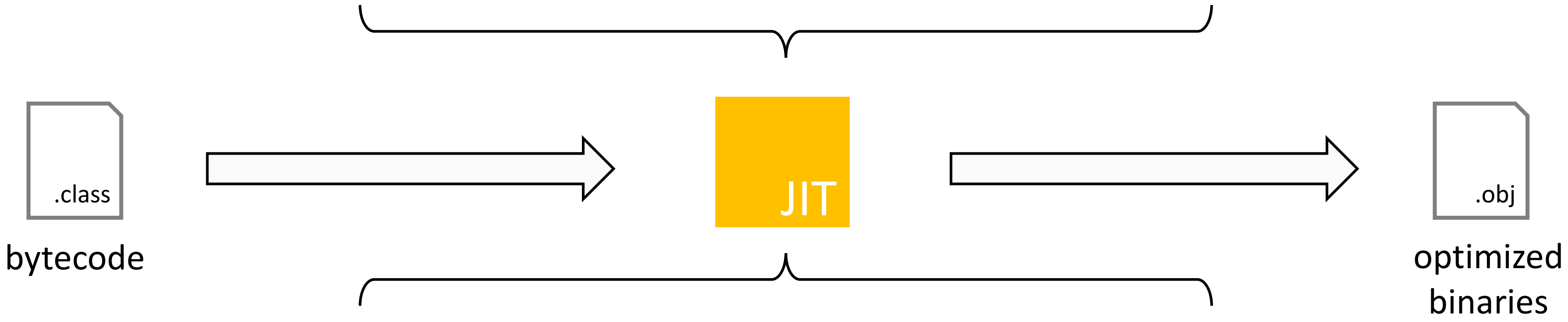
- Global value numbering, Dead code elimination, Loop invariant code movement, Loop vectorization, …

JIT (Just-In-Time) Compilers

- Compile a program while the program is running

Similar to AOT Compilers

- Analyses (LA, IVA, CHA, ..)
- Transformations (GVN, DCE, CSE, GCM, LICM, CFP, Inline, Outline, Loop Unroll., Loop Vec., Reg. Alloc., ...

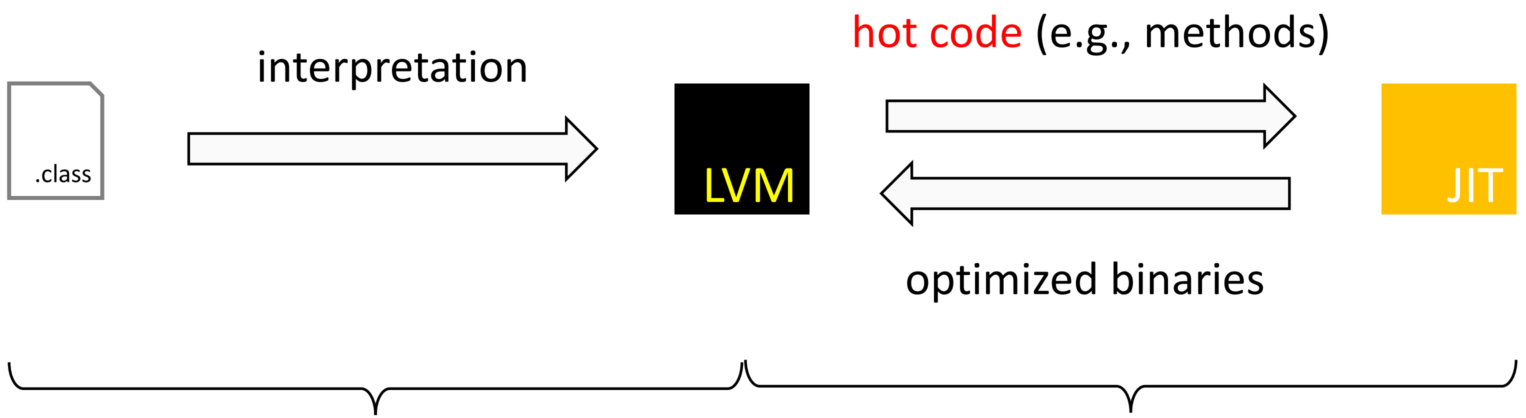


Different from AOT Compilers

- At runtime (Perf.: Graph Coloring to Linear Scan)
- On demand (whole program to partial program)

JIT Compilers: Critical and Widely Used

- Extensively used in various language virtual machines (LVMs)
  - Java VMs, JavaScript engines, .NET runtimes, etc.



Startup Fast

Run Slow

High Peak Performance

Cheaper JIT-Compilation Cost

JIT Compilers: Critical and Widely Used

- Extensively used in various language virtual machines (LVMs)
  - Java VMs, JavaScript engines, .NET runtimes, etc.

OpenJDK

C1/C2 Compiler

TurboFan

.NET

RyuJIT

GraalVM™

Graal Compiler

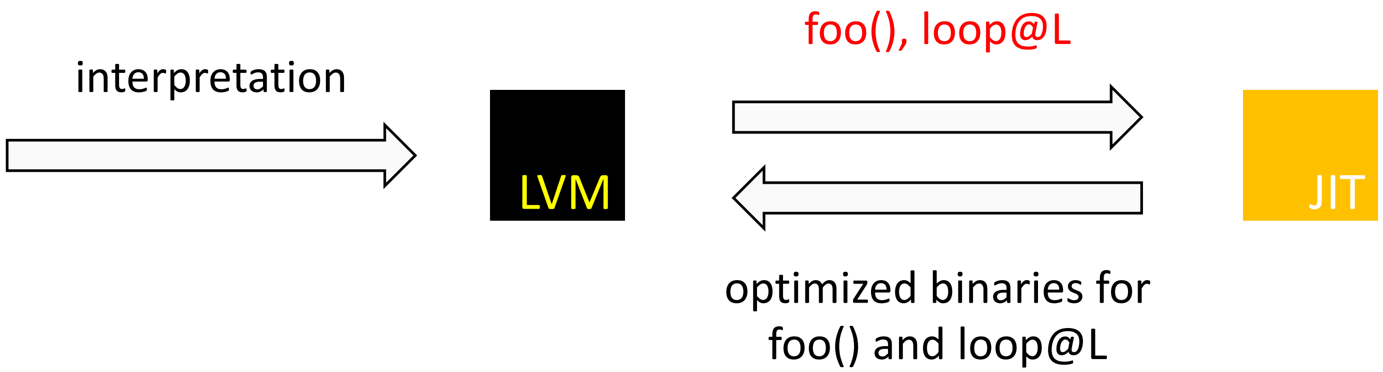
Dynamic, Partial JIT Compilation by Example

- Only hot code can be JIT compiled; all others are left interpreted

```
void bar() { }

int foo() { return 1 }

void main() {
 foo()
@L:
 for i in 1…90_000 {
  foo() // called e.g.
 }      // >1,000 times
 bar()
}
```

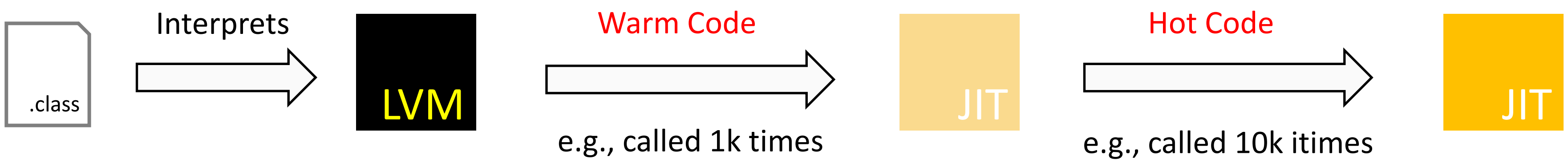
hot

hot



Tiered JIT Compilation

- Multiple JIT compilers, multiple tiers
- Compiles and optimizes code tier by tier



