Fixing the see-saw:

Enabling Precise Optimizations in JIT Compilers, Efficiently

Manas Thakur

CSE, IIT Bombay

May 26th, 2026

IIT Bombay

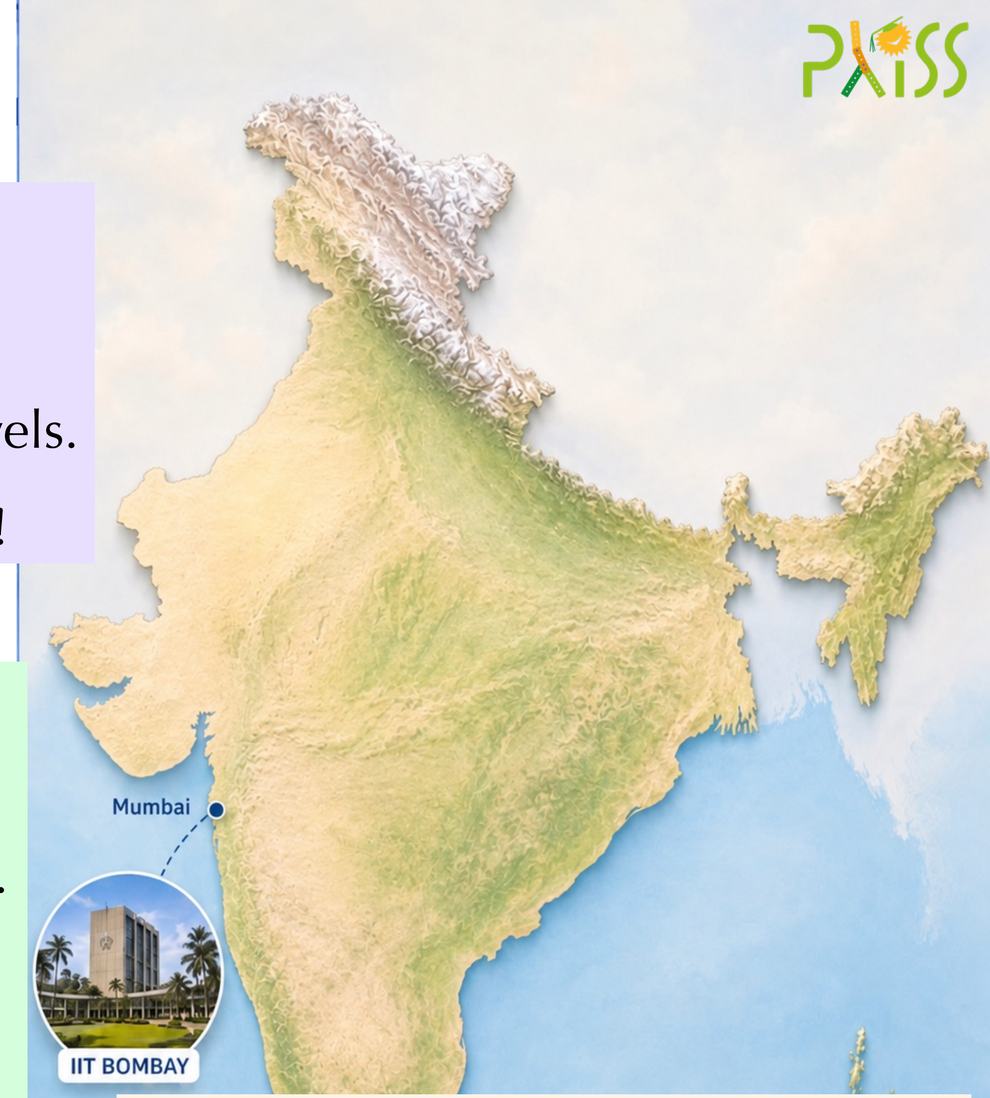
I am another You

Also a photographer: 📸

- Forte: nature and street.
- Write blogs about my travels.
- Join me around Bertinoro!

A JIT Researcher:

- Love reasoning about programs and improving their performance.
- Played with HotSpot, OpenJ9, GraalVM, Ř, SpiderMonkey.
- Proponent of combining AOT and JIT analysis.

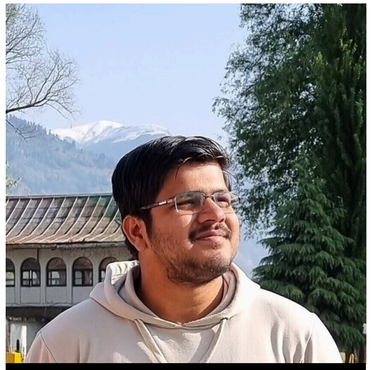


Academic since 7 years:

- CompL group (PLATO lab).
- Popular OO and functional PL courses across degree programs.

Manas Thakur

Content Credits



Aditya Anand, IITB

Henry Zongaro, IBM

Vijay Sundaresan, IBM

Daryl Maier, IBM

I designed a **COOOL** course

➤ Fundamentals (OO abstraction; memory organization).

➤ Heap analysis (points-to information; field- and flow-sensitivity; intra- and inter-procedural analysis; alias analysis; heap cloning).

➤ Optimizations involving allocation, field access and deallocation (pointer and escape analysis; stack allocation; scalar replacement; field privatization; value types and object inlining; object colocation; control-sensitive analysis; garbage collection).

➤ Optimizations involving method dispatch (class hierarchy analysis; rapid type analysis; context-insensitive analysis; object- and type-sensitive analysis).

➤ Speculative optimizations (speculative type resolution; speculative method inlining and polymorphic inline caching; code versioning; deoptimization overheads).

➤ Recent developments (closed-world assumption; dynamic features; mixing AOT and JIT).

20,000,000

50,000

OO Optimizations in JIT Compilers

Two trends dominate modern software systems:

1. Object-oriented programming
   - Java, C#, Scala
   - JavaScript, Python, R

2. Just-in-time compilation
   - JVM languages (Java, Kotlin, Scala, Clojure)
   - .NET ecosystem (C#)
   - JavaScript engines (V8, SpiderMonkey, Webkit)
   - Python (PyPy), R (FastR, Ř)

Our concern: How does a JIT compiler ensure efficient memory behavior in real-world object-oriented applications?

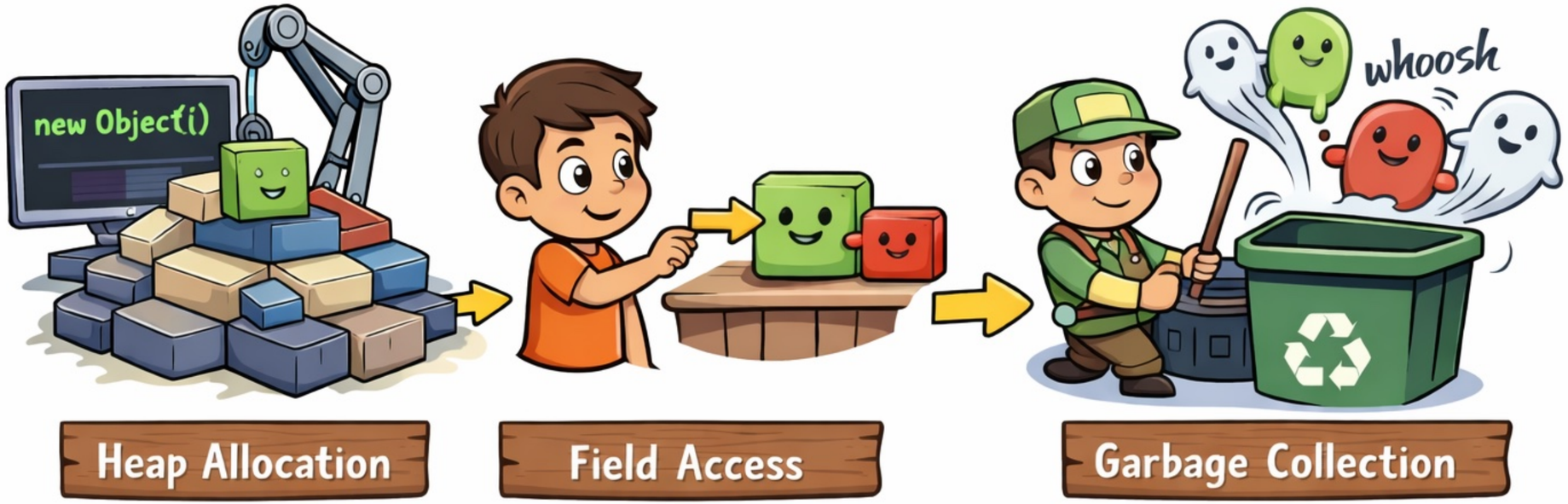
What's special about Java

- Platform independence
  - Write on your laptop, run everywhere (phones, desktops, even clouds)
- Strong static typing
  - Invalid operations are disallowed by the compiler (happy users)
- Exceptions with stack traces instead of shocking segfaults
  - Developers often don't need to beat their heads with gdb and printfs (happy again)
- Automatic memory deallocation (*garbage collection*)
  - Everyone knows the liberation it gives to programmers!

What's problematic about Java

- Platform independence
  - Bytecode is high level; javac cannot perform machine-dependent optimizations
- Strong static typing
  - Impossible to directly talk to the memory and write faster programs
- Exceptions with stack traces instead of shocking segfaults
  - Symbol+State tracking at run-time; overhead again
- Automatic memory deallocation (*garbage collection*)
  - Program pauses; even threaded GC snatches resources from the program

The Story of an Object



Object Allocation

- Objects allocated using `new` in Java are stored on the heap.
- The only way of allocating objects in Java is using `new`.

Java code:

```
x = new ArrayList<T>();
```

Object layout:

```
OFF  SZ                   TYPE DESCRIPTION
  0   8                        (object header: mark)
  8   4                        (object header: class)
 12   4                    int AbstractList.modCount
 16   4                    int ArrayList.size
 20   4   java.lang.Object[] ArrayList.elementData
Instance size: 24 bytes
```

Bytecode:

```
0: new           #7                  // class java/util/ArrayList
3: dup
4: invokespecial #9                  // Method java/util/ArrayList."<init>":()V
7: astore_1
```

Field Access

- Each field access requires a memory load.
- Field accesses are often wrapped inside getter calls.
- Reference fields may load objects from different parts of the heap, not necessarily resident in the same cache block.

```
class T { int y; ...}
...
x = new ArrayList<T>();
...
int y = ((T) x.get(k)).getY();
```

- Three memory loads (cache misses) if different objects are on different pages (blocks).

Garbage Collection

```
a = new T();    // O1
a.f = new T(); // O2
foo(a);
a.f = b;
// Can we now collect O1 and/or O2?
```

- If an object is not reachable from any reference any more, then it can be GCed.
- Even if it is reachable but the reference is not used, then also it can be GCed.
- Knowing this requires:
  - Maintaining reachability at run-time.
  - Liveness analysis.
  - Interprocedural analysis (because `foo` may create newer references to `O1` and/or `O2`).
- Even concurrent GC threads snatch resources from the program.

Lesser the work the GC has to do, better would be the performance.

Let's Optimize Object Allocation, Access, as well as Deallocation

Objects in Java

- Say an abstract object allocated at line `l` is denoted as O_l.
- Method `foo` allocates three objects O_4, O_5 and O_6 on the heap.
- An object reachable from the static field `g` is accessible outside its scope of allocation.

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

Heap versus Stack Allocation

- All the objects are by default allocated on the heap.
- Garbage collector clears out unused objects periodically.
  - Has its own overheads, even if run concurrently.
- Memory allocated on the activation record (stack frame) of a method gets freed automatically after method execution.
- Accesses on stack are more efficient than those on heap.
- Many objects are known to be short lived.

Can we allocate (some) objects on the stack?

Captured Objects [1]

- Consider string concatenation:

```
int i1 = ...;
String s1 = ...;
String s2 = ...;
System.out.println(i1 + s1 + s2);
```

The `StringBuilder` object is only used for the duration of this statement, and only within this method.

- Equivalent of the Java bytecode produced by javac:

```
new StringBuilder().append(i1).append(s1)
        .append(s2).toString();
```

Captured Objects [2]

- Consider iteration over an `ArrayList`:

```
int sum(ArrayList<Integer> alist) {
    int total = 0;
    for (Integer val : alist) {
        total += val.intValue();
    }
    return total;
}
```

Calls `ArrayList.iterator()`, which creates an `ArrayList$Itr` object used for the duration of the loop.

Captured Objects [3]

- Consider boxed primitives:

```
private HashMap<Integer, List<String>> dictionary =
                            new HashMap<Integer, List<String>>();

public List<String> lookupWordsOfLength(int len) {
   return dict.get(new Integer(len));
}
```

New `Integer` instance created simply to look for `HashMap` entry

Escaping Objects

- An object is said to escape its method of allocation if it can outlive the activation record of that method.

- This can happen in various scenarios:
  - Returned from allocating method
  - Stored into a parameter's field
  - Assigned to a static field
  - Assigned to a thread object's field
  - Passed to an unanalyzed method
  - Reachable from another escaping object

Let's understand with a few examples.

Escaping Objects [1]

- Object returned from a method:

```
private class ModularInt {
   private int val;
   private int modulus;
   public ModularInt(int v, int mod) {
      this.val = val % m; this.modulus = m;
   }
   public ModularInt successor() {
      return new ModularInt(val+1, modulus);
   }
}
```

`ModularInt` object created here is accessible in the caller.

Escapes!

Escaping Objects [2]

- Object stored into the field of a parameter:

```
public class Point {
   private double x; private double y;
   public Point(double x, double y) {
      this.x = x; this.y = y;
   }
}
...
public setLocation(double x, double y) {
   this.loc = new Point(x, y);
}
```

`Point` object created here can be accessed through the `loc` field after return from `setLocation`.

Escapes!

Escaping Objects [3]

- Object passed as an argument:

```
public class WordTally {
    private Map<String, Integer> map = new HashMap<>();
    public void bump() {
        int currTally = map.containsKey(s) ? map.get(s) : 0;
        map.put(s, new Integer(currTally+1));
    }
}
```

`Integer` object created in this invocation can be accessed on later invocations through `Map.get`.

Escapes!

Escape Analysis for Stack Allocation

- An object is said to escape its method of allocation if it can outlive the activation record of that method.
- Non-escaping (captured) objects can be allocated on the stack frame of the capturing method.
- Typically determined using **Escape Analysis**.

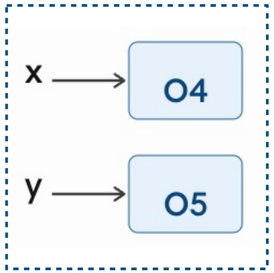
- Let us draw a **points-to graph** to represent points-to relationships:
  - Nodes represent variables and (abstract) objects.
  - Edges (directed) represent points-to relationships.

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

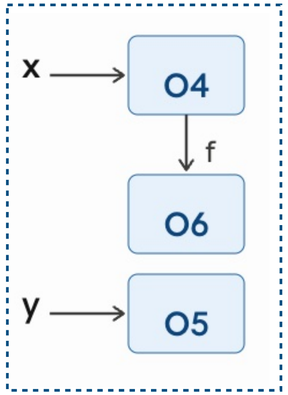
Escape Analysis using Points-to Information

Line 4:

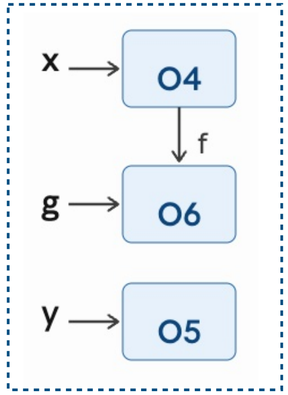
Line 5:



Line 6:



Line 7:

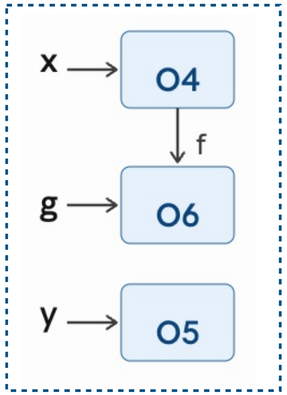



```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

Escape Analysis using Points-to Information

- An object is said to escape its method of allocation if it can outlive the activation record of that method.
- **Points-to sets after line 7:**
  - `x -> {O4}`
  - `y -> {O5}`
  - `O4.f -> {O6}`
  - `g -> {O6}`
- **Escape analysis results for `foo`:**
  - O_4 and O_5 do not escape.
  - O_6 escapes (at line `10`).

Line 7:




```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

Precise Escape Analysis: What does it need?

