Precision without Regret: Optimistic Stack Allocation in JIT Compilers

Aditya Anand
Advisor: Prof. Manas Thakur
Indian Institute of Technology Bombay

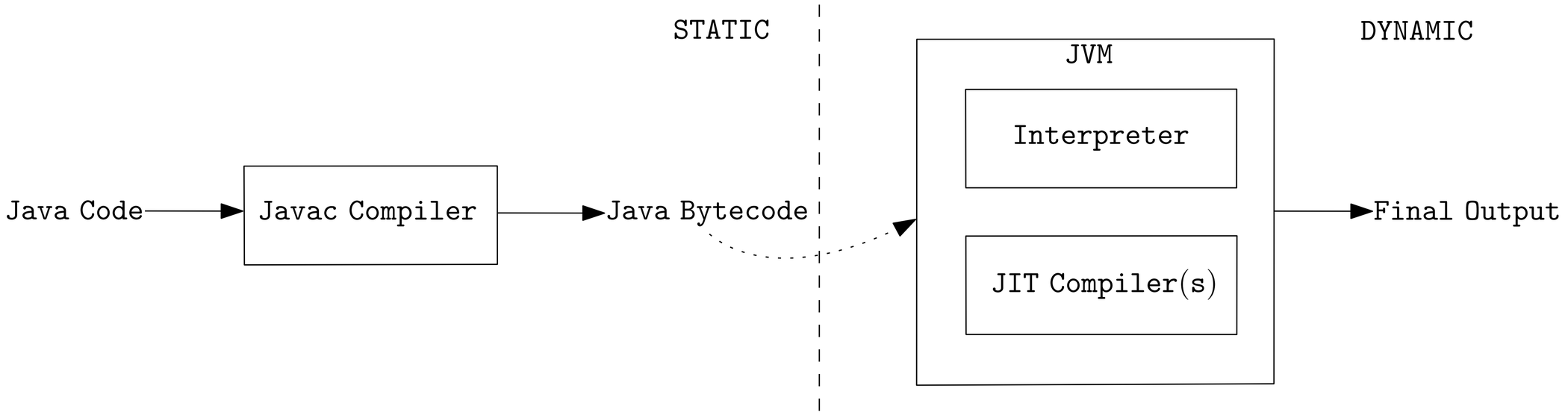
11th February 2026

Compilation in Programming Languages

- Languages like C, C++ :
  - Use static compilers (gcc, g++).
  - Generate executable which can be directly executed on machine.
  - Optimizations performed will be based on statically available information.

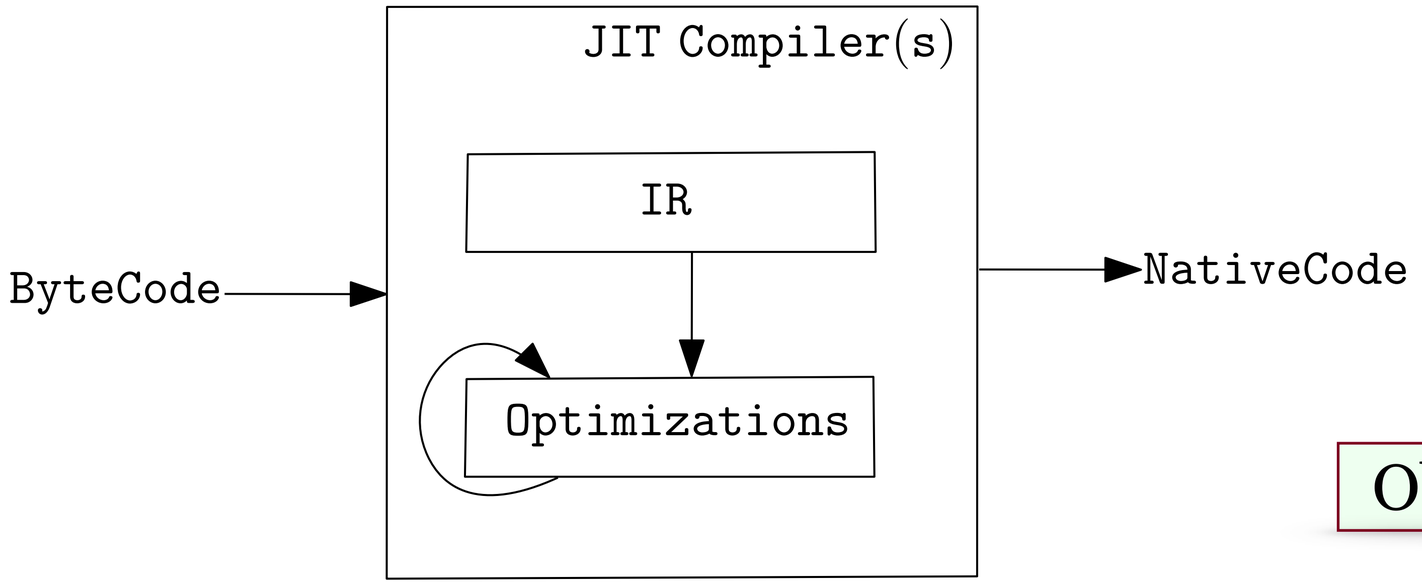
- Languages like Java, C# and Scala:
  - First get compiled by a static compiler.
  - Compiled output is passed to a managed runtime for further execution.