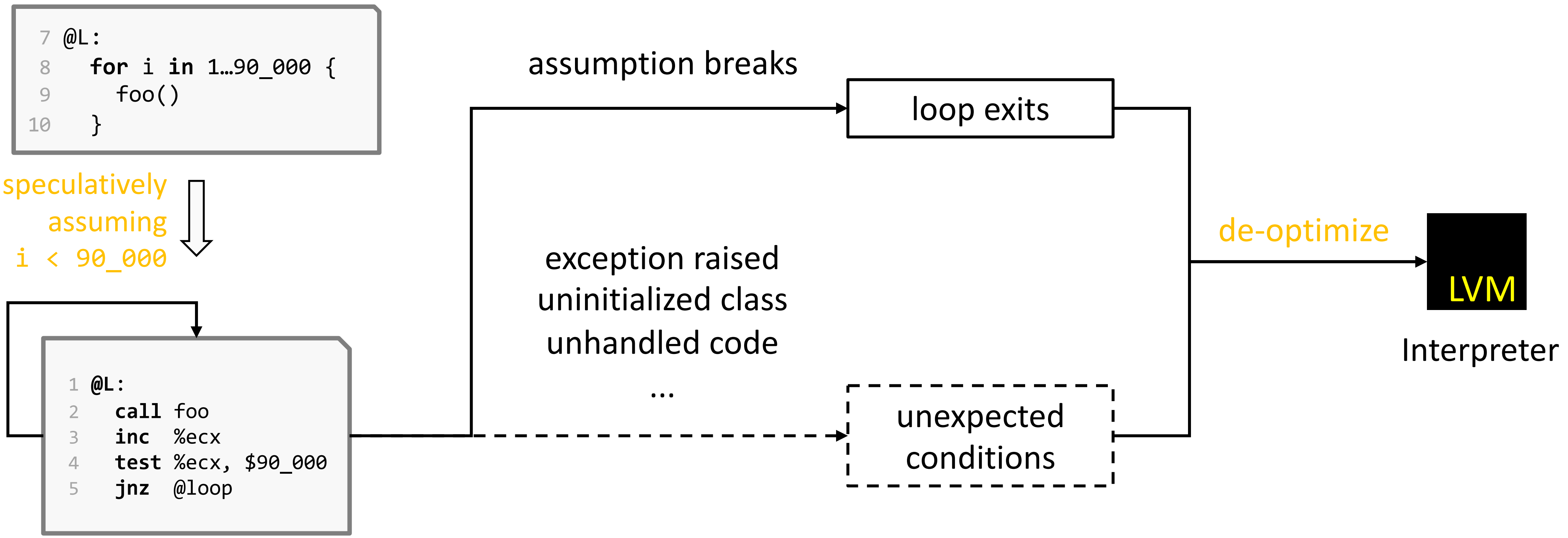
- Tier 0
- Interpretation
- Fast startup
- No optimizations

- Tier 1
- Mild optimizations
- Fast compilation
- Well optimized

- Tier 2
- Aggressive optimizations
- Slow compilation
- Super optimized

Bails Out: Speculations and De-Optimizations

- JIT Compilations are based on speculative assumptions
- De-optimization: LVM reverts to the interpreter



JIT Compilation: Super-Challenging Task

- Challenge I: Very deep LVM components
  - Only hot code can touch JIT compilers
- Challenge II: Pretty complicated compilations
  - On-the-fly, partial compilations and tiered compilations
  - Speculative compilations and de-optimizations
  - Many sophisticated optimization passes
- Challenge III: Intensive interactions
  - Interpreters and compiled code
  - GC and compiled code ...

Existing: Testing LVMs

- Program generators
  - JVM: dexfuzz, Java*Fuzzer, JFuzz, JAttack
    - dexfuzz: grammar-based bytecode
    - Java*Fuzzer, JFuzz: grammar-based source code
    - JAttack: template-based
  - JavaScript: jsfunfuzz, Fuzzilli, SkyFire, LangFuzz, CodeAlchemist
    - jsfunfuzz, Fuzzilli: grammar-based coverage-guided
    - SkyFire: grammar-based crowdsource
    - LangFuzz, CodeAlchemist: grammar-based borrowing problematic code from existing code base
- Program mutators
  - JVM: classfuzz, classming, JavaTailor, JOpFuzz
    - classfuzz: random mutation
    - classming: mutation by jump instructions
    - JavaTailor: randomly insert existing code
    - JOpFuzz: fuzz JVM's options
  - JavaScript: Superion, NAUTILUS, DIE
    - Superion, NAUTILUS: grammar-aware AFL mutations
    - DIE: aspect-preserving mutation

Challenge I
Very Deep LVM Components

Few are JIT compiler bugs
(most parser, verifier, interpreter)

Existing: Testing JIT Compilers

- Differential testing: ALL-INT vs ALL-JIT vs Default
  - JVM: dexfuzz@VEE14
  - JavaScript: JIT-Picker@CCS22
  - Smalltalk: Ranger@PLDI22
- Heuristic fuzzing
  - JVM: JITFuzz@ICSE24
  - JavaScript: FuzzJIT@CCS23

Challenge II & III
Pretty Compilated Compilations
Intensive LVM Interactions

Few affected optimizations
Few considered JIT choices
Unaware of compilation space

Ours (CSE), Compared with SOTAs

- Challenge I: 85 bugs and all the bugs found are JIT compiler bugs

SOTAs	Venue	#Bug	How many bugs can reach JIT compilers?
JAttack	ASE 22	6	Depends on templates
JavaTailor	ICSE 22	10	Depends on ingredients
classming	ICSE 19	14	Occasionally reach
JITfuzz	ICSE 23	36	Depends on seeds and mutators
JOptFuzz	ICSE 23	41	Depends on JVM options
classfuzz	PLDI 16	62	Occasionally reach
Ours: CSE	SOSP 23	85	Reach by design

Ours (CSE), Compared with SOTAs

- Affected 3 affected production JVMs with 3 bug types:
  - More than >20% are mis-compilations

	HotSpot	OpenJ9	ART	Total
Mis-compilation	1	9	8	18 (21%)
Crashes	30	28	8	66 (78%)
Performance Issues	1	0	0	1 (1%)
Total	32	37	16	85

Ours (CSE), Compared with SOTAs

- Challenge II/III: >20 affected optimizations and LVM components

HotSpot Components	Cx	#Bugs
Inlining	C1	1
Ideal Graph Building	C2	4
Ideal Loop Optimization	C2	10
Global Constant Propagation	C2	1
Global Value Numbering	C2	5
Escape Analysis	C2	1
Register Allocation	C2	2
Code Generation	C2	3
Code Execution	C2	3

OpenJ9 Components	#Bugs
Local Value Propagation	1
Global Value Propagation	2
Loop Vectorization	1
De-optimization	1
Register Allocation	1
Code Generation	2
Recompilation	1
Other JIT Components	6
Garbage Collection	13

Thanks from JIT Compiler Developers

TH

Anonymous
Bug reports you submitted for the HotSpot JVM
To: Li Cong

Hi Cong,

I'm Tobias from the HotSpot Compiler Team at Oracle and I noticed that you filed quite a few bug reports for the JITs recently, thanks a lot for that!

Now I was wondering, what kind of test generator are you using to find these bugs? Assuming it's part of your research, is there anything you could share with us?

Thanks and best regards,
Anonymous

From HotSpot

pshipton commented on Jun 15 Contributor

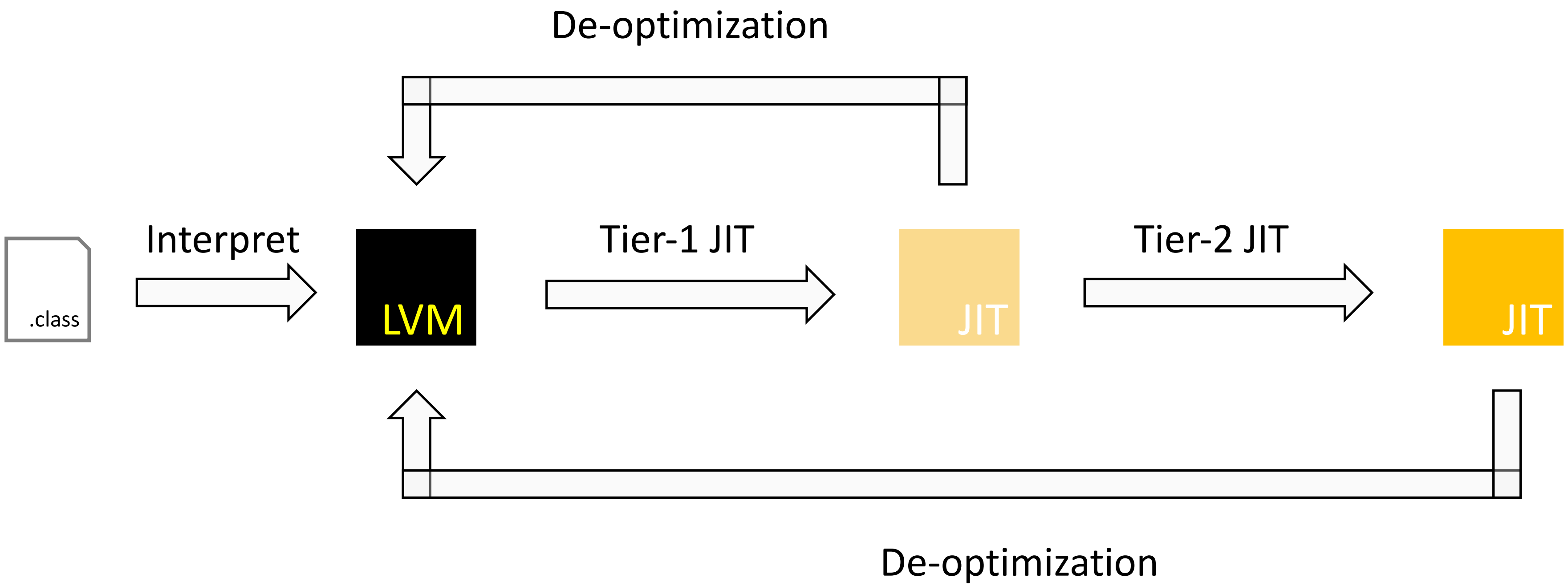
We have questions about using the test cases you've provided in our regression test suite. Are there any existing licenses or concerns about putting the test cases under the Eclipse v2 license? **@llxia** is interested in having you open a Pull Request to deliver the test cases, if you are are agreeable to that. We'd try to make it easy so you don't need to be concerned much about test frameworks. You'd have to sign the Eclipse ECA agreement for that, which is the typical requirement for delivering code. See https://github.com/eclipse-openj9/openj9/blob/master/CONTRIBUTING.md