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

- Capturing O_5 requires an intraprocedural reachability analysis over points-to graph.
- Capturing O_4 requires an interprocedural analysis.
- Capturing O_6 until it escapes requires a flow-sensitive interprocedural analysis.

Variants of Stack Allocation

```
1: class C {
2:     static D g;
3:     int foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
8:         return (10+y.h);
9:     }
10:     void bar(D p) {
11:        g = p.f;
12:    }
13:}
14:class D { D f; int h; }
```

- O_4 can be allocated on `foo`'s stack frame.
- But O_5 only exists for its field `h` to be accessed via `y.h` (line 8).
- Hence we can get rid of O_5 altogether, and replace its field(s) with primitive variables.

- An optimization called **scalar replacement**.
- Possible when the program does not need to access a stack-allocatable object's header.

aka **Discontiguous Stack Allocation**

Representative Optimized Code

```
1: class C {
2:     static D g;
3:     int foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
8:         return (10+y.h);
9:     }
10:     void bar(D p) {
11:        g = p.f;
12:    }
13:}
14:class D { D f; int h; }
```

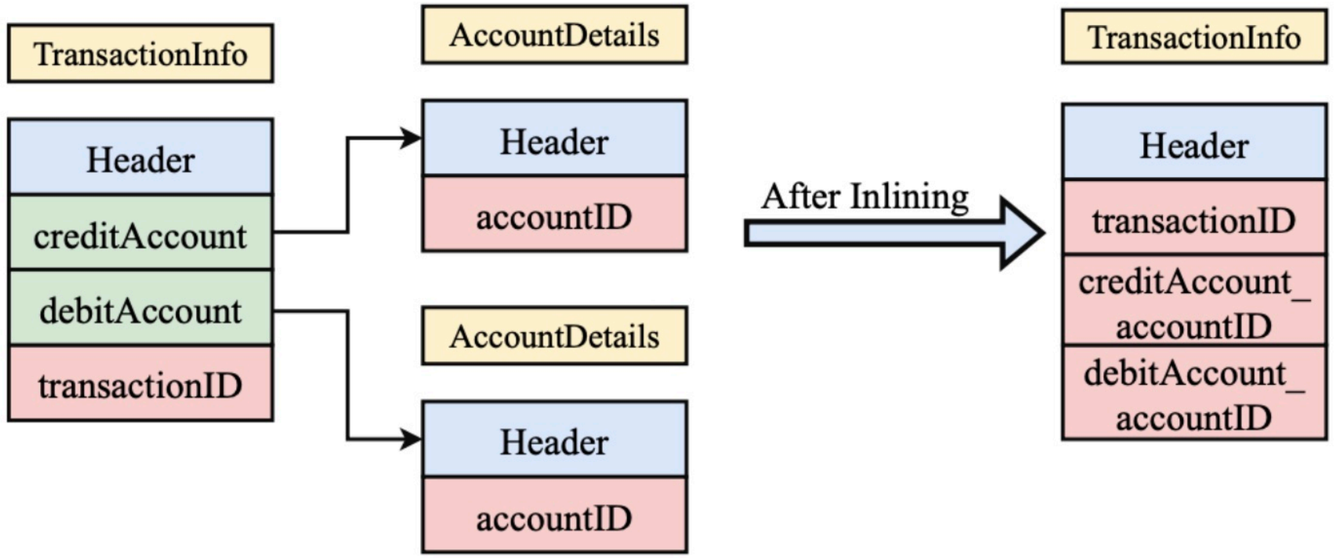
Post Escape Analysis

```
1: class C {
2:     static D g;
3:     int foo() {
4:         D x = stacknew D();//O4
5:         int y_h = 0;
6:         x.f = new D();//O6
7:         bar(x);
8:         return (10+y_h);
9:     }
10:     void bar(D p) {
11:        g = p.f;
12:    }
13:}
14:class D { D f; int h; }
```

- There also is *partial-escape analysis*, which can stack allocate O_6 until line 10, but let's consider it out-of-scope for today's discussion.

Is the story over for non-stack-allocatable objects?

- An object that is often accessed through a field of a “container”, can be allocated inside the container:



- An optimization called **object inlining**.
- Reduces memory footprint, eliminates field indirections, improves cache locality.

When can we (not) inline an object?

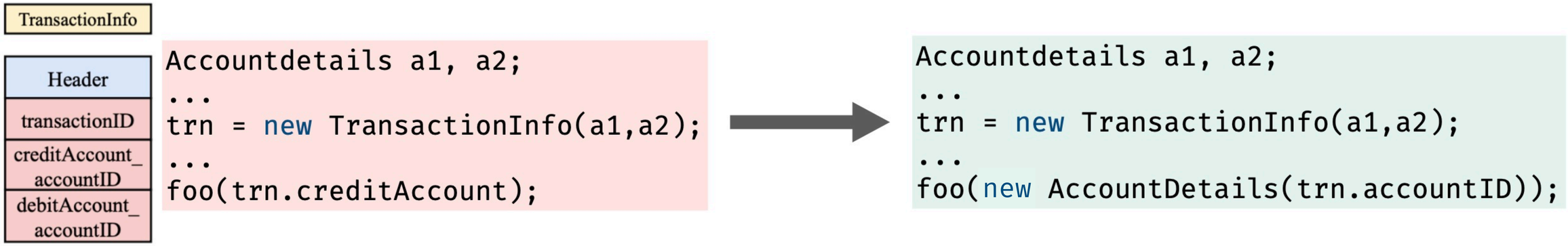
- When there are mutable references to the object from outside the container in which it is inlined.

- Unless we update those references to point inside the container ==> complicated and risky (may have to tweak with accessibility of the container, as well as disable the possibility of its stack allocation!).

- Java is bringing object inlining for **immutable value types**
  - All objects of value-type classes would be inlined inside all their containers, subject to a user-defined size threshold, during JIT compilation.

When *should* we (not) inline an object?

- When an inlined value-type object is needed in its entirety, it has to be *recreated:*



- Use *escape analysis* and identify inlined objects that need to be recreated more often than being accessed through container fields!

Arjun H Kumar, Bhavya Hirani, Hang Shao, Tobi Ajila, Vijay Sundaresan, Daryl Maier, and Manas Thakur. "VFlatten: Selective Value-Object Flattening using Hybrid Static and Dynamic Analysis". CGO 2026.

So sometimes we won't inline an object, but…

- The essential problem with inlining objects is that it destroys their header.
- The benefits included reduced memory, fewer indirections, and better cache locality.

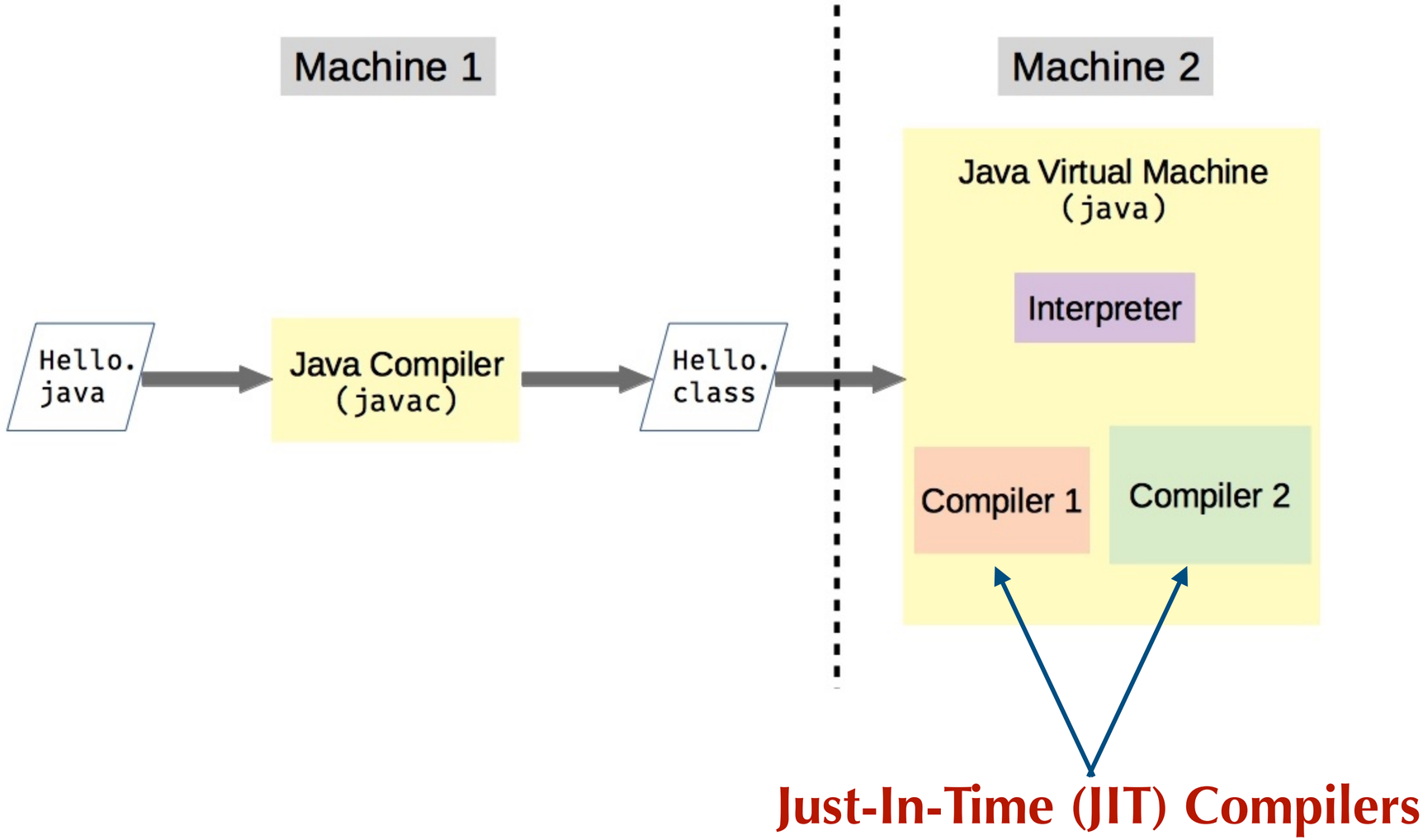
- *But we can still achieve the last two without destroying the header!*

- We can allocate a value-type object near (or next to) its container.
- An optimization called **object colocation**.

Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. "Compiler-Assisted Object Inlining with Value Fields". PLDI 2021.

Where does this all happen for Java programs?

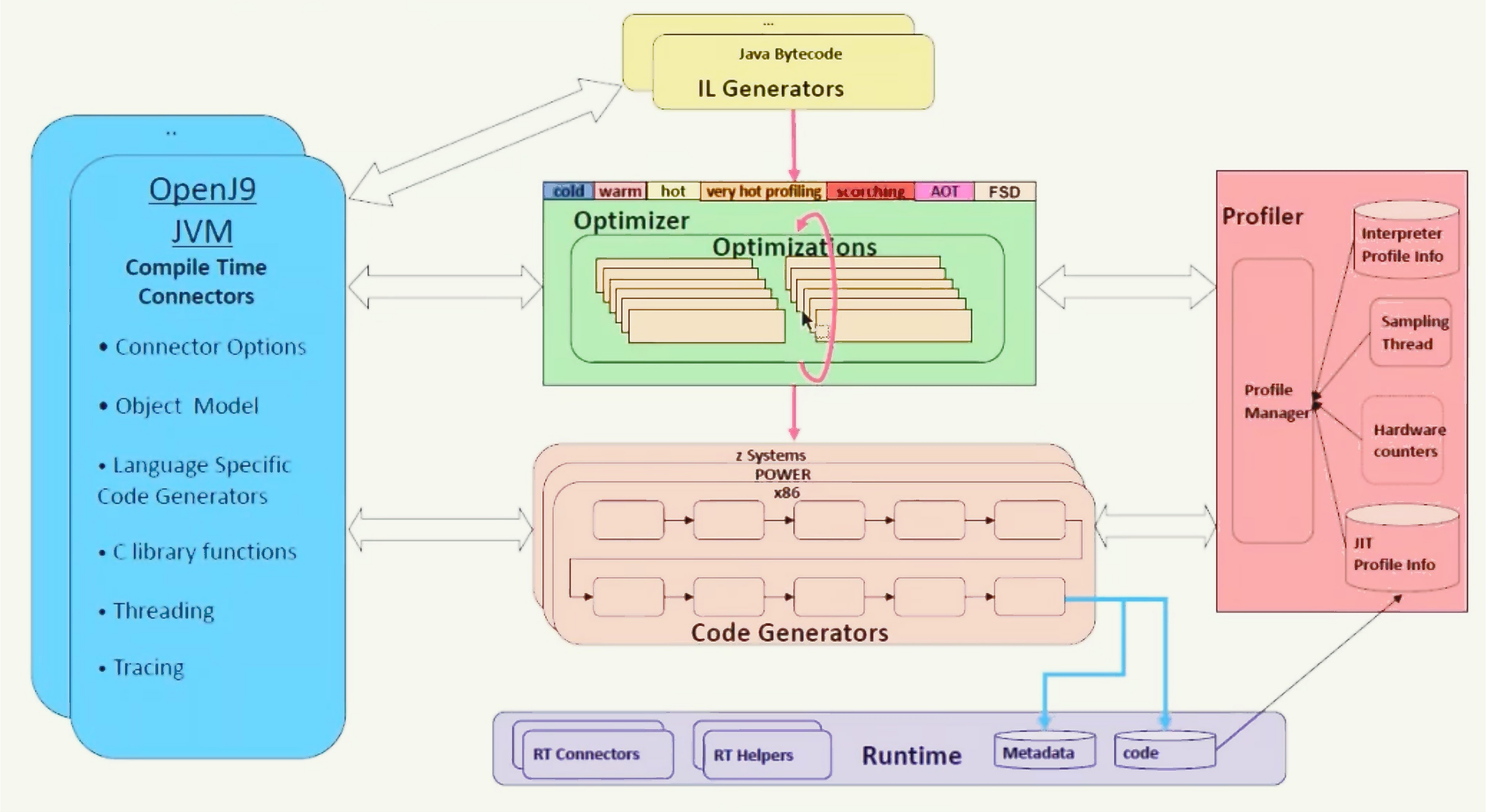
Java Virtual Machines



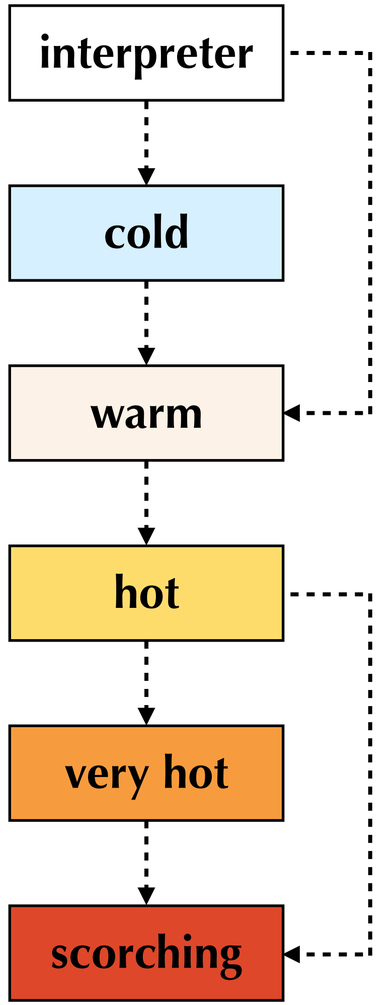
Typical JIT Pipeline

- Starts off with interpretation of bytecodes
- **Hot spots** get identified by *profiling*:
  - Method invocation counts
  - Backedge counts
- Identified code regions are inserted into a *compilation queue*
- Compiler threads compile methods in the background, while bytecode interpretation continues
- Entry points of methods changed dynamically
- Hot loops replaced *on-the-stack*

Inside a real JVM: Eclipse OpenJ9

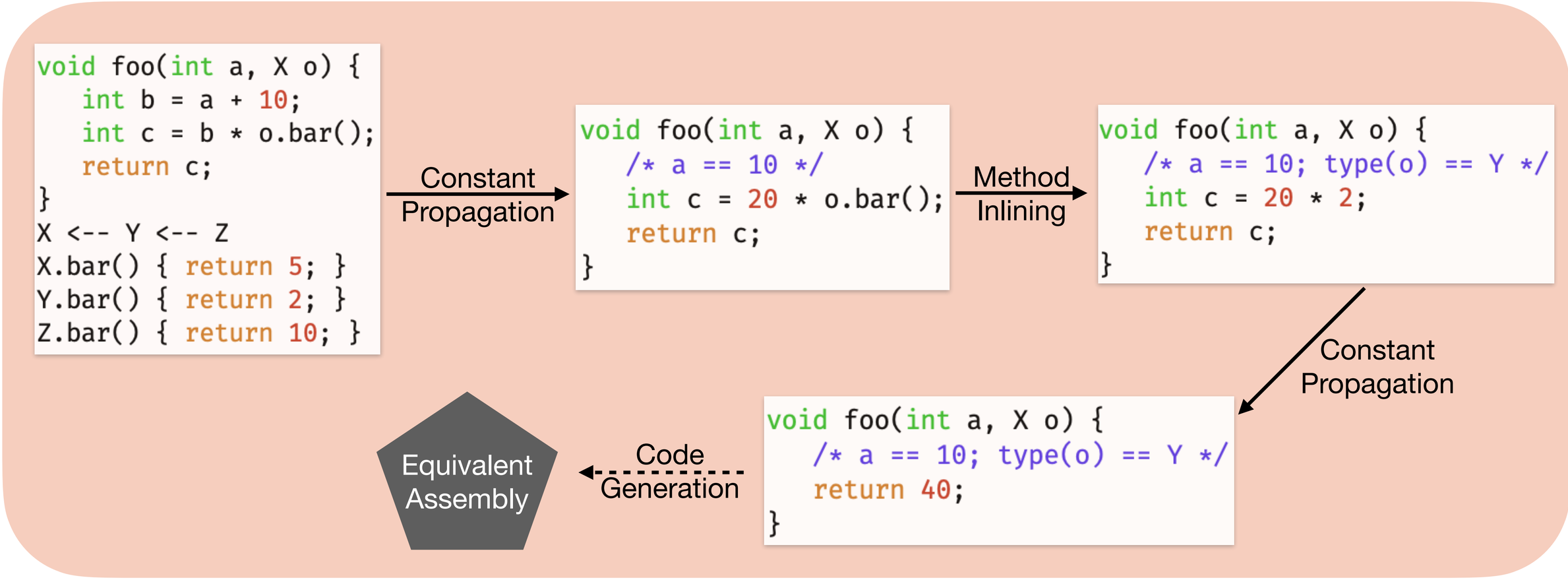


Tiered Compilation Model



- Tiered compilation allows the same method to be compiled multiple times with increasing levels of optimization
- Methods start out running bytecode directly (interpreter)
- After many invocations (or via *sampling*), code gets compiled at cold or warm levels
- Low overhead sampling thread is used to identify hot methods
- Subsequent level employs more optimizations
- Transition to scorching goes through a temporary profiling step

JITs shine due to **Speculation**

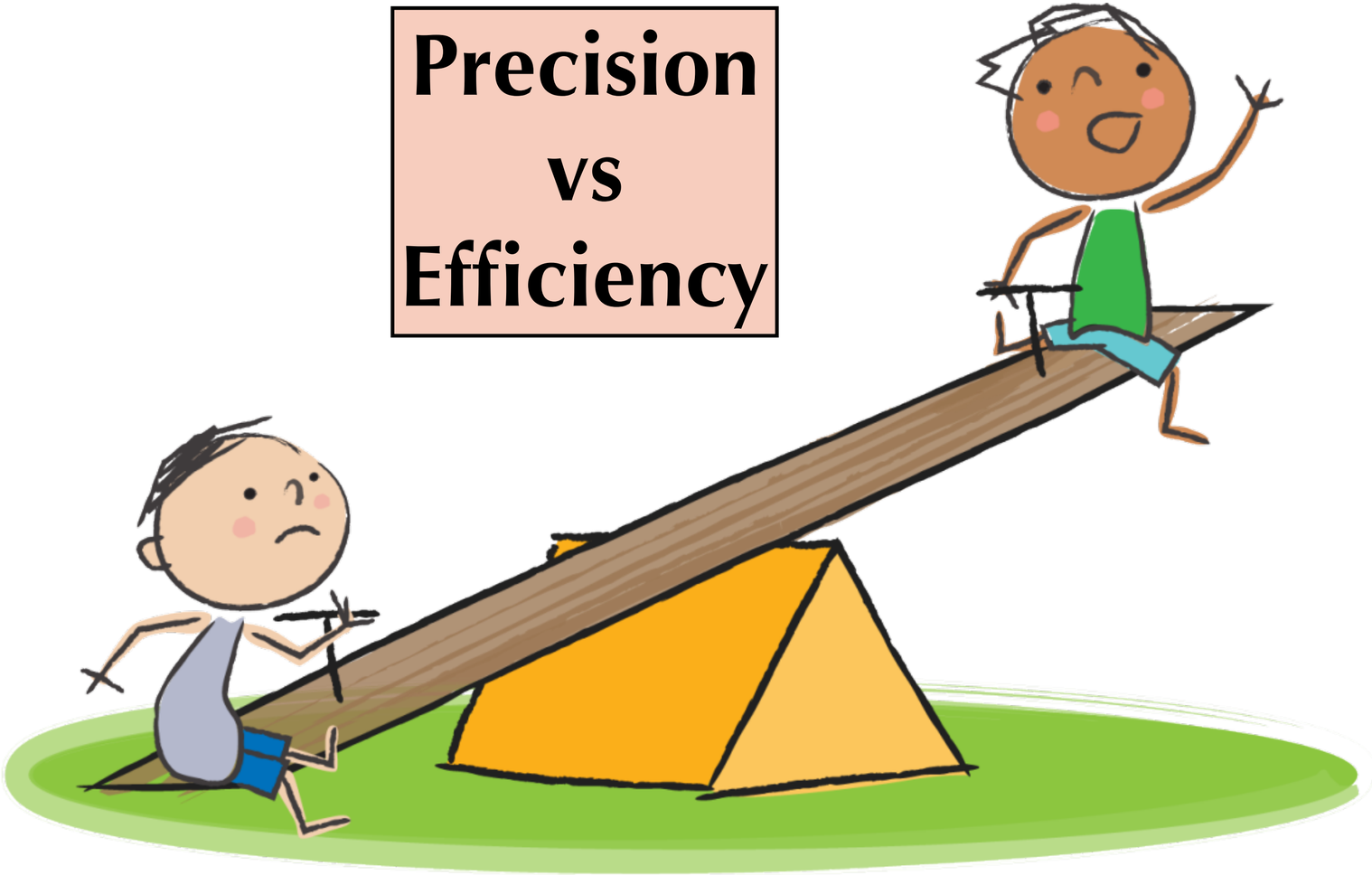


- JITs in managed runtimes can perform *speculative optimizations* based on run-time profiles, and support *deoptimization* when assumptions go wrong.

What's problematic about JITs

- Time spent in JIT compilation directly affects the execution time of the program.
- Speculation and deoptimization notwithstanding, most JITs perform very imprecise analyses and lose opportunities for optimization.
- Will `gcc -Oxyz` beat standard JVM invocation?
- How to bring decades of advancements in static program analysis to the JIT world?

What's problematic about JITs



Escape Analysis in Existing JIT compilers

- **BIG trouble:** Time spent in program analysis adds up to the execution time of the program.
- Typical JIT compilers can afford only imprecise (e.g., intraprocedural flow-insensitive) analyses.
- Inlining helps, but is not always possible.
- HotSpot's and OpenJ9's JIT compilers can stack allocate O_5, and sometimes O_4.

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D();//O4
5:         D y = new D();//O5
6:         x.f = new D();//O6
7:         bar(x);
           ...
8:     }
9:     void bar(D p) {
10:        g = p.f;
11:    }
12:}
13:class D { D f; }
```

Fixing the see-saw:

Enabling Precise Optimizations in JIT Compilers, Efficiently

Manas Thakur

CSE, IIT Bombay

May 28th, 2026

IIT Bombay

Recap: Static-Analysis Guided Dynamic Optimization

- JIT compilers do not have enough resources to perform deep interprocedural analysis, and hence end up being limited in the aggressiveness of resultant optimizations (e.g. stack allocation).
- Hence, we could:
  - Perform more precise* (flow-sensitive, interprocedural, context-sensitive?) escape analysis over Java Bytecode, say using the Soot analysis framework.
  - Store results either in separate files or as annotations in class files.
  - Use the obtained results to perform stack allocation of a (possibly) large number of objects during JIT compilation.
- **Let's see what all it takes.**

*Manas Thakur and V. Krishna Nandivada. "PYE: Precise and Efficient Analysis of Java Programs." TOPLAS/OOPSLA 2019.

The Library Problem

- The whole program may not be available statically.
- Even if it is, the libraries used (e.g. JCL) may differ across the analysis and the execution machines.
- Prior work on *partial program analysis* treats calls to such methods conservatively.
- Back to *most things escape!*

- **Idea:**
  - Use dependencies between available and unavailable parts of a program to derive and propagate **conditional values**, and generate **partial summaries** during static analysis.
  - Resolve conditions during JIT compilation (much faster than whole-program JIT analysis).

Partial Summaries

- For each method `m`,
  - **Traditional summary**, f_m `: D -> Val`

    For escape analysis,
    - Domain `D` includes object allocation sites, parameters, return value, etc.
    - Lattice of dataflow values `Val = {Escapes, DoesNotEscape}`, with `Escapes` being the conservative element.
  - **Partial summary**, g_m `: D -> P(CVal)`, where `CVal` is the set of conditional values.

Conditional Values

- A conditional value is a 3-tuple of the form $\tau = \langle \Theta, v, v' \rangle$

  where:

  - Θ is a pair $\langle u, x \rangle$, in which
    - u is a method
    - x is a program element in u

- For an element e in a method m, say $\tau = \langle\langle u, x \rangle, v, v' \rangle$ and $\tau \in g_m(e)$.
- τ evaluates to v' if $f_u(x) == v$.
- If u is a library method, τ cannot be evaluated statically.

So here's what we got

- τ evaluates to v' if $f_u(x)$ == v.

```
1: class B {
2:     static B global;
3:     void foo() {
4:         B r1 = new B(); //O4
5:         B r2 = new B(); //O5
6:         global = r2;
7:         r1.f = new B(); //O7
8:         List lst = new AList<>();
9:         lst.add(r1);
10:        B x = r1.f;
11:        synchronized(x) {...}
12:    }
13: }
```

$g_{foo}(O_4)$ = {<<AList.add,O_{p1}>,D,D>,

<<AList.add,O_{p1}>,E,E>}

// Shorthand: {<AList.add, O_{p1}>}

$g_{foo}(O_5)$ = {Θ_E}

$g_{foo}(O_7)$ = {<AList.add,O_{p1}>,
<AList.add,O_{p1}.f>}

$g_{foo}(O_{11})$ = {<AList.add,O_{p1}>,
<AList.add,O_{p1}.f>}

Evaluation of Conditional Values

```
1: class B {
2:     static B global;
3:     void foo() {
4:         B r1 = new B(); //O4
5:         B r2 = new B(); //O5
6:         global = r2;
7:         r1.f = new B(); //O7
8:         List lst = new AList<>();
9:         lst.add(r1);
10:        B x = r1.f;
11:        synchronized(x) {...}
12:    }
13: }
```

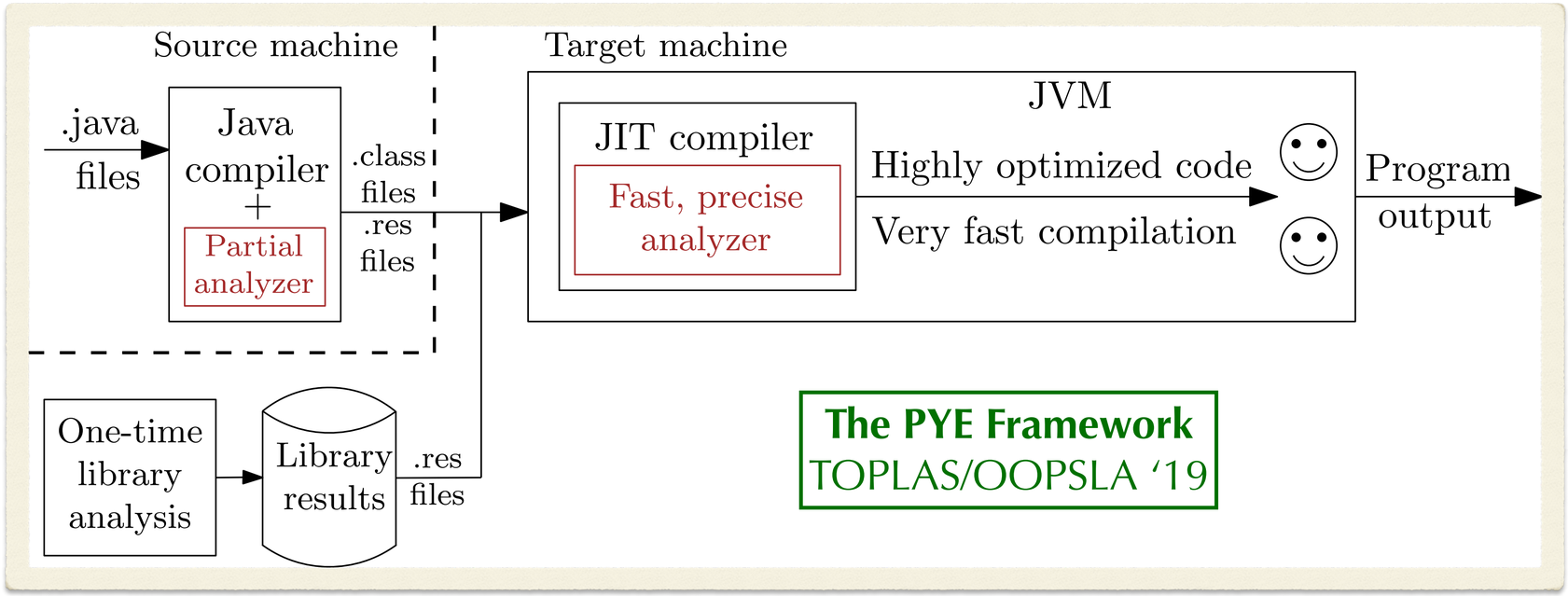
```
a: class AList<E> extends List<E> {
b:     void add(E elem) {
c:         arr[size++] = elem;
d:     }
e: }
```

$$g_{foo}(O_{11}) = \{<AList.add, O_{p1}>, <AList.add, O_{p1}.f>\}$$

$$g_{AList.add}(O_{p1}) = \{<AList.add, O_{a0}>, <AList.add, O_{a1}>\}$$

After solving the above equations, the synchronization at line 11 can be elided.

Static+Dynamic Analysis



- Implementation in Soot and HotSpot
- Shown to eliminate useless synchronization and null checks
- Applicable for many analyses (those that can be carried across AOT and JIT stages)

Correctness, Precision and Efficiency of Staging

- Staged static+dynamic analysis is the same as partial evaluation of dependencies among program elements (or objects, for escape analysis)!

- Thus:
  - It would produce the same result as a whole-program analysis performed during JIT.
  - The static (AOT) component is maximally efficient w.r.t. the statically resolvable dependencies.

- In fact, given a set of dependencies, the JIT component can be generated automatically.

Aditya Anand and Manas Thakur. "Partial Program Analysis for Staged Compilation Systems." SAS 2022; FMSD 2024.

"Welcome to the real world. It sucks. You're going to love it!"

Dynamism inhibits Static Analysis [DCL]

- Dynamic Class Loading:
  - Ability to load classes dynamically, as they are needed.

```
class A {
   void foo() {
      return 10;
   }
}
class Test {
   int bar(A x) {
      int y = 10 + x.foo();
      return y;
   }
   void m() {
      if (this.bar() == 0)
         launch();
   }
}
```

Not present statically:

```
class B extends A {
   void foo() {
      return -10;
   }
}
```

If class `B` gets loaded dynamically, statically inlining `foo` into `bar` (and possibly that `bar` into `m`) may go **wrong**.

Dynamism inhibits Static Analysis [HCR]

- Hot Code Replace
  - Ability to modify code in a live execution (usually in a debugging session)

```
class A {
   void foo() {
      return 10;
   }
}
class Test {
   int bar(A x) {
      int y = 10 + x.foo();
      return y;
   }
   void m() {
      if (this.bar() == 0)
         launch();
   }
}
```

Change using HCR:

```
class A {
   void foo() {
      return -10;
   }
}
```

If the user replaces method `foo`'s body, statically inlining `foo` into `bar` (and possibly that `bar` into `m`) may go **wrong**.

Dynamism inhibits Static Analysis [REFL]

- Reflection
  - Ability to read class/method names and create instances, based on run-time values.