From OpenJ9

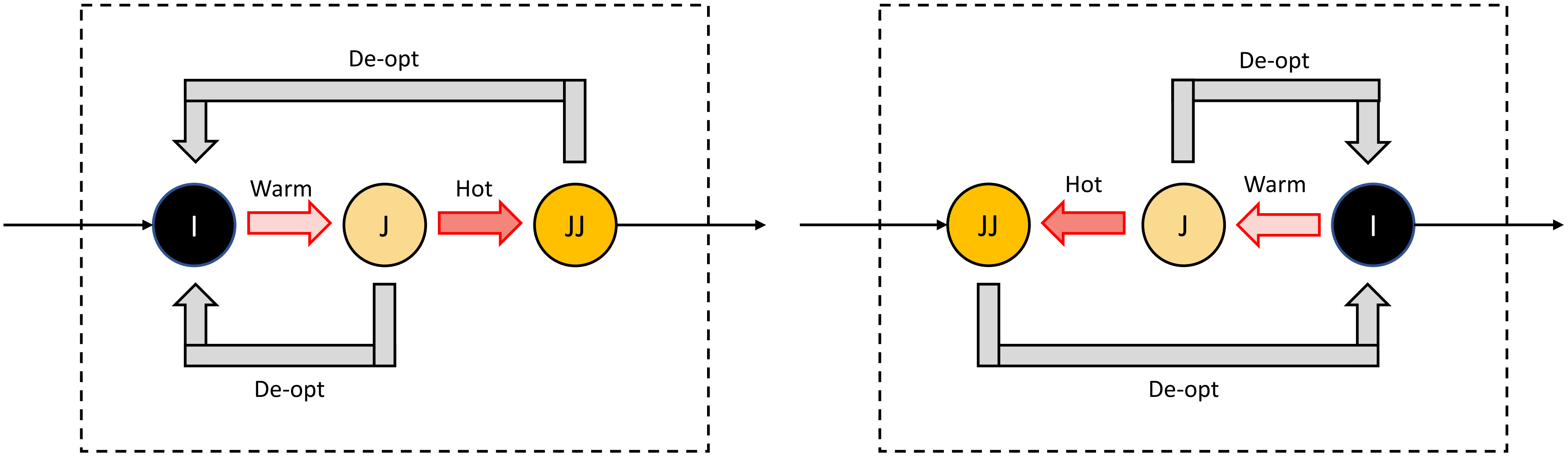
How did We Achieve This?

The Nature/Mechanism of JIT Compilation

- Each program running in the LVM are frequently transited from/to interpreted bytecode and JIT compiled code



Key Insights/Observation:
One Code Block, Multiple Execution State



(Assuming⁺ A) Simplest, Method-Based LVM

- Assumption I: All compiled code blocks are methods
  - Suppress other code blocks (e.g., loops)

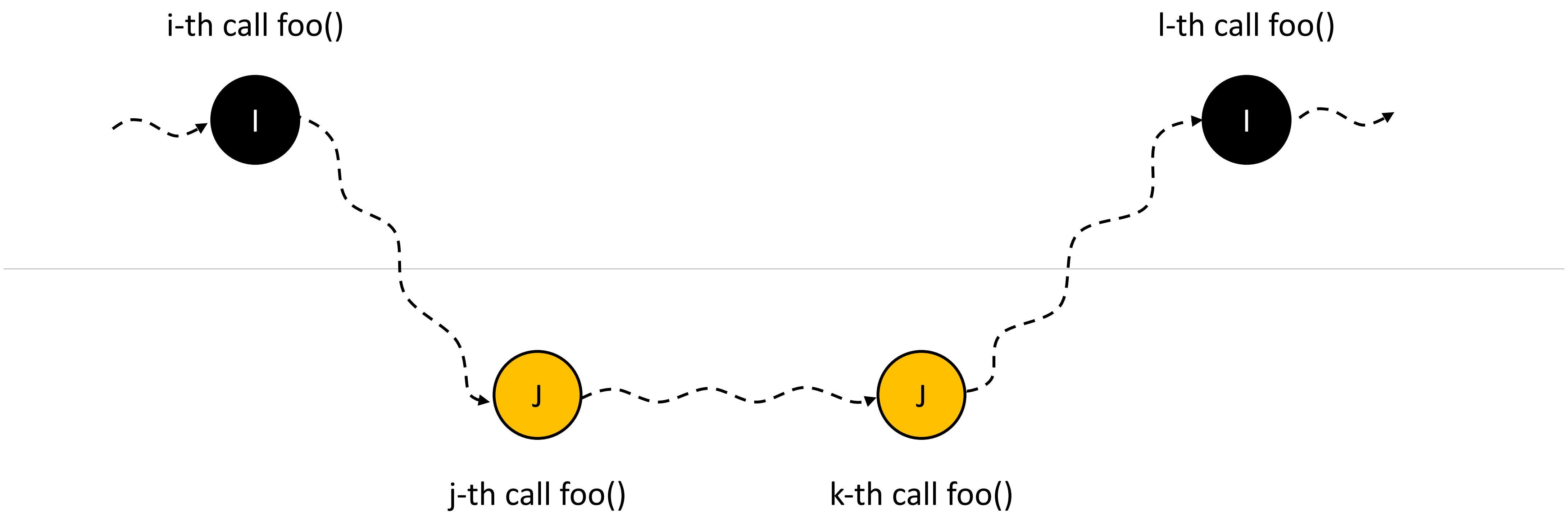
- Assumption II: Tier-0 interpreter and Tier-1 JIT compiler only
  - No other JIT compilation tiers

- Assumption III: De-optimization happens only when a method exits
  - Tier-1 JIT compiler can have no other uncommon conditions

[+] These assumptions are made only to have this slides/talk streamlined and easy to follow, which is not made by our work.