```
interface Game { void play(); }
class Cricket implements Game {
   void play() {
      ...
   }
}
class Soccer implements Game {
   void play() {
      ...
   }
}
```

The static compiler may know very little (and sometimes nothing) about the call graph rooted at the call site `mtd.invoke(obj);`.

```
class M {
   void m(String c) throws Exception {
      Class<?> cls = Class.forName(c);
      Method mtd = cls.getDeclaredMethod("play", null);
      Object obj = cls.newInstance();
      mtd.invoke(obj); // obj is the receiver
   } }
```

Static+Dynamic Analysis with Dynamism

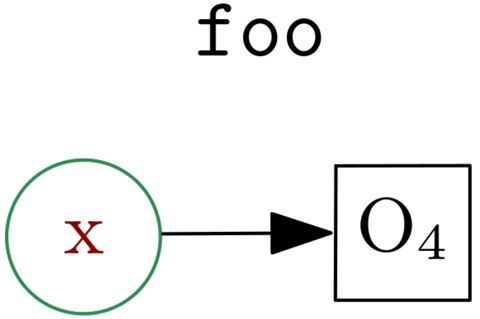
```
1. class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```

Static+Dynamic Analysis with Dynamism

```
1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.       A x = new A();  // O4
5.       A y = new A();  // O5
6.       x.f = new A();  // O6
7.       A p = x.f;
8.       bar(p, y);
9.       r.zar(p, q);
10.  } /* method foo */
```

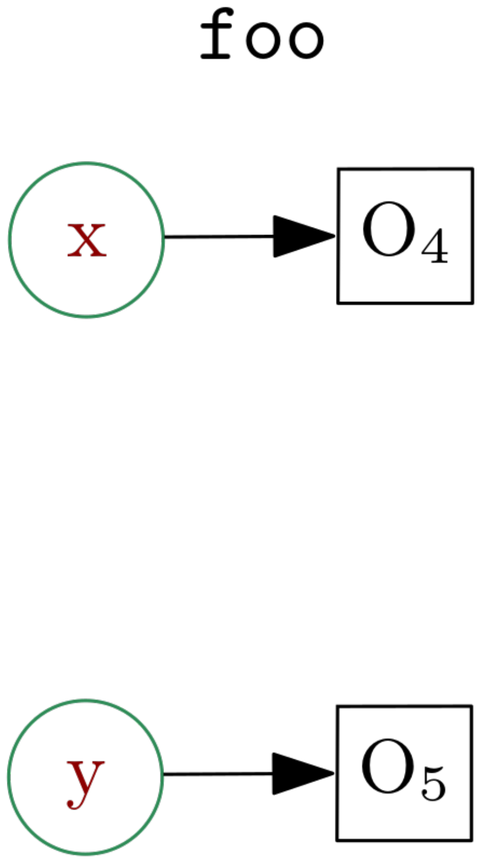
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Static+Dynamic Analysis with Dynamism

```
1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.       A x = new A();  // O4
5.       A y = new A();  // O5
6.       x.f = new A();  // O6
7.       A p = x.f;
8.       bar(p, y);
9.       r.zar(p, q);
10.  } /* method foo */
```

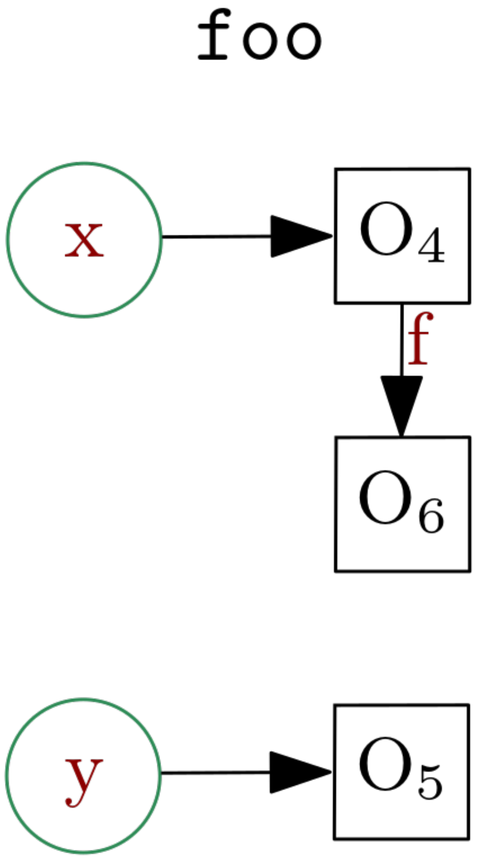
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Static+Dynamic Analysis with Dynamism

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

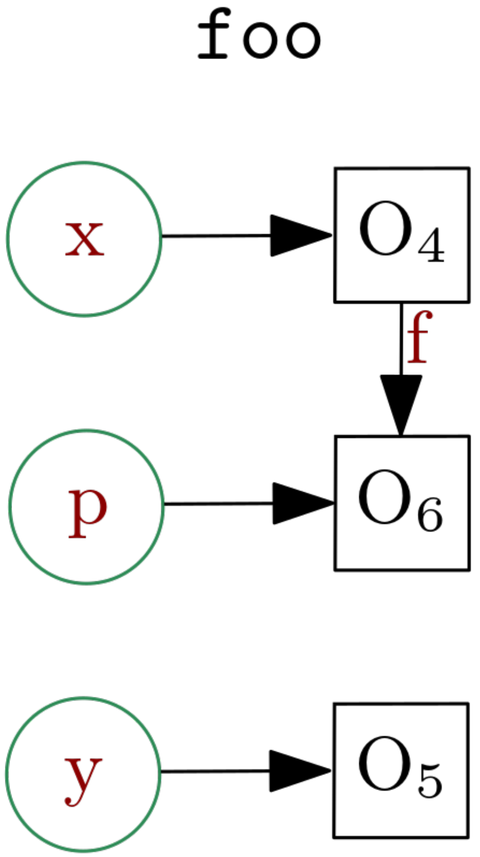
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Static+Dynamic Analysis with Dynamism

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

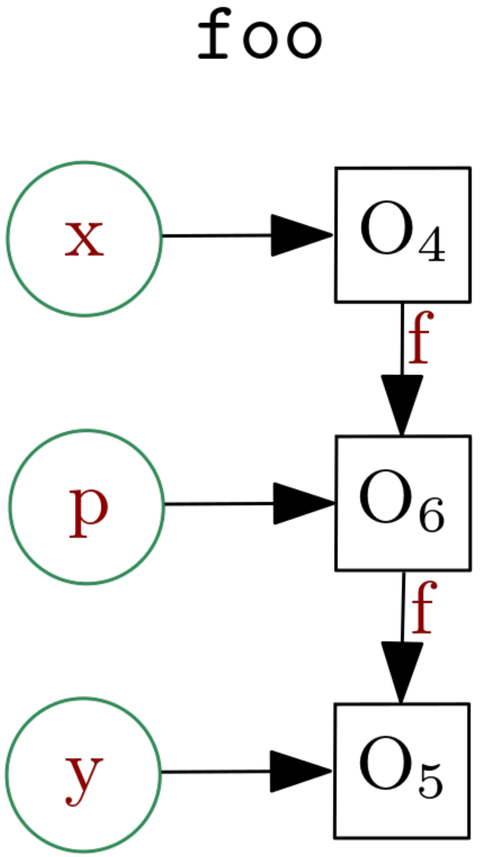
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Static+Dynamic Analysis with Dynamism

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```

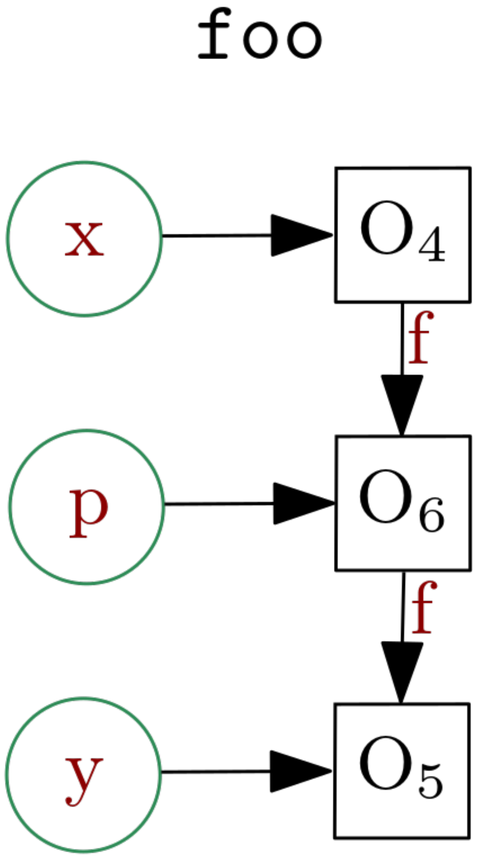


Static+Dynamic Analysis with Dynamism

```
1. class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

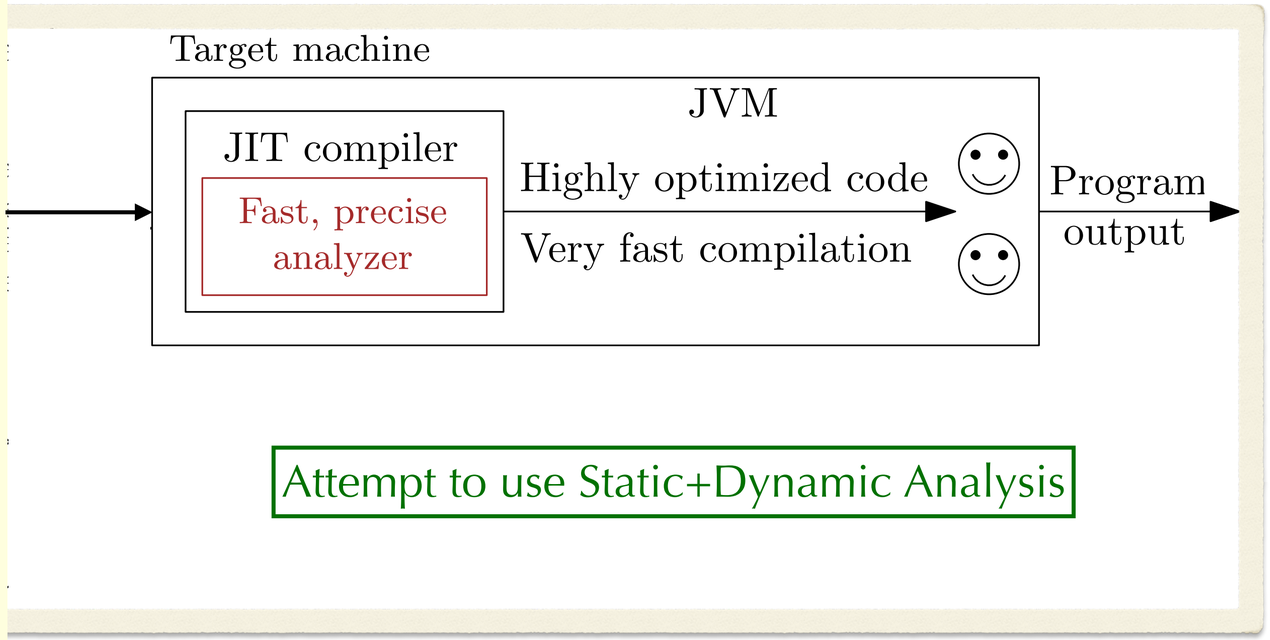
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```

Stack Allocate
O_4, O_5 and O_6



Static+Dynamic Analysis

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A(); // O4
     A y = new A(); // O5
     x.f = new A(); // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  }
  void zar(A p, A q) {...}
  void bar(A p1, A p2) {
     p1.f = p2;
  }
}
```



Stack Allocate

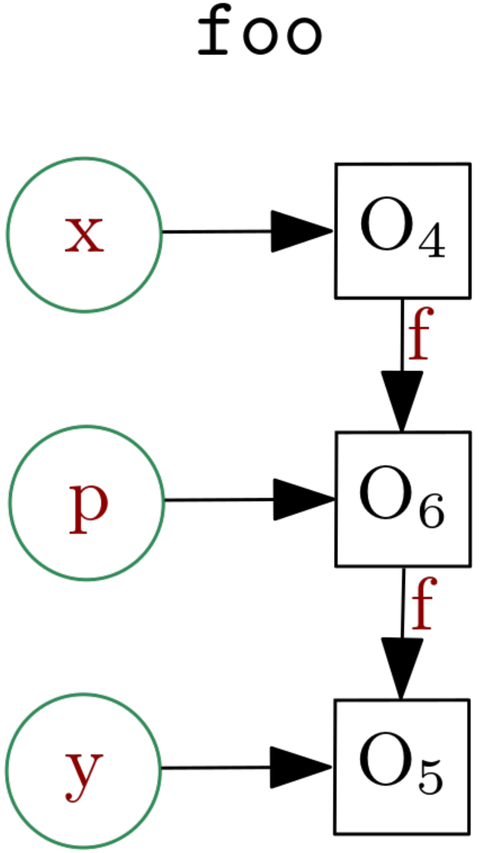
O_4, O_5 and O_6

Static+Dynamic Analysis with Dynamism

```
1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.       A x = new A();  // O4
5.       A y = new A();  // O5
6.       x.f = new A();  // O6
7.       A p = x.f;
8.       bar(p, y);
9.       r.zar(p, q);
10.  } /* method foo */
```

```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
16. class B extends A
17.   void zar(A p, A q) {
18.       q.f = p;
19.   } /* method zar */
20. } /* class B */
```

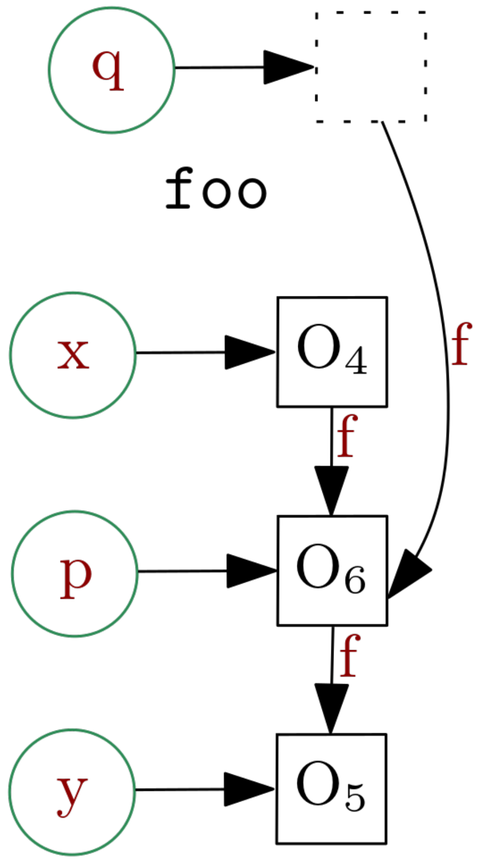
Dynamically loaded



Static+Dynamic Analysis with Dynamism

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

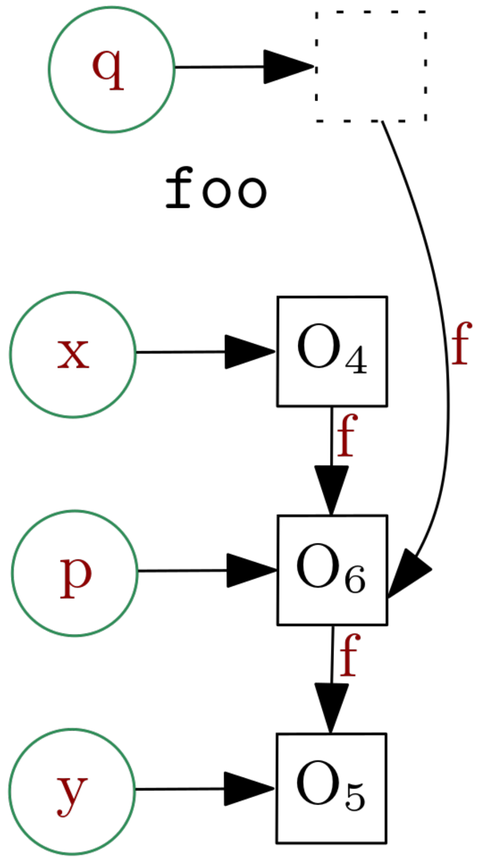
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
16. class B extends A
17.   void zar(A p, A q) {
18.       q.f = p;
19.   } /* method zar */
20. } /* class B */
```



Static+Dynamic Analysis with Dynamism

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
16. class B extends A
17.   void zar(A p, A q) {
18.       q.f = p;
19.   } /* method zar */
20. } /* class B */
```



Incorrect allocation on stack

Static+Dynamic Analysis with Dynamism

- Static analysis often cannot capture possible dynamic features at run-time.
- Static-analysis results might go wrong and lead to incorrect stack allocation!

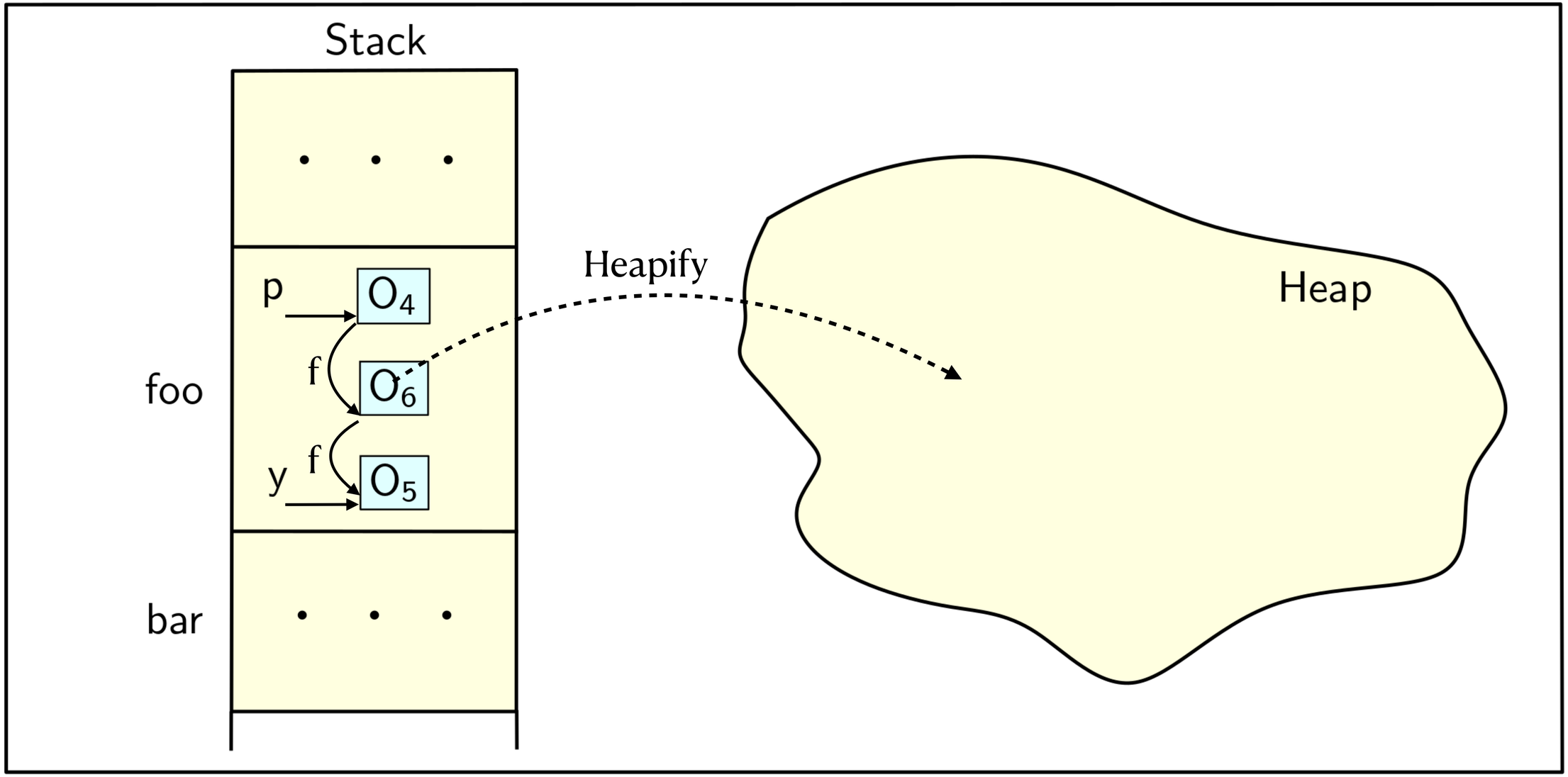
- **Pessimist:** Do not use static-analysis results in a VM.

- **Optimist:** Use static-analysis results and invalidate if needed!

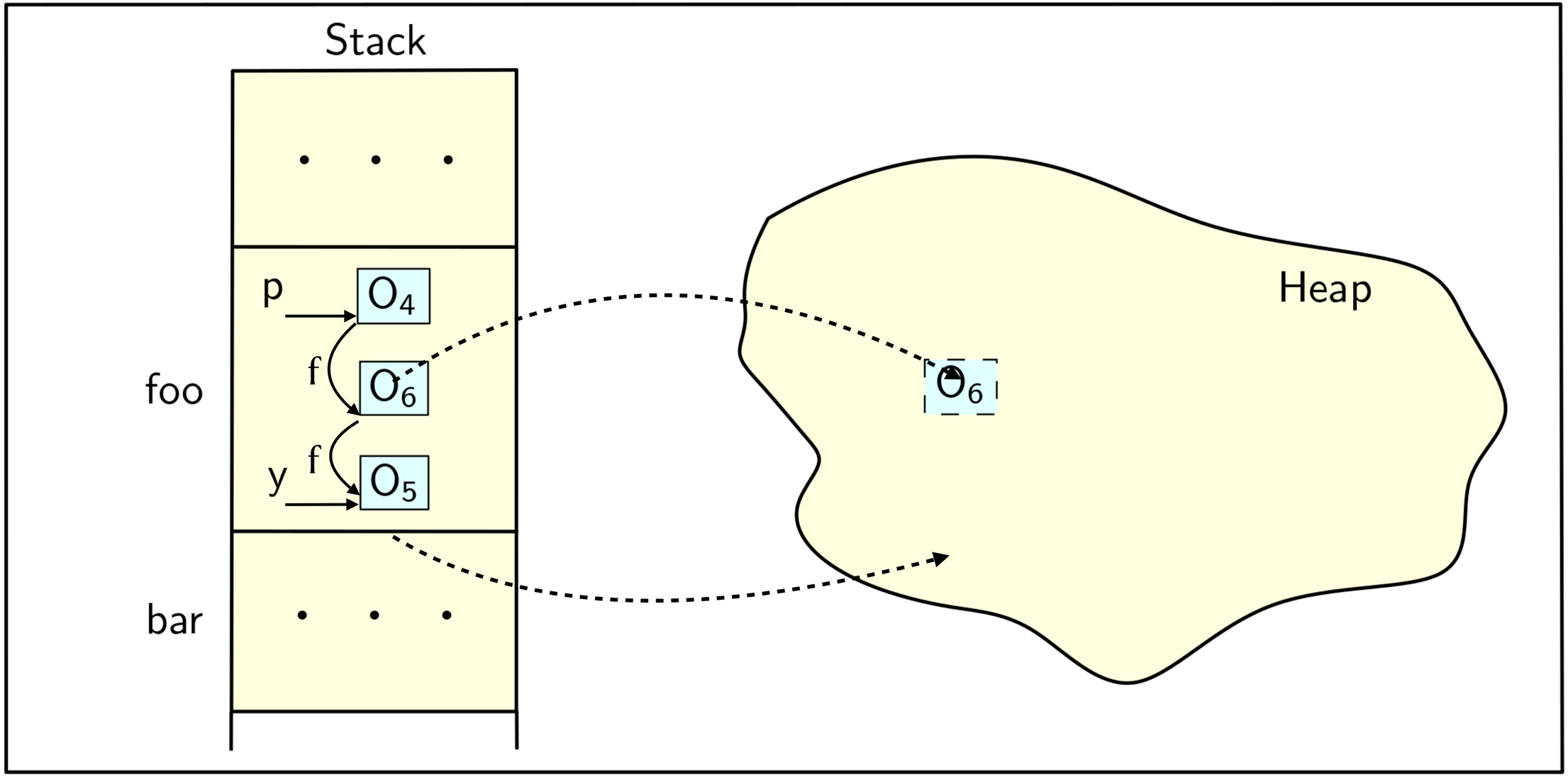
- **Idea:** Treat static-analysis results as hints to perform *optimistic stack allocation*, and *detect+repair* incorrect stack allocation dynamically.

Dynamic Heapification

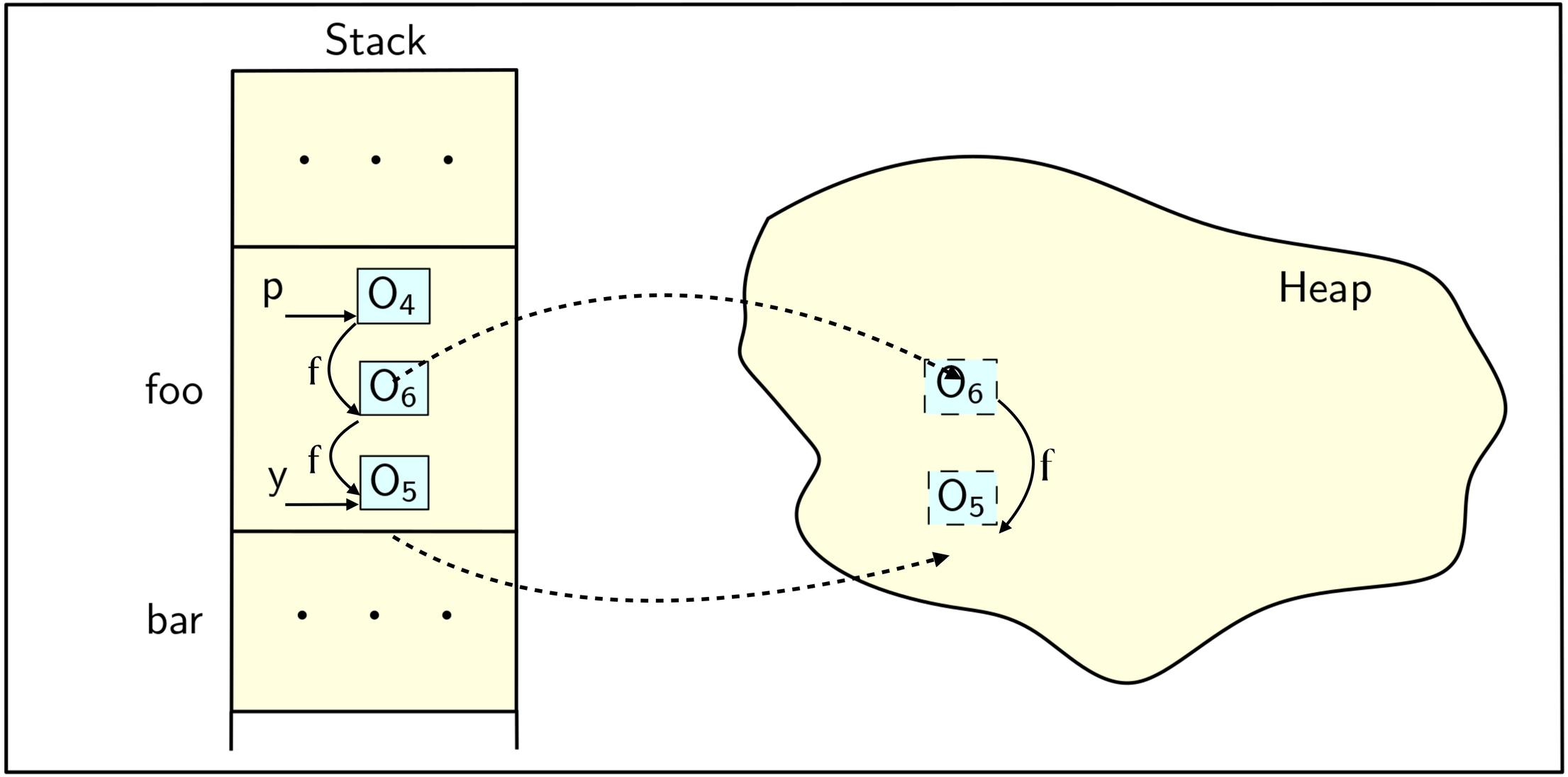
Dynamic Heapification



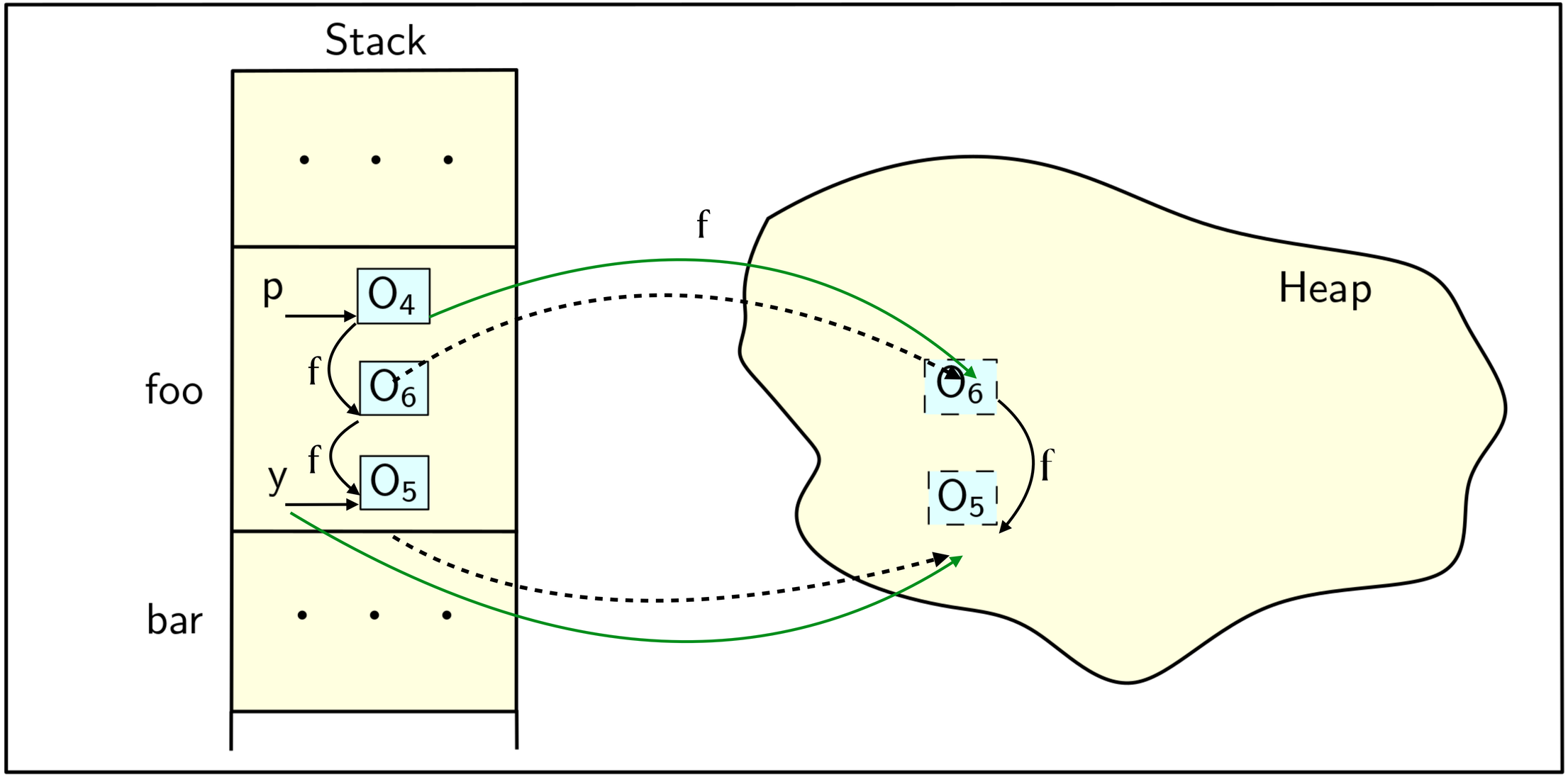
Dynamic Heapification



Dynamic Heapification



Dynamic Heapification



Heapification requires Heapification Checks

- Heapification checks are executed at run-time (inserted by the *codegen* routine of the JIT compiler, as well as added to the interpreter).
- How often? At each store, return, throw, …, basically at each possible escape point.
- And see how do they look for `a.f = b`:
  - if O_b is on stack
    - if O_a is on heap then heapify the object(s) "reachable" from b
    - else if both are on the current stack frame then do nothing
    - else if
      - i) O_a and O_b are on the same stack frame: Do nothing
      - ii) O_a is on a stack frame that is popped *before* O_b's: Do nothing
      - iii) O_a is on a stack frame that is popped *after* O_b's: Heapify

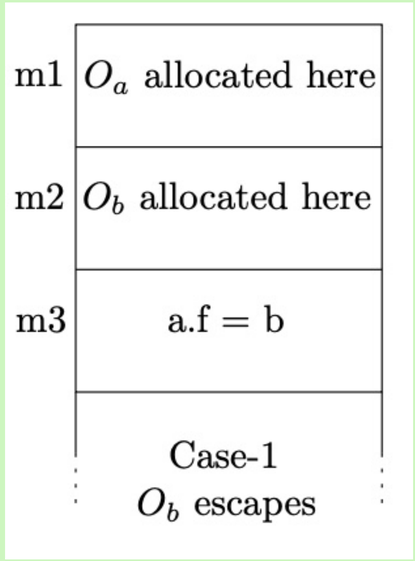
```
ptsto(a) = {Oa}
ptsto(b) = {Ob}
```

Can be checked by *walking the stack*.