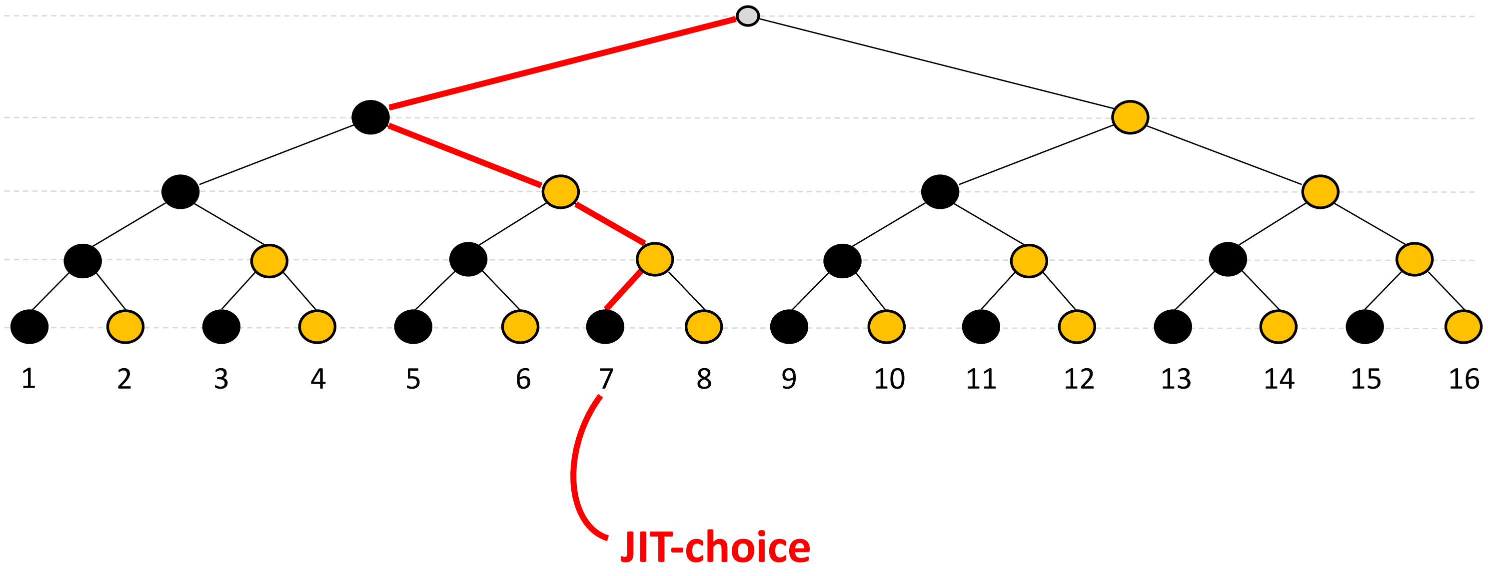
Two-State Transition of Each Call

- Finding I: Each method call is either in INT state or JIT state



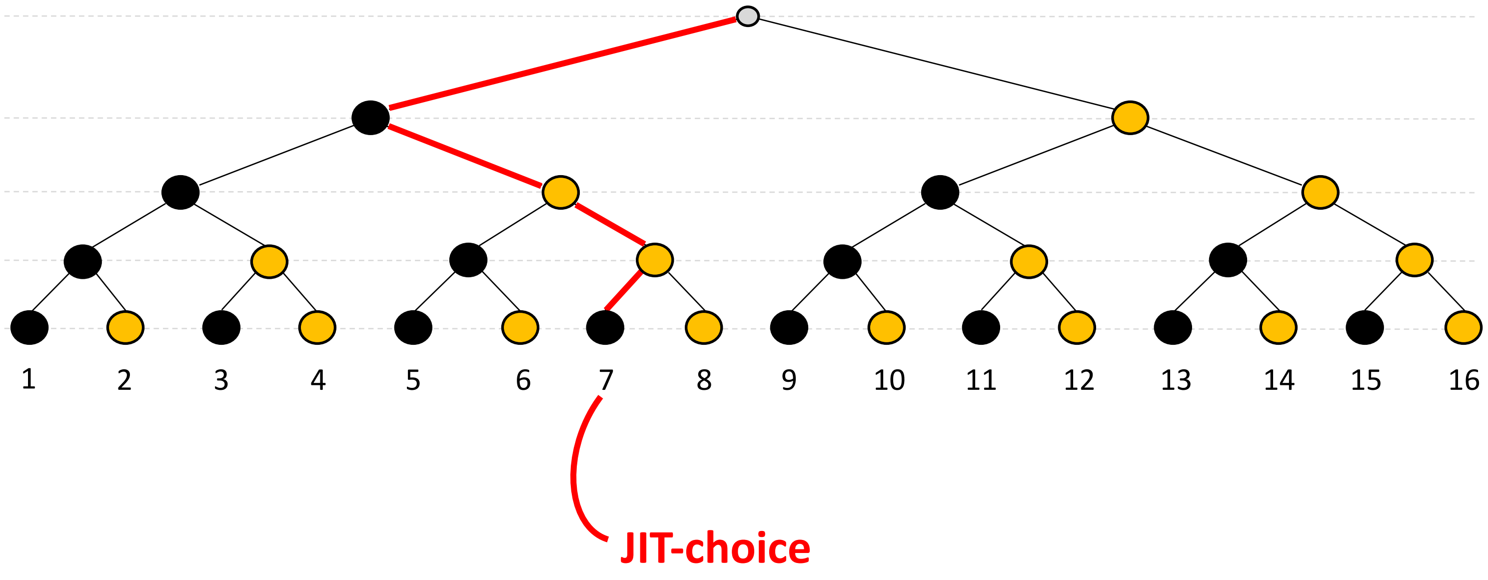
Compilation Space (modulo LVM)

- Finding II: N method calls => $\Omega(2^N)$ different JIT compilation choices
  - An *exponentially* large space: a big opportunity for testing



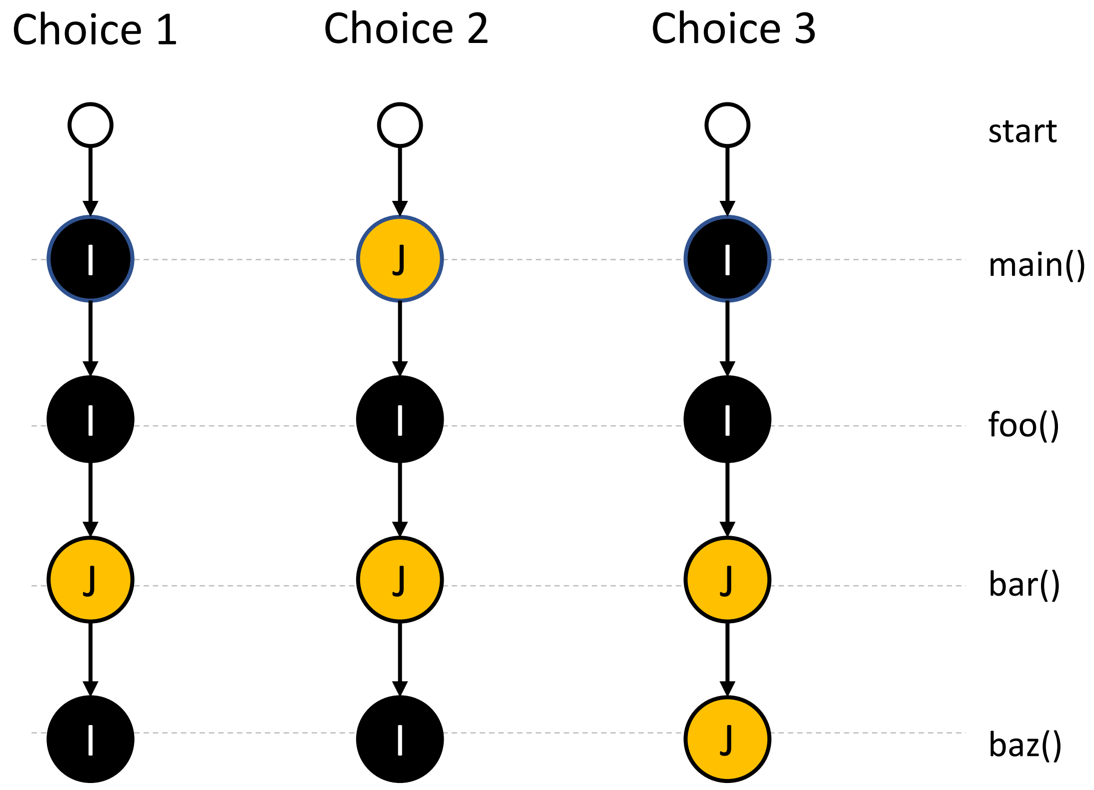
Key: The Same Program Output

- Finding III: Running the program with any choice lead to same result
  - Resolves the challenging, *oracle* problem in testing



Example: Consistently Print 2

```
int baz() { 1+1 }

int bar() { baz() }

int foo() { bar() }

void main() {
  print(foo())
}
```

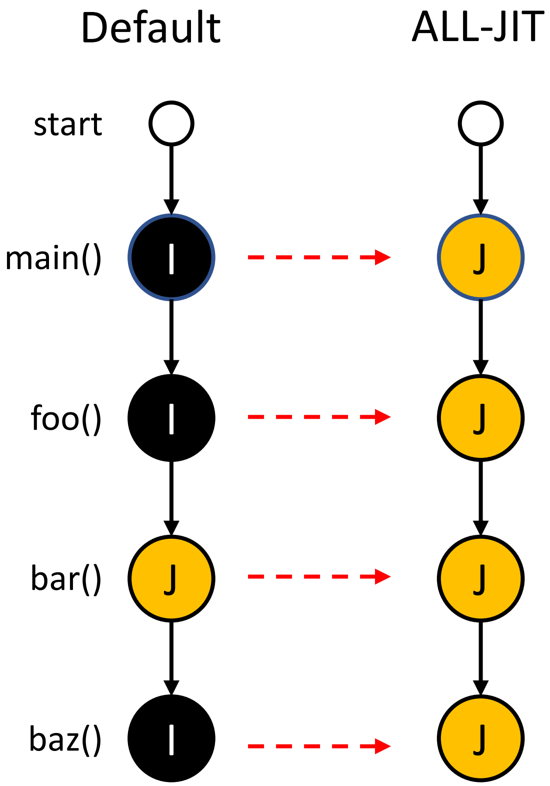


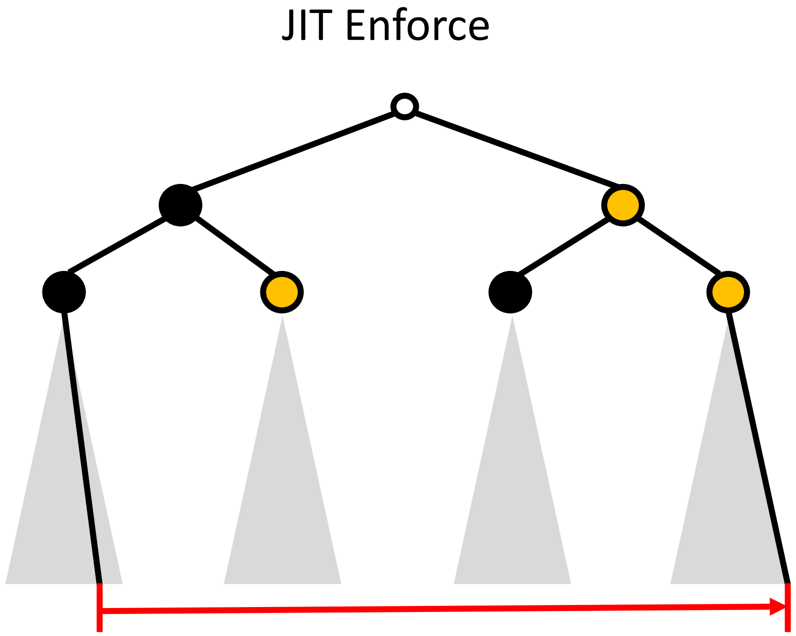
Big Opportunity and Test Oracle! How Could Validate JIT Compilers with Compilation Space?

Compilation Space and JIT-Choices

- JIT Enforce: Push every method call to be in JIT state

```
int baz() { 1+1 }

int bar() { baz() }

int foo() { bar() }

void main() {
  print(foo())
}
```

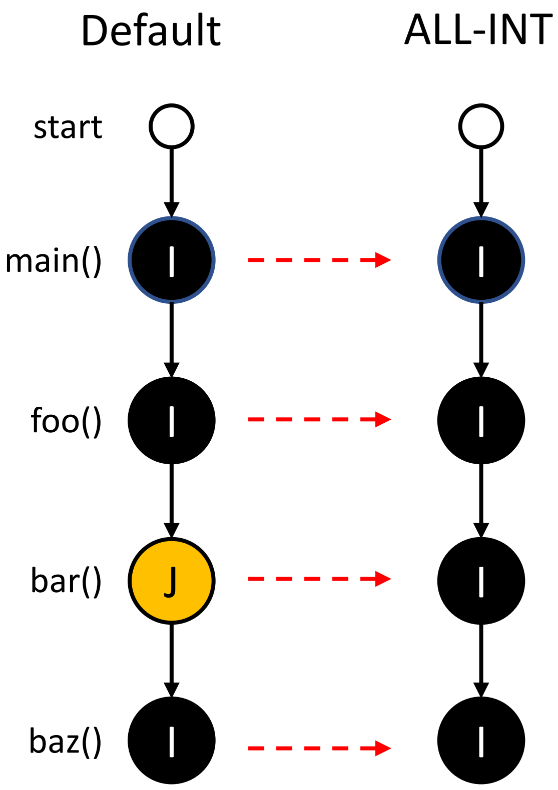


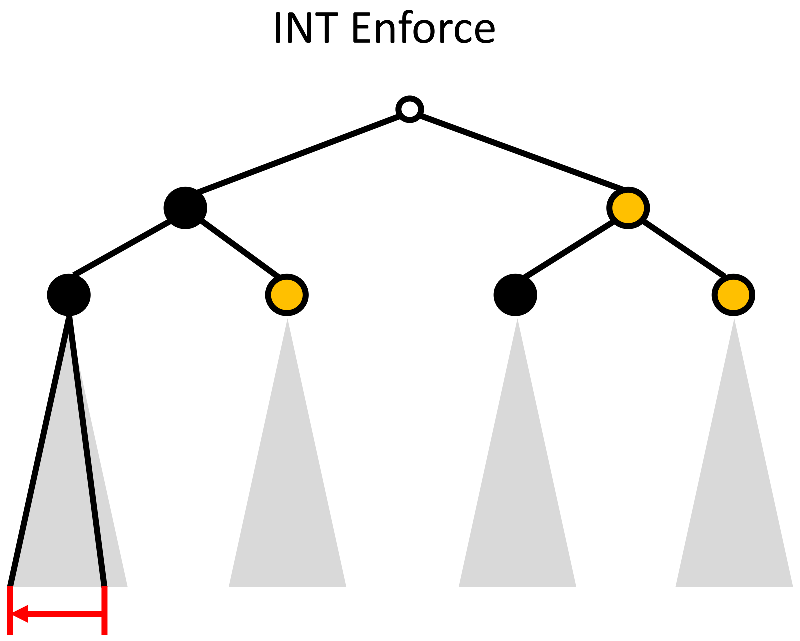


Compilation Space and JIT-Choices

- INT Enforce: Pull every method call to be in INT state

```
int baz() { 1+1 }

int bar() { baz() }

int foo() { bar() }

void main() {
  print(foo())
}
```

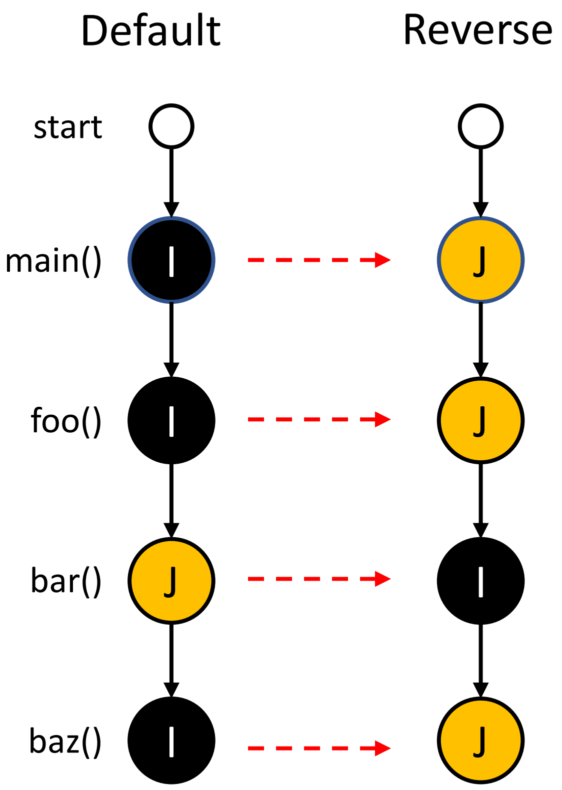


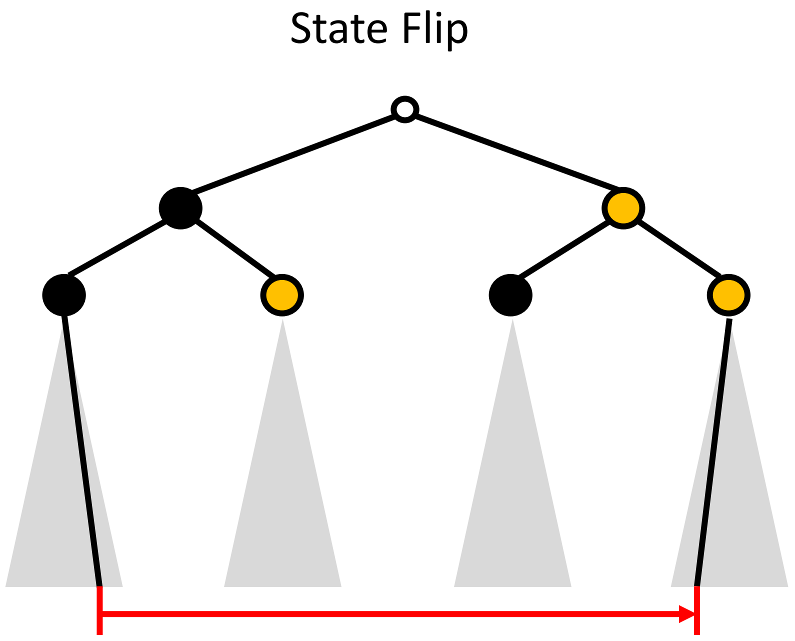


Compilation Space and JIT-Choices

- State Flip: Flip every method call to be the other state

```
int baz() { 1+1 }

int bar() { baz() }

int foo() { bar() }

void main() {
  print(foo())
}
```