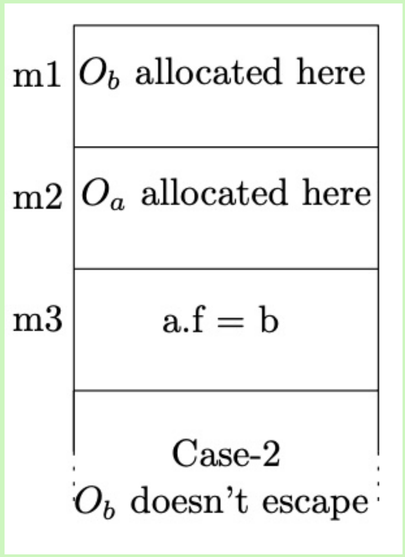
Possibilities at a Store Statement

```
class T {
    T f;
    void m1() {
        m2(...);
    }
    void m2() {
        m3(...);
    }
    void m3(T a, T b) {
        a.f = b;
    }
}
```

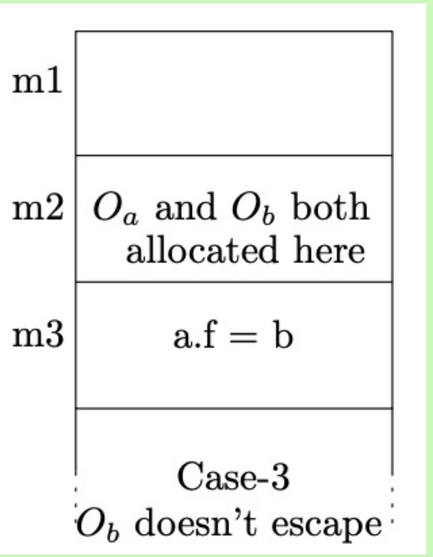
Stack addresses grow downward (higher to lower)



Heapify O_b



Do nothing



Do nothing

Heapification Checks with Fewer Stack Walks

```
a.f = b
```

0_a is not on heap and neither 0_a nor 0_b are on the current stack frame

i) 0_a and 0_b are on the same stack frame: Do nothing

ii) 0_a is on a stack frame that is popped *before* 0_b's: Do nothing

iii) 0_a is on a stack frame that is popped *after* 0_b's: Heapify

&0_a <= &0_b is a sufficient check

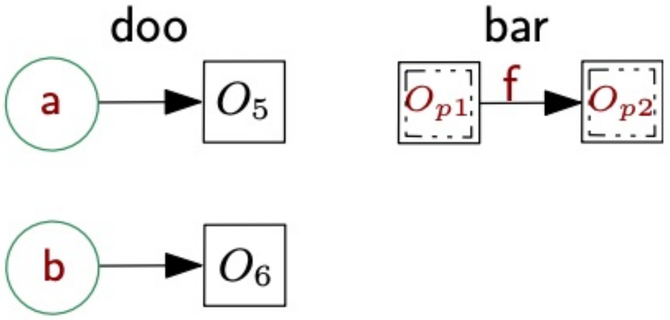
&0_a <= &0_b would still *work* if 0_b was placed on the stack before 0_a

➤ **Idea:** Determine a "stack order" for allocating non-escaping objects on stack frames, and use it to reduce the overhead of heapification checks at run-time.

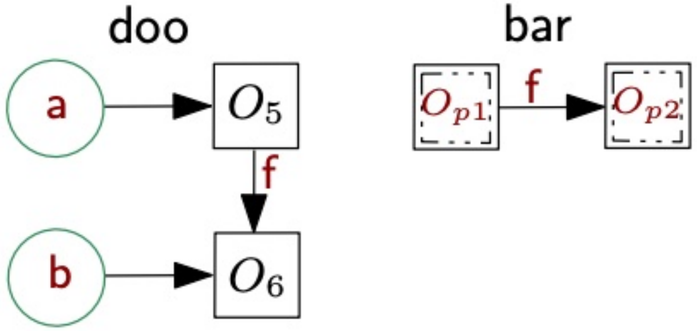
Determining Stack Orders

```
1   class A { B g; }
2   class D {
3      A f;
4      void doo(D q) {
5         D a = new D(); // O5
6         A b = new A(); // O6
7         a.f = new A(); // O7
8         A p = a.f;
9         a.f.g = new B(); // O9
10        zar(p, q);
11           bar(a, b);
12     } /* method doo */
13     void zar(A p, D q) {
14        q.f = new A(); // O14
15        ...
16     } /* method zar */
17     void bar(D p1, A p2) {
18        p1.f = p2;
19     } /* method bar */
20  } /* class D */
```

Intraprocedural:

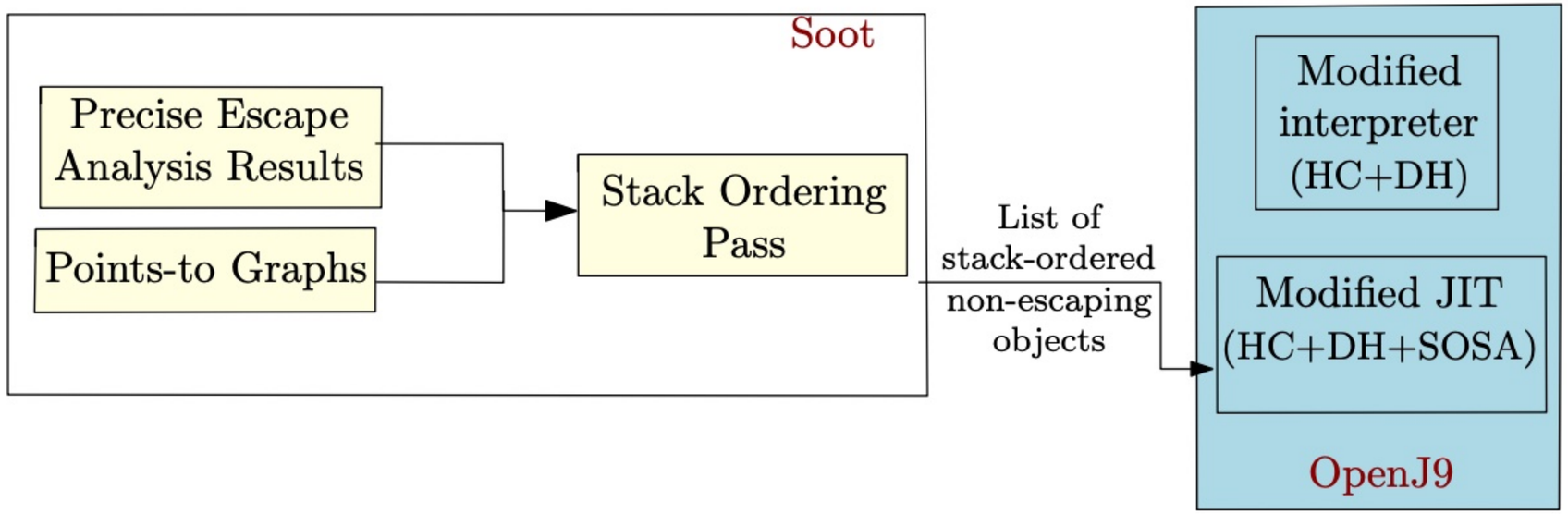


Interprocedural:



O_6 should be stack allocated before O_5 (decreasing addresses).

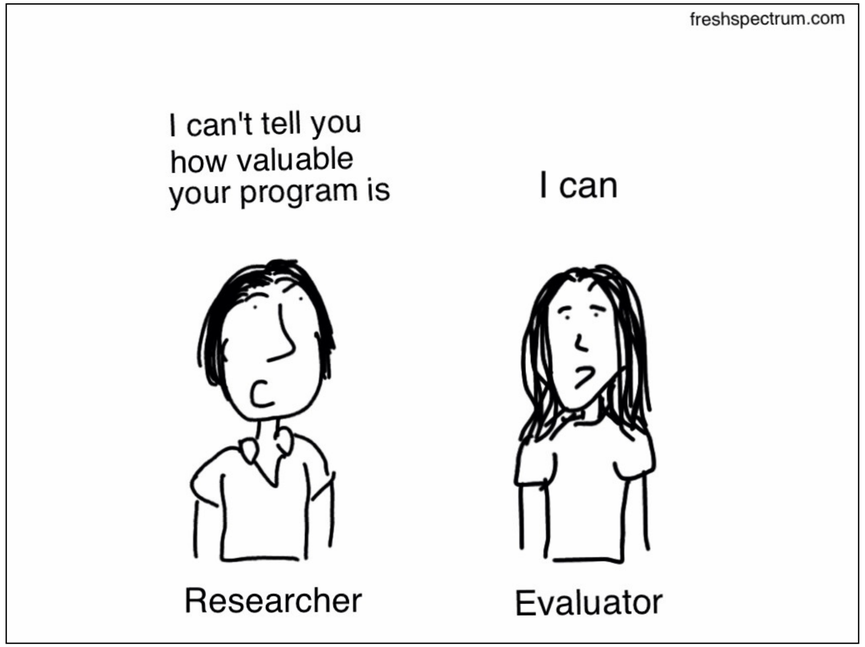
Thus, Here's What We've Got



Optimistic Stack Allocation with Efficient Dynamic Heapification

Evaluation

- **Implementation:**
  - Static analysis: Soot
  - Runtime components: OpenJ9
- **Evaluation modes:**
  - BASE: Stack allocation with the existing scheme
  - OPT: Stack allocation with our optimistic scheme
- **Measures:**
  - Enhancement in stack allocation
  - Impact on performance and garbage collection



Enhancement in Stack Allocation

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Enhancement in Stack Allocation

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Enhancement in Stack Allocation

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

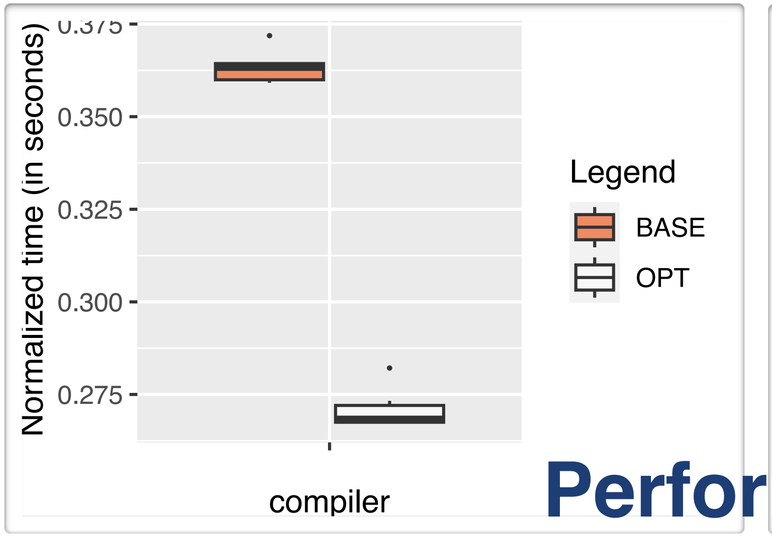
Enhancement in Stack Allocation

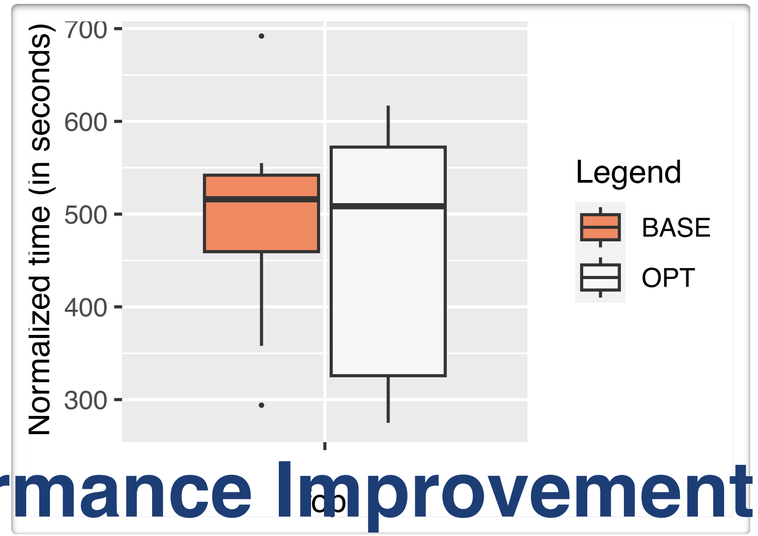
	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (2.23%)	775MB	72 (3.13%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

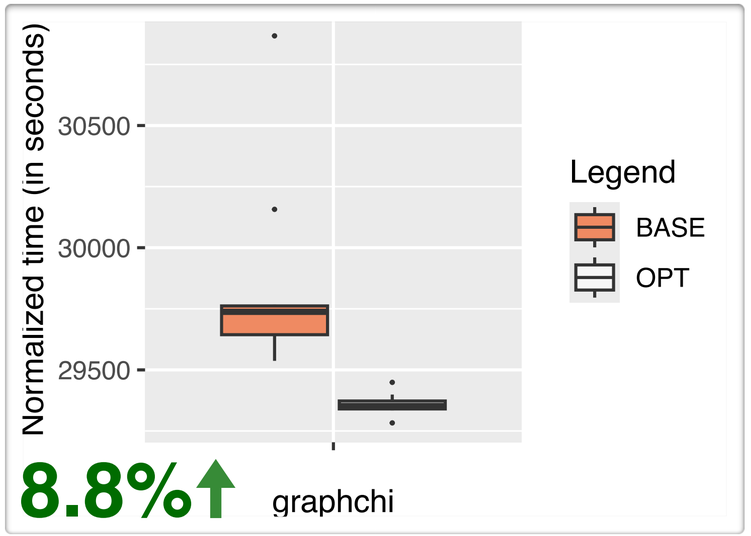
Stack Allocation: 71%↑ Stack Bytes: 54%↑

(Less Heap Allocation)