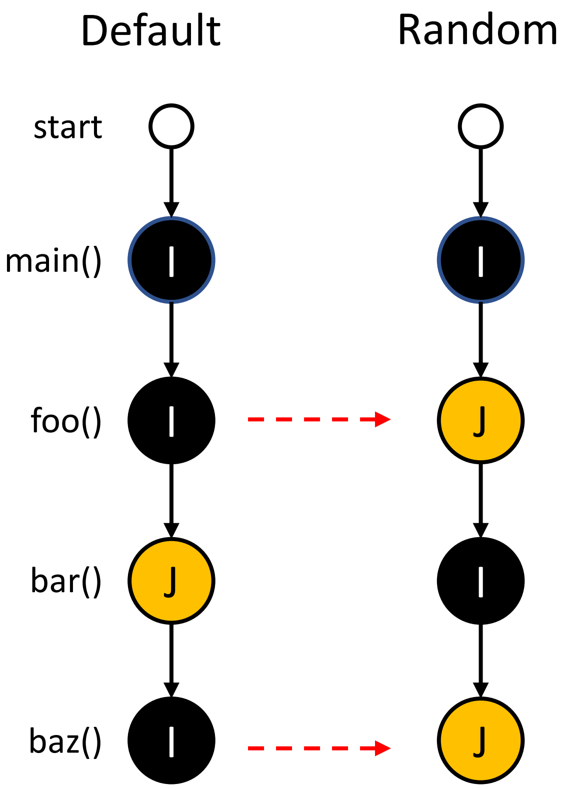


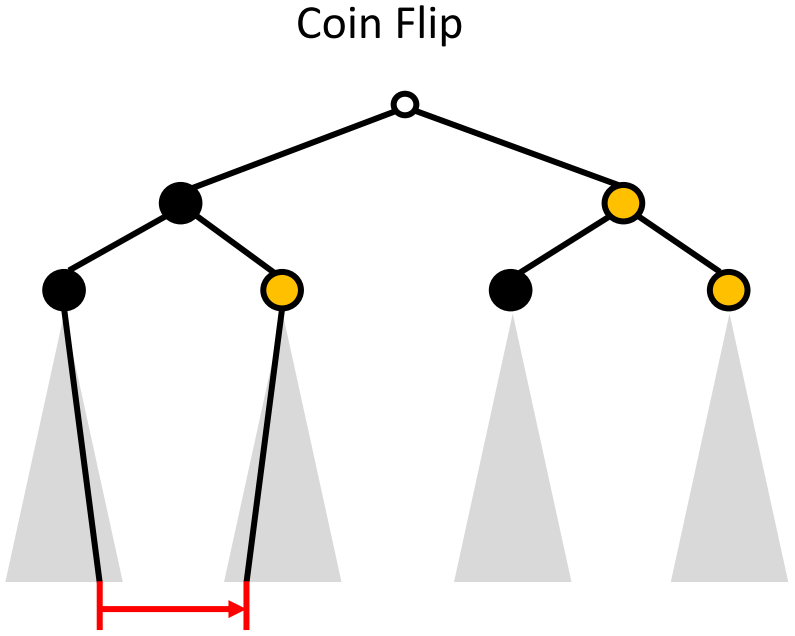


Compilation Space and JIT-Choices

- Coin Flip: Flip some random method calls to the other state

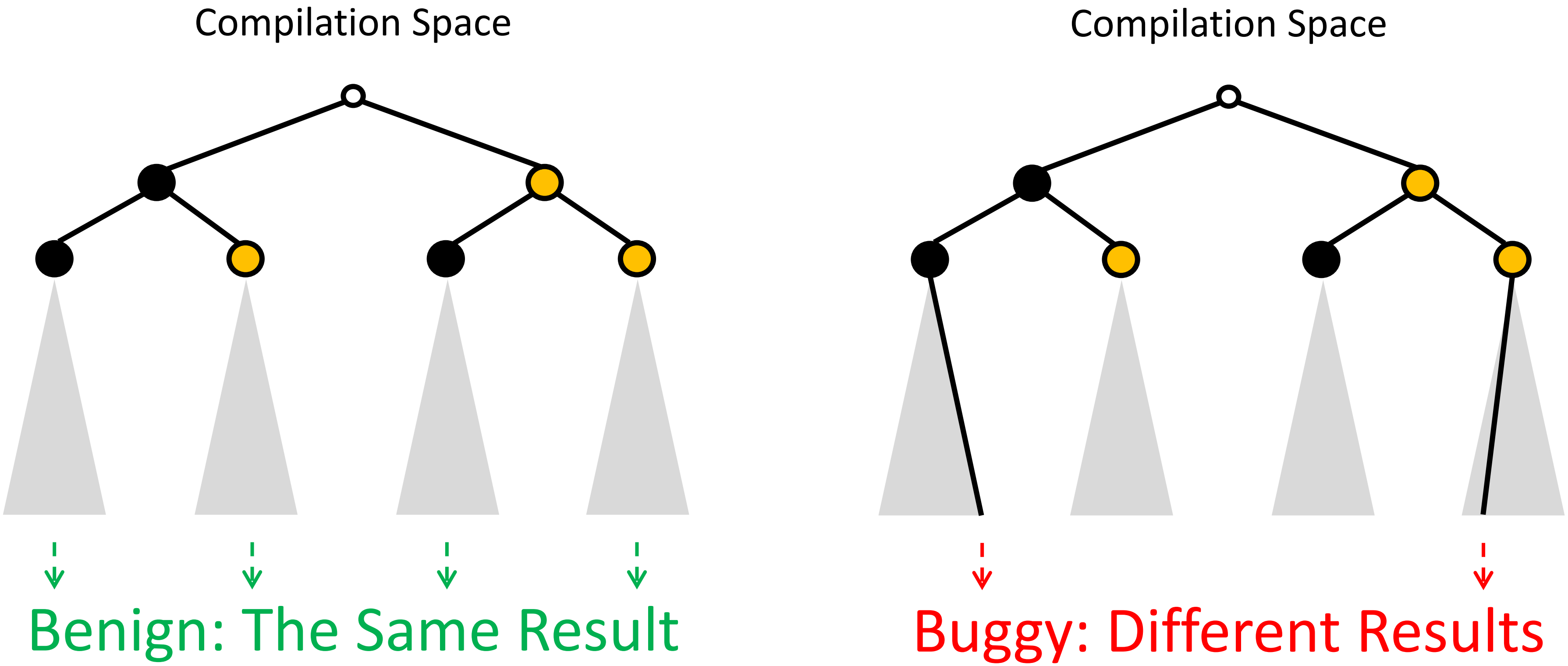
```
int baz() { 1+1 }

int bar() { baz() }

int foo() { bar() }

void main() {
  print(foo())
}
```





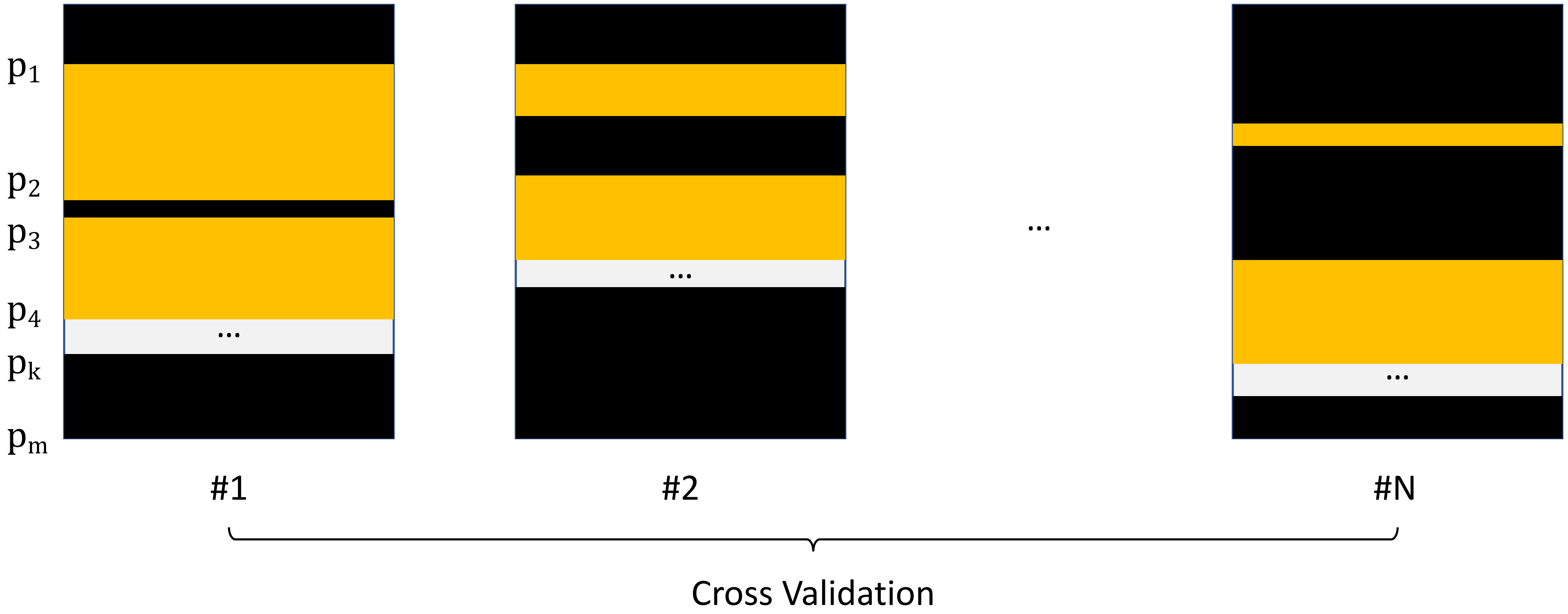
Compilation Space Exploration (CSE)

- Ideally: cross-validate the result of every JIT-choice for each program



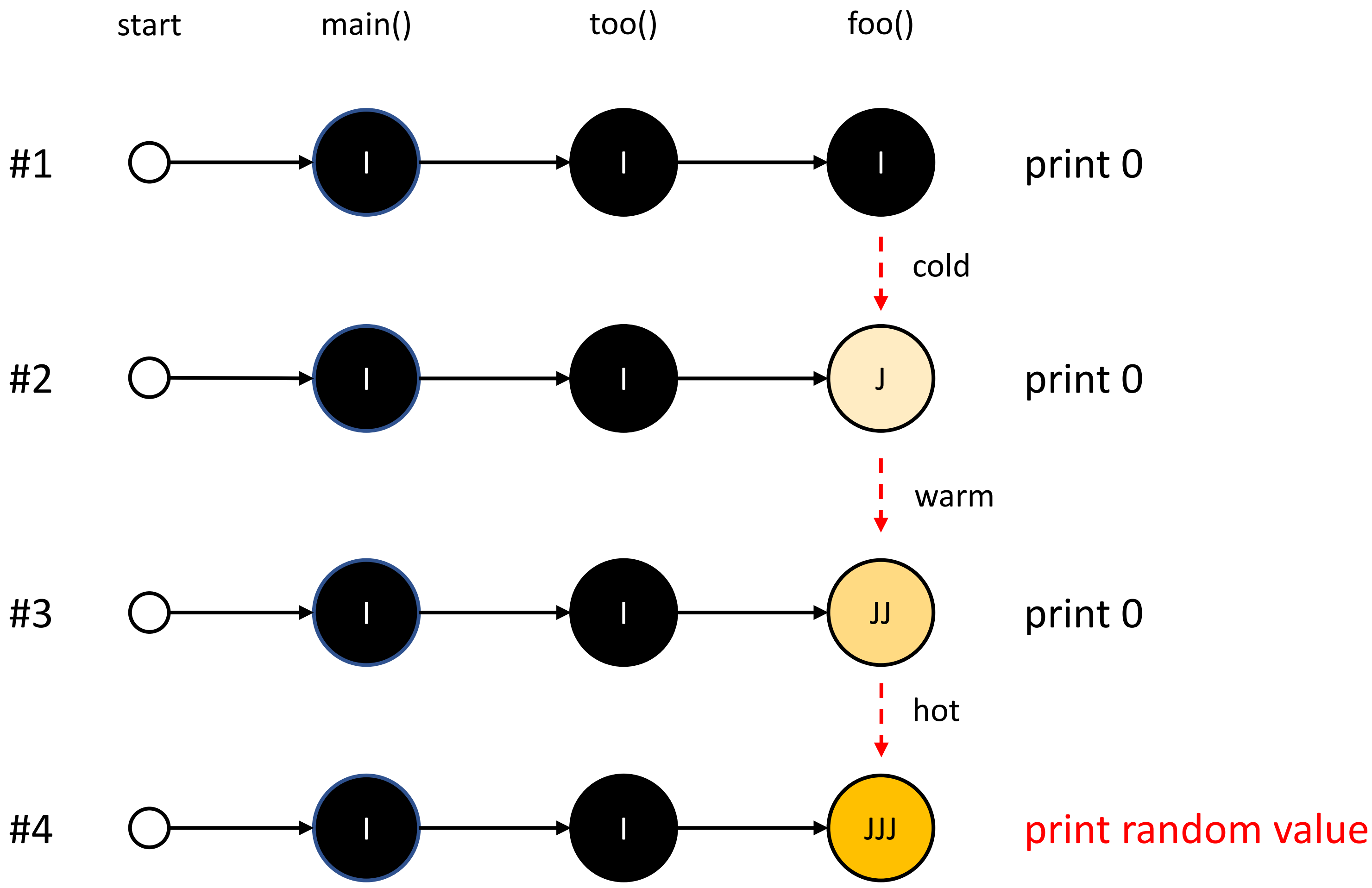
Small Explanation Hypothesis [Yanyan@Chinasoft22]

- Simple explanation of each choice: run the program to program point p_1 by interpretation, then to p_2 with JIT compilation, then p_3 …



Mis-Compilation Example – OpenJ9

```java
import java.net.Socket;

class T {
  int a;
  long z;

  void foo(boolean b, int c) {
    c *= --c;
    Socket t = new Socket();
    String[][] q = {};
    String p = "-000-0";
    for (int k = 395; k < 5172; k += 1) {
      z += c;
      try {
        if (q[a][1].equals(p)) {}
      } catch (Throwable x) {
      } finally {
      }
    }
  }

  void too(int i, int k) {
    int j = 1;
    while (++j < 104) foo(true, k);
  }

  public static void main(String[] g) {
    T t = new T();
    t.too(t.a, t.a);
    System.out.println(t.z);
  }
}
```



How to Realize CSE ?

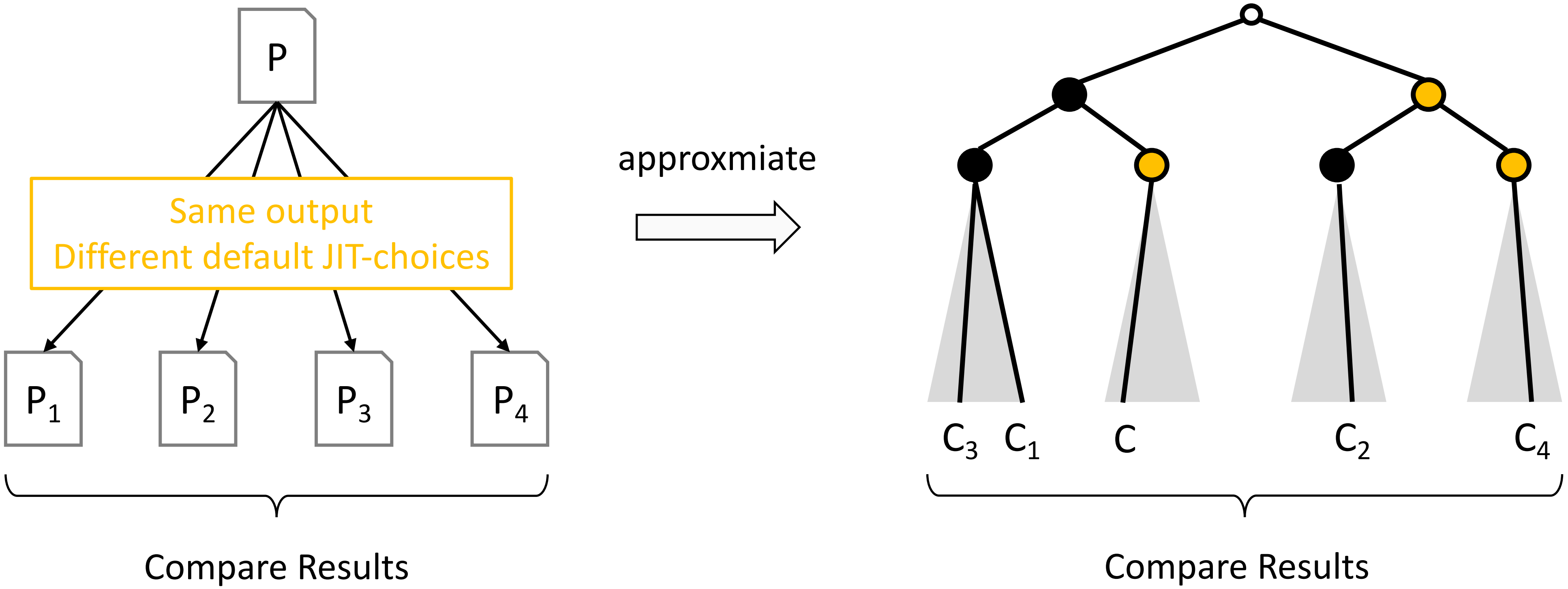
Two Obvious Options

Practical | LVM-agnostic
Expertise | LVM-specific

- Modify LVM implementations **Cumbersome** **Lightweight**
  - Pros: Generate as any JIT-choice as we like
  - Cons: Considerable VM-specific, manual, expertise effort, technically
    - Developers don't buy, at all
- Levarage JIT compiler-related options **Limited** **Powerful**
  - Pros: Easy to implement (“-XCompileCommand”, “-XX:+DeoptimizeRandom”, “-Xjit”)
  - Cons: Limited options, and incomplete exploration
    - Fair amount of VM-specific options understanding

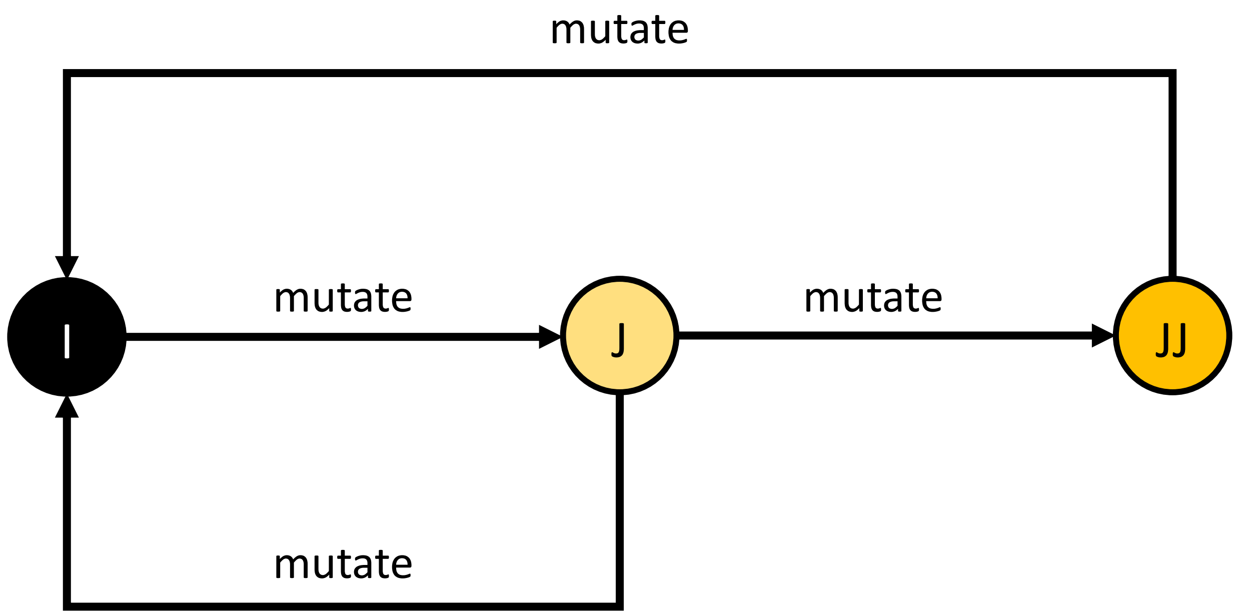
Semantics-Preserving Mutations: Approximating CSE from Source-Level