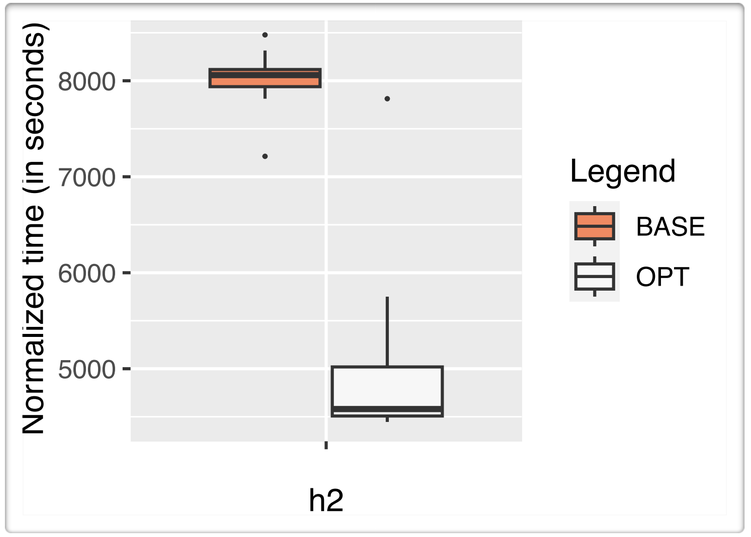
Performance Improvement

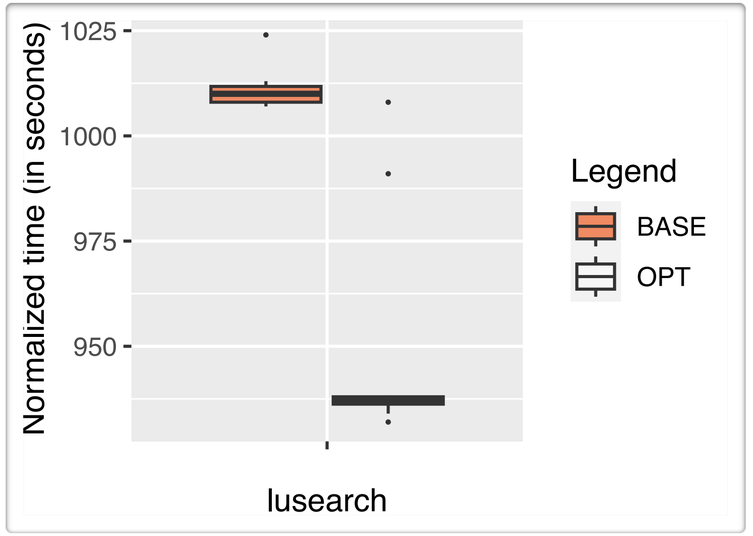


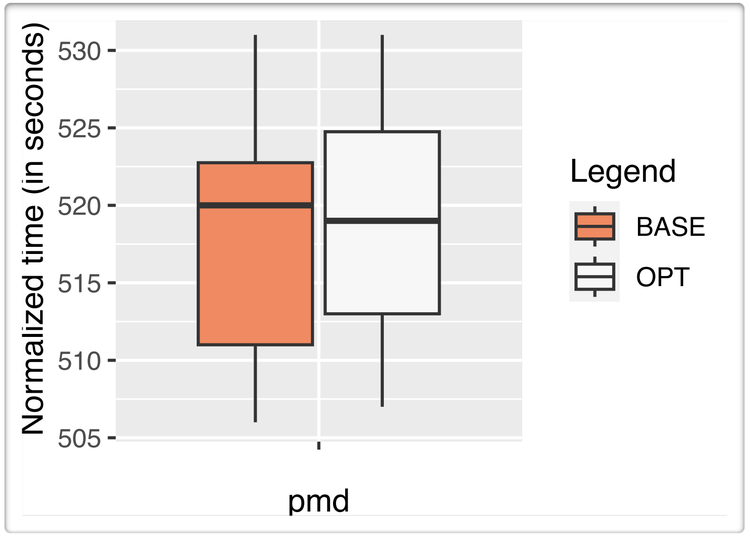




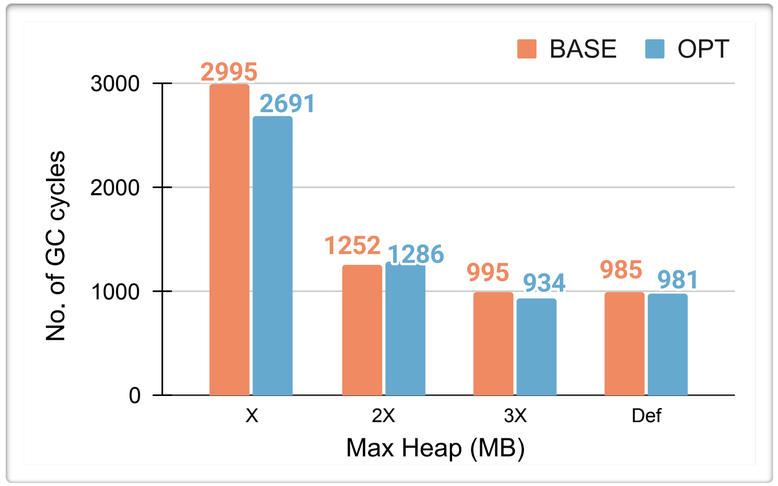
Performance Improvement: 8.8%↑



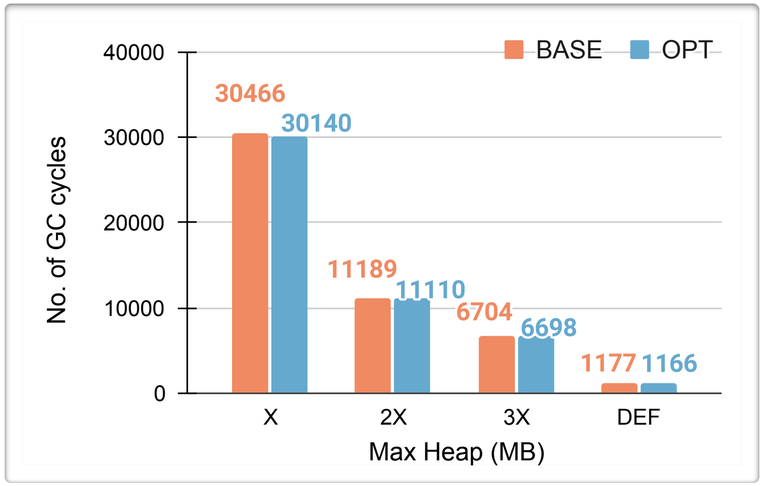




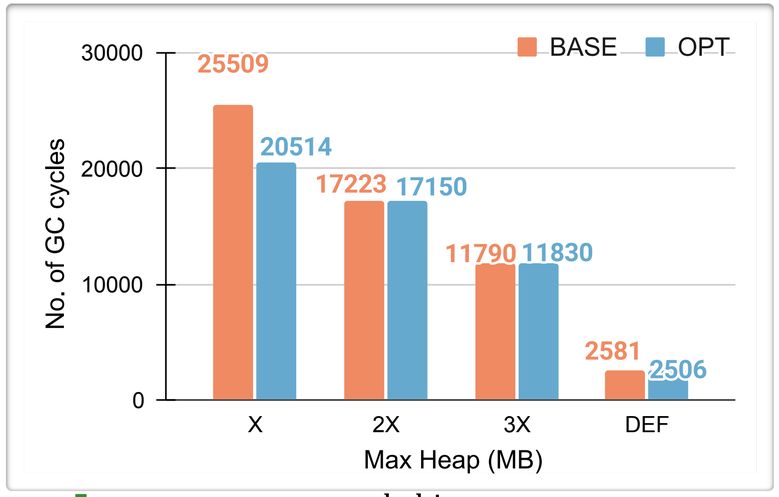
Reduction in Garbage Collection



compiler

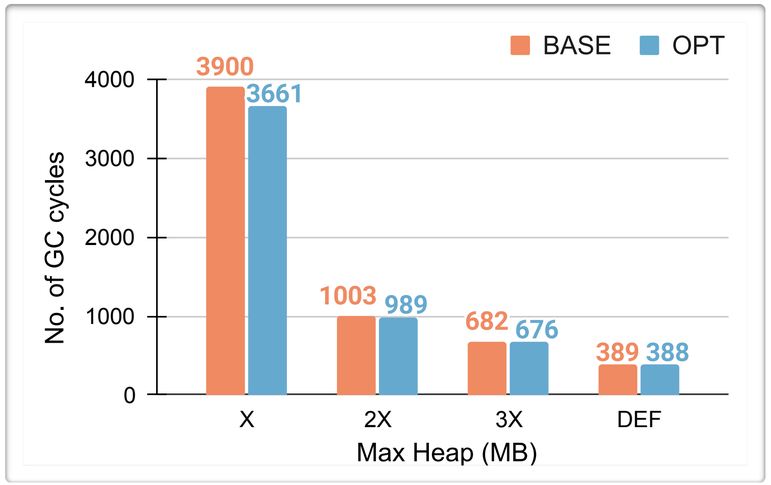


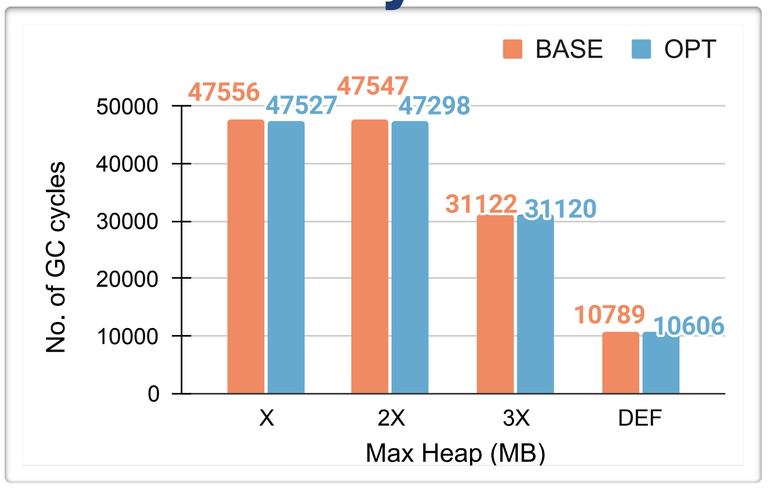
fop

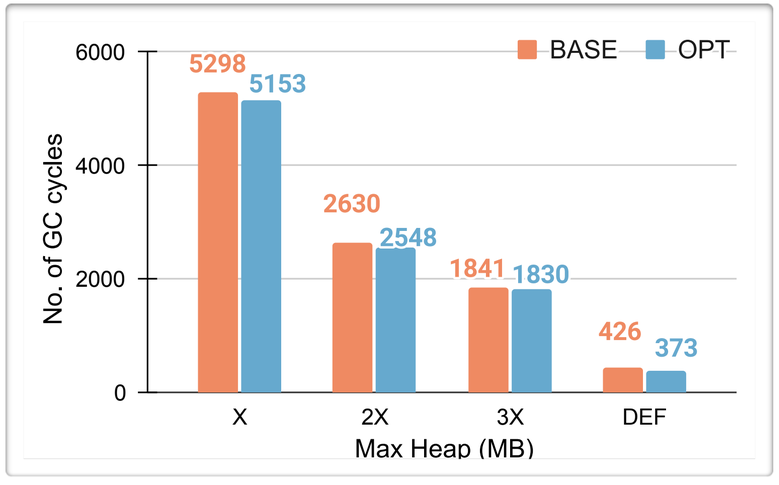


graphchi

Fewer GC Cycles: 5.3% ⬇



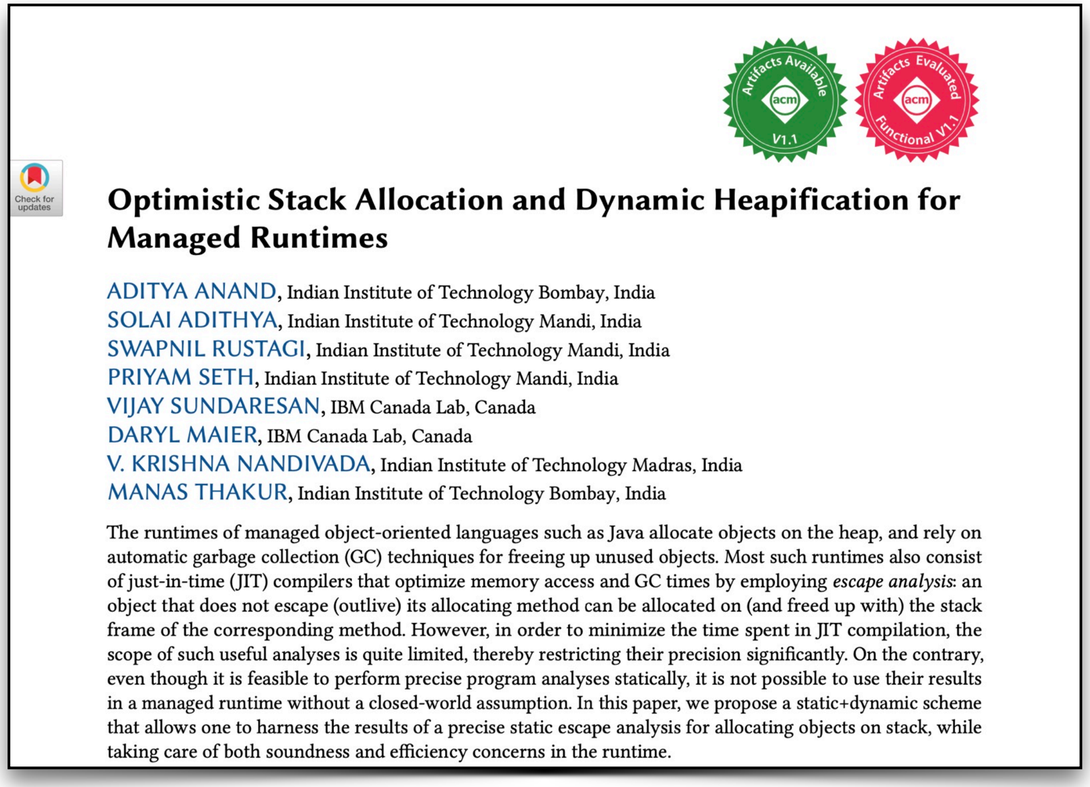




Takeaways

- Improved an important OO optimization.
- Precision without thwarting JIT efficiency.
- Functional correctness with efficient run-time checks.
- **First sound and efficient usage of AOT results in a JIT compiler!**

PLDI 2024



Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

Recent Works on AOT+JIT Analysis

PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs

TOPLAS 19

MANAS THAKUR and V. KRISHNA NANDIVADA, IIT Madras

ZS3: Marrying Static Analyzers and Constraint Solvers to Parallelize Loops in Managed Runtimes

CASCON 22

Rishi Sharma*
EPFL
rishi-sharma@outlook.com

Shreyansh Kulshreshtha*
Publicis Sapient
shreyanshkuls@outlook.com

Manas Thakur
IIT Mandi
manas@iitmandi.ac.in

Check for updates

Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

PLDI 24

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

Check for updates

Principles of Staged Static+Dynamic Partial Analysis

SAS 22

Aditya Anand and Manas Thakur(✉)

Indian Institute of Technology Mandi, Kamand, India
ud21002@students.iitmandi.ac.in, manas@iitmandi.ac.in

RESEARCH

Check for updates

Partial program analysis for staged compilation systems

FMSD 24

Aditya Anand[1] · Manas Thakur[1]

Received: 30 April 2023 / Accepted: 16 May 2024 / Published online: 13 June 2024

All these works assume AOT analysis is better than JIT.

Examples of Run-time Information

- **Profile information:**
  - Basic invocation and loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.

How can we get the best of static analysis and run-time information?

- **Dynamic class hierarchy:**
  - Information about loaded subclasses of a given class during execution.

- **Inlining table:**
  - Information about the methods inlined at various callsites.

Combining Static Analysis and Speculation in JIT Compilers

Possibility 1: Polymorphic Callsites

```
1. class A {
2. static A global;
     .  .  .
4.   void foo(A z) {
5.         A x = new A(); // O5
6.         A y = new A(); // O6
7.         x.f = new A(); // O7
       . . .
14.        z.bar(x);
15.    } /* method foo */
16. } /* class A */
```

```
17. class B extends A {
18.    void bar(A p2) { . . . }
19. } /* class B */
```

```
20. class C extends A {
21.    void bar(A p3) {
22.       global = p3;
23.    }
24. } /* class C */
```

$\sqcap$: {B, C } —> [O_5],[O_7] [Escaping]

Possibility 1: Polymorphic Callsites

```
1. class A {
     . . .
4.   void foo(A z) {
5.       A x = new A(); // O5
6.       A y = new A(); // O6
7.       x.f = new A(); // O7
     . . .
14.      z.bar(x);
15.  } /* method foo */
16. } /* class A */
```

```
17. class B extends A {
18.   void bar(A p2) { . . . }
19. } /* class B */
```

```
20. class C extends A {
21.   void bar(A p3) {
22.     global = p3;      Escapes
23.   }
24. } /* class C */
```

polymorphic_cond

A.foo()[..][z14, {B},{O5, O7}]

During JIT compilation:

- Class Hierarchy (CHTable): C is not loaded.
- Most of the times z is of type B.