- Approximation: P with different choices by {P_1, P_2, …} with default choices



INT ↔ JITs Semantics-Preserving Mutations

- Mutate code blocks from INT to JITs or from JITs to INT or within JITs

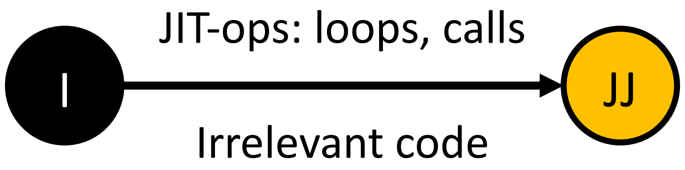


JIT-Op Neutral Mutation

- Mutate with help of JIT-relevant operations (code structs)
  - INT to JITs: loops and method calls
  - JITs to INT: various unexpected conditions
- Insert irrelevant, synthetic code that extensively reuse existing variables to avoid being optimized out by the JIT compilers

```
int zero() { return 0 }

void main() {
 // synthetic code
 for i in 1...1000 {
  x += zero() // <--- I to J
 }
 // synthetic code
}
```

hot



JIT-Op Neutral Mutators (in This Work)

- This work focuses mainly on INT to JITs (method calls, and loops)

Loop Inserter

```
for (...) {
 // synthetic code
}
```

Statement Wrapper

```
for (...) {
 // synthetic code
 if (exec) {
  <<wrap_stmt>>
  exec = false
 }
 // synthetic code
}
```

semantics preserving code

Method Invoker

```
for (...) {
 precall = true
 <<wrap_method>>()
 precall = false
}
```

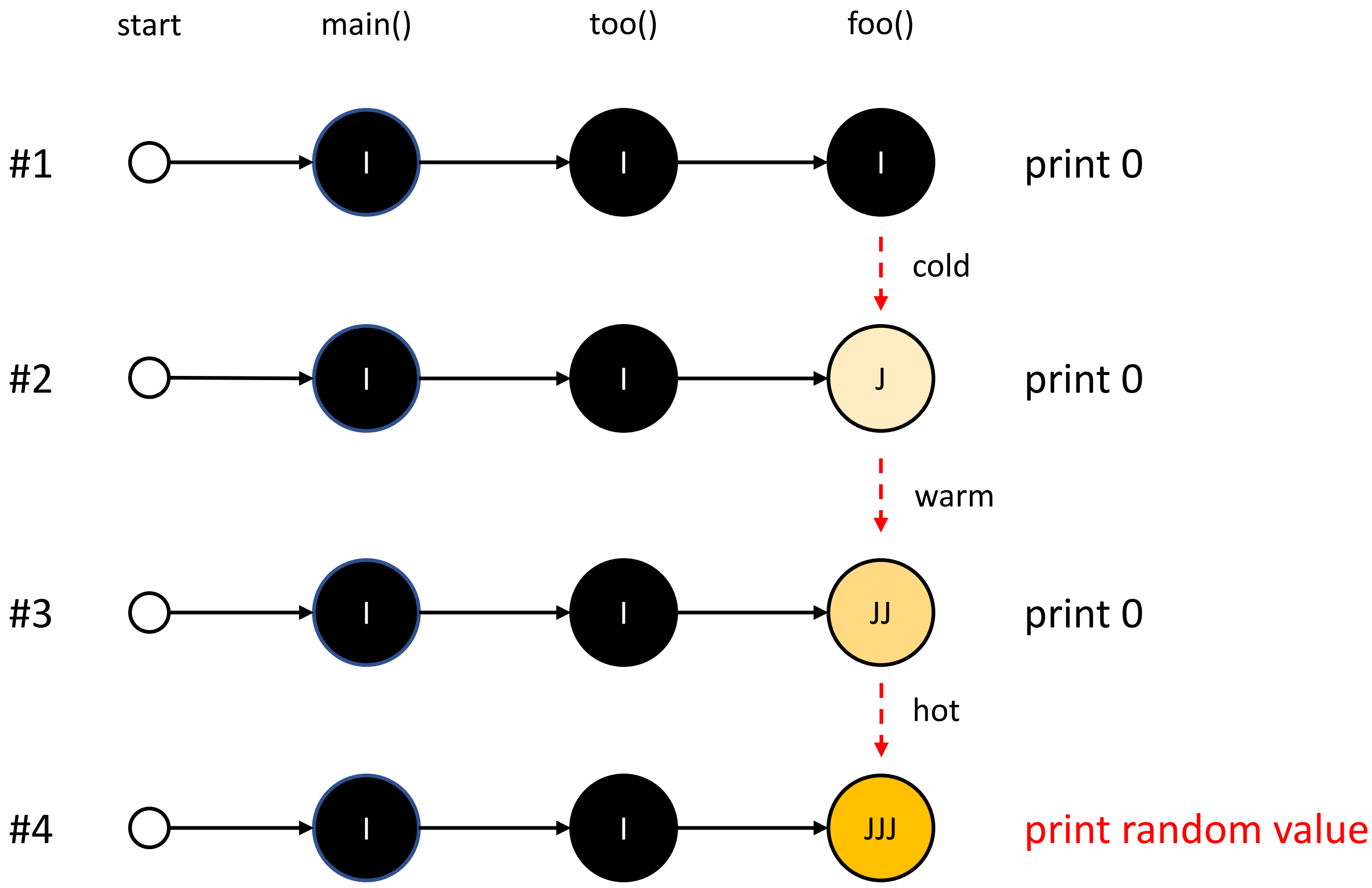
semantics preserving code

```
<<wrap_method>>() {
 if (precall) {
  // synthetic code
  return
 }
 // orig. method body
}
```

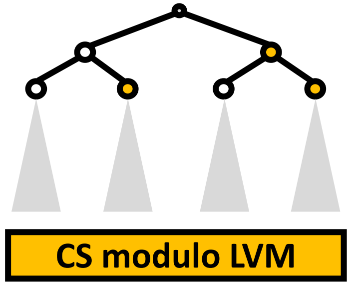
semantics preserving code

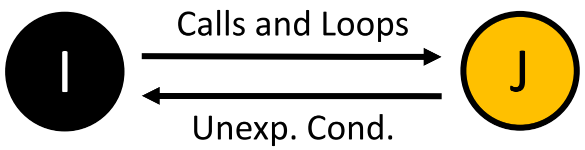
Mis-Compilation Example – OpenJ9

```java
import java.net.Socket;

class T {
  int a;
  long z;

  void foo(boolean b, int c) {
    c *= --c;
    Socket t = new Socket();
    String[][] q = {};
    String p = "-000-0";
    for (int k = 395; k < 5172; k += 1) {
      z += c;
      try {
        if (q[a][1].equals(p)) {}
      } catch (Throwable x) {
      } finally {
      }
    }
  }

  void too(int i, int k) {
    int j = 1;
    while (++j < 104) foo(true, k);
  }

  public static void main(String[] g) {
    T t = new T();
    t.too(t.a, t.a);
    System.out.println(t.z);
  }
}
```



Summary





JIT-Op Neutral Mutation

- Compilation Space modulo LVM: Resolve the oracle problem in testing JIT compilers of modern LVMs
- Compilation Space Exploration: Thoroughly explore the compilation space and cross-validate the resulting program output
- JIT-op Neutral Mutation: Approximate CSE from source-code level by semantics-preserving mutations with JIT-ops
- Artemis: JoNM implementation specifically for JVMs

Limitations

- Limited JITs to INT support
  - Avoid unexpected conditions from being optimized out
  - Difficult to find unexpected conditions that work across many JVMs
- Limited type support: do not support fp32 and fp64
- Do not support concurrent code

Unleashing the Power of Compilation Space

- **Coverage-guided mutation**: guide mutation by the coverage of the compilation space
- **Efficient exploration**: interesting JIT-choices gain high priorities
- **Whitebox integration**: combine with JIT options or profiling data
- **Novel mutators**: coin novel neutral mutators targeting JITs to INT
- …

Thanks

Artemis

Cong Li

ICS (NJU)

AST (ETHz)