Possibility 2: Branches

```
1. class A {
      . . .
5.   void foo(A p1) {
6.       A y = new A(); // O6
         . . .
9.       if (p1 instance A) {
         . . .
11.      } else {
13.          global = y;  Escapes
14.    }
15.    y.f = p1;
16.   } /* method foo */
```

⊓ : —> O_6 (Escaping)

Possibility 2: Branches

```
1. class A {
     . . .
5.  void foo(A p1) {
6.       A y = new A(); // O6
         . . .
9.       if (p1 instance A) {
         . . .
11.      } else {
13.         global = y;
14.    }
15.    y.f = p1;
16.   } /* method foo */
```

During JIT compilation:

- The “then” branch is found to be taken most number of times.

branching_cond

`A.foo()[..][9,{O6}]`

Possibility 3: Method Inlining

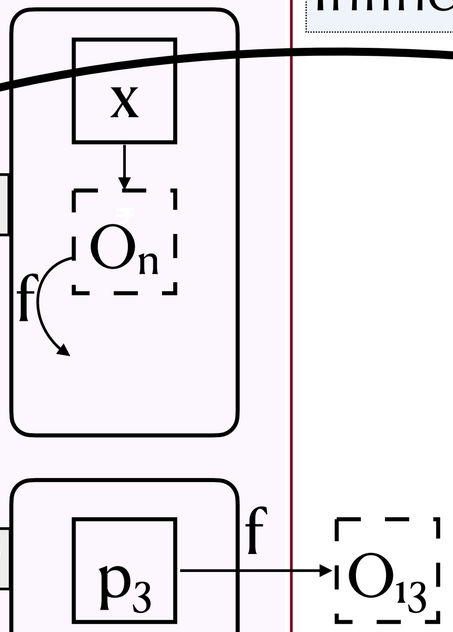
```
1. class A {
2.   void foo() {
3.     . . .
4.     z.bar(x);
5.     r.foobar(p, q);
6.   } /* method foo */
7.   void bar(A p1) { . . . }
8.   void foobar(A p2) { . . . }
9. } /* class A */
```

```
10. class B extends A {
11.   void bar(A p3) {
12.     // p3's pointee doesn't escape
13.     p3.f = new A(); // O13
14.     p3.foobar(p3.f);
15.   } /* method bar */
16.   void foobar(A p4) { . . . }
17. } /* class B */
18. class C extends A { . . . }
```

O_{13} marked as Escaping.

Possibility 3: Method Inlining

```
1. class A {
2.   void foo() {
3.     . . .
4.     z.bar(x);
5.     r.foobar(p, q);
6.   } /* method foo */
7.   void bar(A p1) { . . . }
8.   void foobar(A p2) { . . . }
9. } /* class A */
```



Inline

```
10. class B extends A {
11.   void bar(A p3) {
12.     // p3's pointee doesn't escape
13.     p3.f = new A(); // O13
14.     p3.foobar(p3.f);
15.   } /* method bar */
16.
17. } /* class B */
18. class C extends A { . . . }
```

Callee object on caller's stack frame

inlining_cond

A.foo()[..][4,B.bar(p3),{O13}]

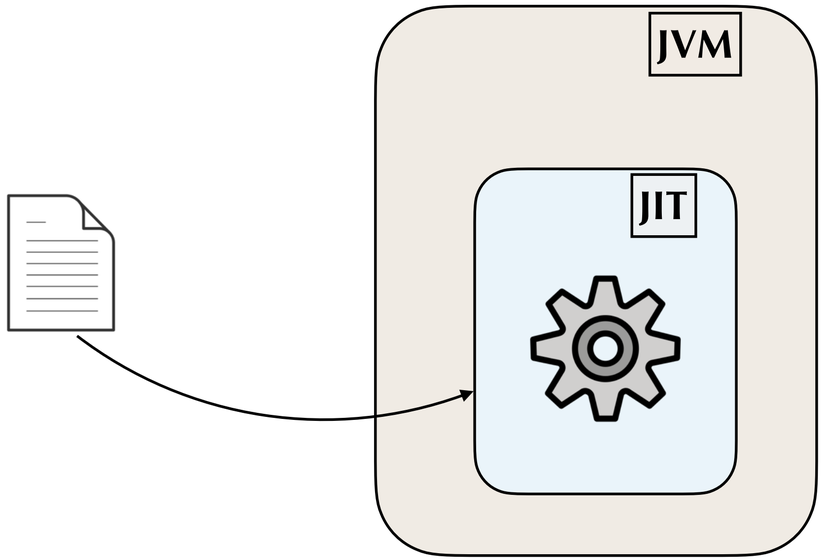
Enriched Static-Analysis Results

Unconditional

Conditional

`A::foo()` [direct_allocation] [polymorphic_cond] [branching_cond] [inlining_cond]

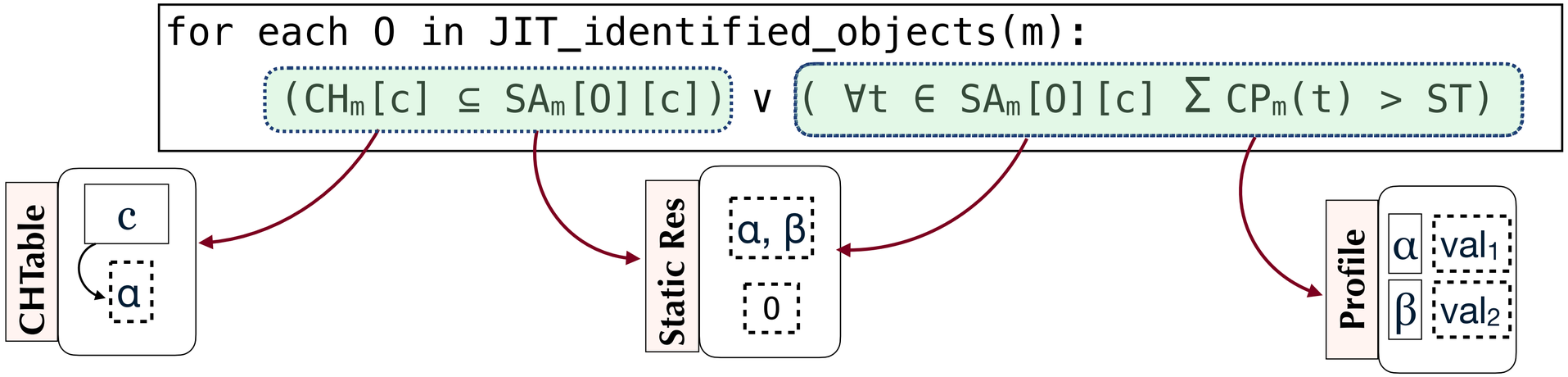
Statically generated results



Speculative allocation done during JIT compilation

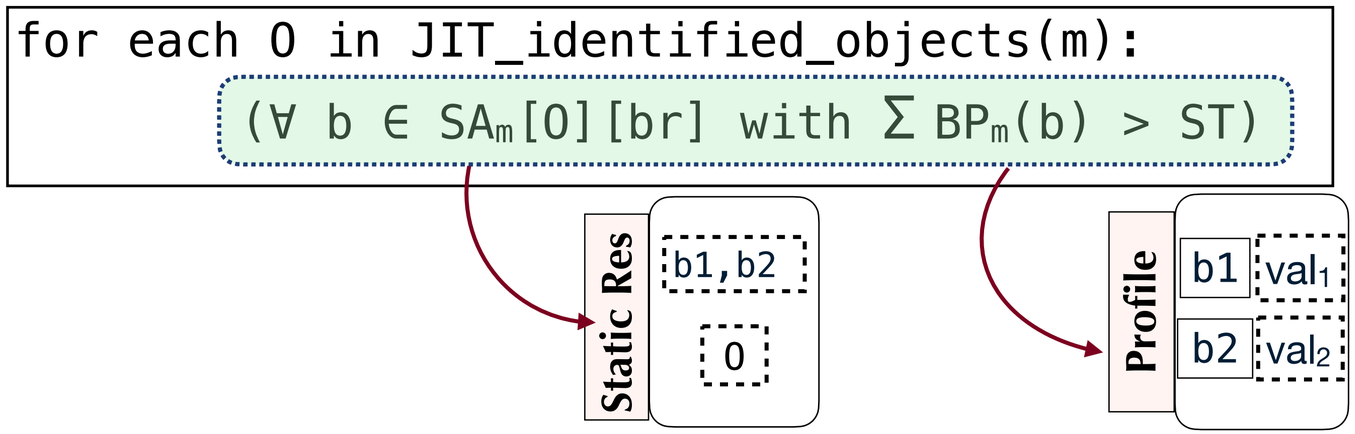
Actions in the JIT Compiler

1. Polymorphic Callsites



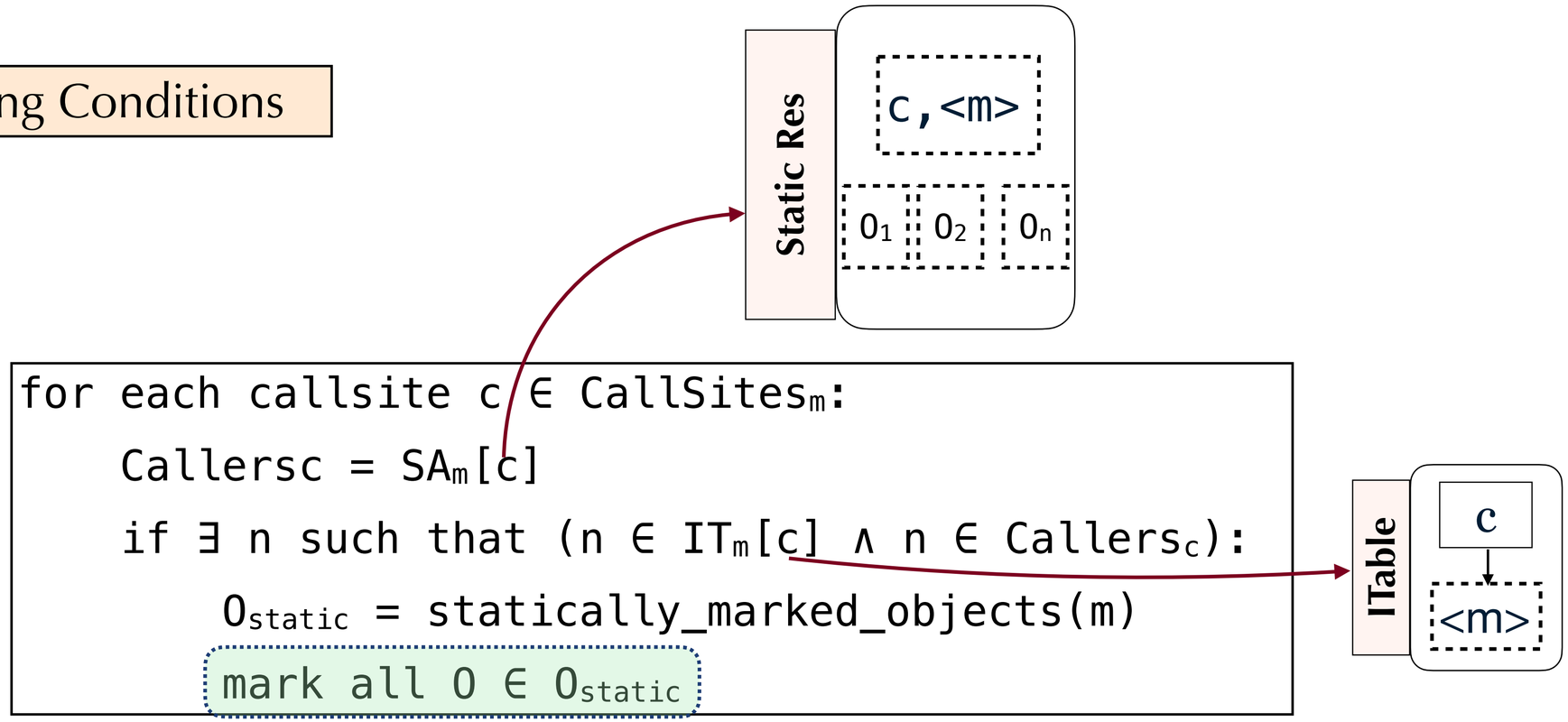
Actions in the JIT Compiler

2. Branching Conditions



Actions in the JIT Compiler

3. Inlining Conditions

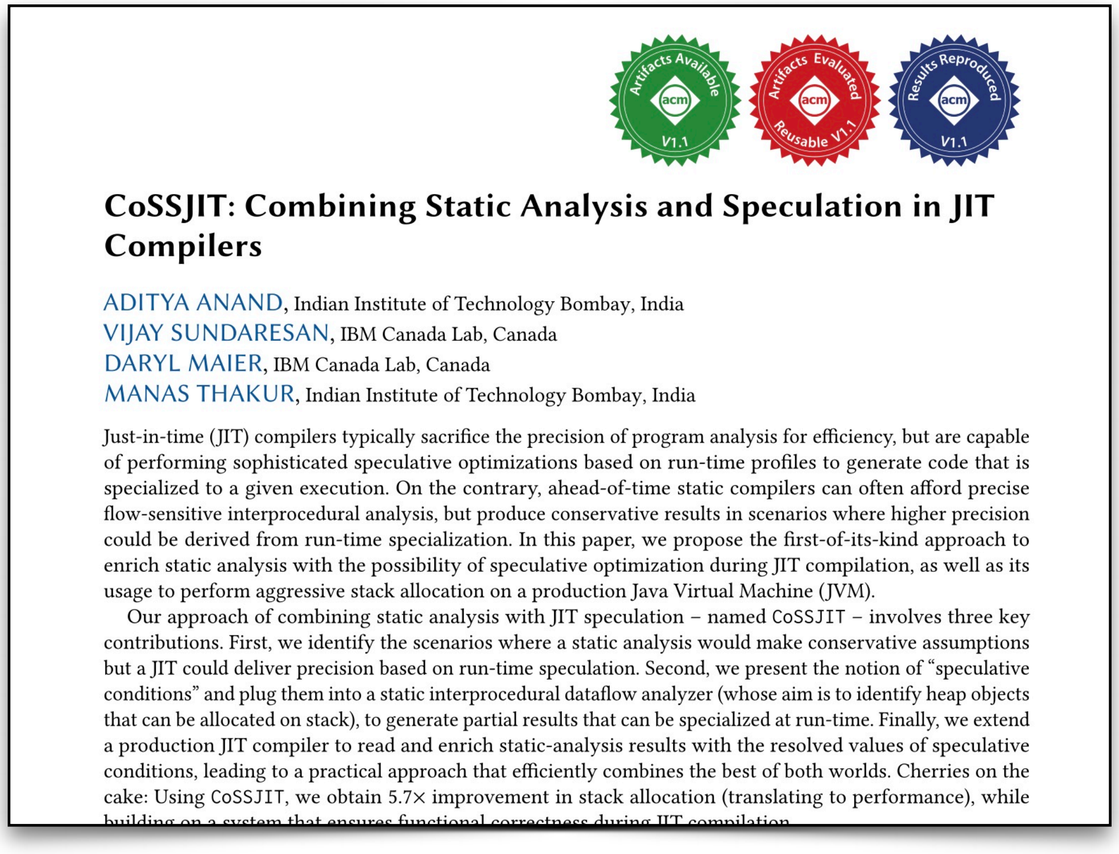


Summary of Results

- Numbers: Even better!
  - Higher stack allocation.
  - Lesser GC.
  - Better performance.

- **First work to combine static (AOT) and dynamic (JIT) analyses to achieve the best of both the worlds.**

OOPSLA 2025



CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

ADITYA ANAND, Indian Institute of Technology Bombay, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
MANAS THAKUR, Indian Institute of Technology Bombay, India

Just-in-time (JIT) compilers typically sacrifice the precision of program analysis for efficiency, but are capable of performing sophisticated speculative optimizations based on run-time profiles to generate code that is specialized to a given execution. On the contrary, ahead-of-time static compilers can often afford precise flow-sensitive interprocedural analysis, but produce conservative results in scenarios where higher precision could be derived from run-time specialization. In this paper, we propose the first-of-its-kind approach to enrich static analysis with the possibility of speculative optimization during JIT compilation, as well as its usage to perform aggressive stack allocation on a production Java Virtual Machine (JVM).

Our approach of combining static analysis with JIT speculation – named `CoSSJIT` – involves three key contributions. First, we identify the scenarios where a static analysis would make conservative assumptions but a JIT could deliver precision based on run-time speculation. Second, we present the notion of "speculative conditions" and plug them into a static interprocedural dataflow analyzer (whose aim is to identify heap objects that can be allocated on stack), to generate partial results that can be specialized at run-time. Finally, we extend a production JIT compiler to read and enrich static-analysis results with the resolved values of speculative conditions, leading to a practical approach that efficiently combines the best of both worlds. Cherries on the cake: Using `CoSSJIT`, we obtain 5.7× improvement in stack allocation (translating to performance), while building on a system that ensures functional correctness during JIT compilation

What are we doing further?

- How to **productize** such techniques and bring them to real-world compiler pipelines?
- Can we **optimize objects** that can't be allocated on the frame of the allocating method?
- How can we **tune the JVM** to be lighter, and even better, in presence of AOT results?
- Which **other analyses** and optimizations can benefit from an AOT+JIT approach, and which ones can *now* be made to work?
- How much can one gain from this approach for **languages that are more dynamic**?
- How to make it easier for **new researchers** to get started with such technologies?

Read more

OOPSLA26, CGO26, OOPSLA25, PLDI24, FMSD24, OOPSLA23, …

What are we doing further?

Learn more

https://www.cse.iitb.ac.in/~manas