Program Translation in Java



- Static: Javac generates bytecode.
- Dynamic: Interpreter and JIT compiler generate the final output.

Program Translation in Java



Object Allocation

- Static: Javac generates bytecode.
- Dynamic: Interpreter and JIT compiler generate the final output.

Objects in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:
    - Unburden programmer from making complex allocation-deallocation decisions and reduce the possibility of harmful memory bugs.

- Challenges:
    - Access time is high.
    - Garbage collection is an overhead.

Stack Allocation

- Memory allocated on stack:
  - Less access time.
  - Get freed up as soon as the allocating method returns.
- Escape Analysis
  - Determines the set of objects that do not escape the allocating method.
- In case of Java:
  - Escape analysis is performed: Just-in-time (JIT) compilation — Imprecise
  - Very few objects get allocated on stack.

Staged Static + Dynamic Analysis

- Idea: Staged Static+Dynamic analysis for Managed Runtimes.
  - Offload the costly analysis to static time.
    - Perform precise (context-, flow-, field-sensitive) escape analysis ahead of time.
  - Use analysis results in the JIT to enable additional optimizations.
    - Statically generated escape analysis result to optimistically allocate objects on stack at run-time.

Challenges with Static Analysis

- Challenges:
  - Dynamic Features: Dynamic Class Loading (DCL), Hot-Code Replacement (HCR) allows code changes.
  - An object that was stack allocated based on static-analysis results, might start escaping at run-time.

How to safely allocate objects on stack in a managed runtime?

Motivating Example

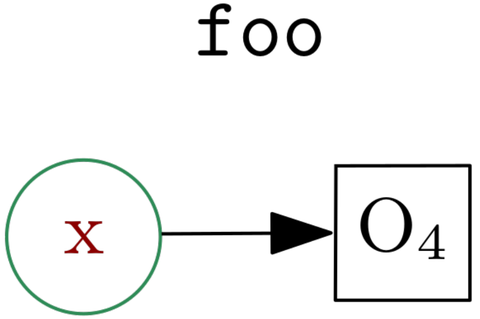
```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();   // O4
5.        A y = new A();   // O5
6.        x.f = new A();   // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

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

Motivating Example

```
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 */
```

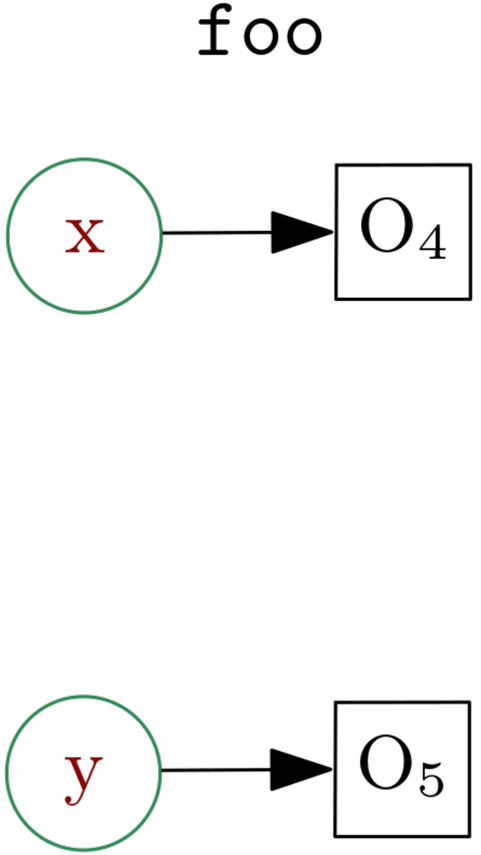
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.     p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Motivating Example

```
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 */
```

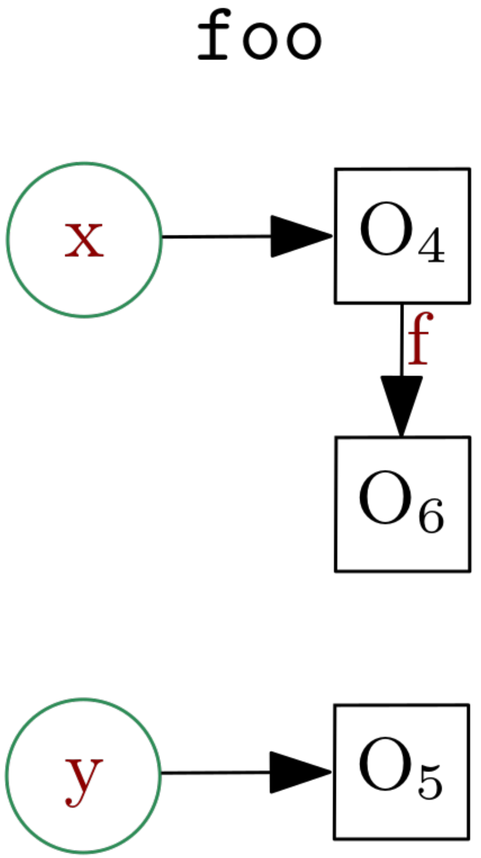
```
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.       p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Motivating Example

```
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 */
```

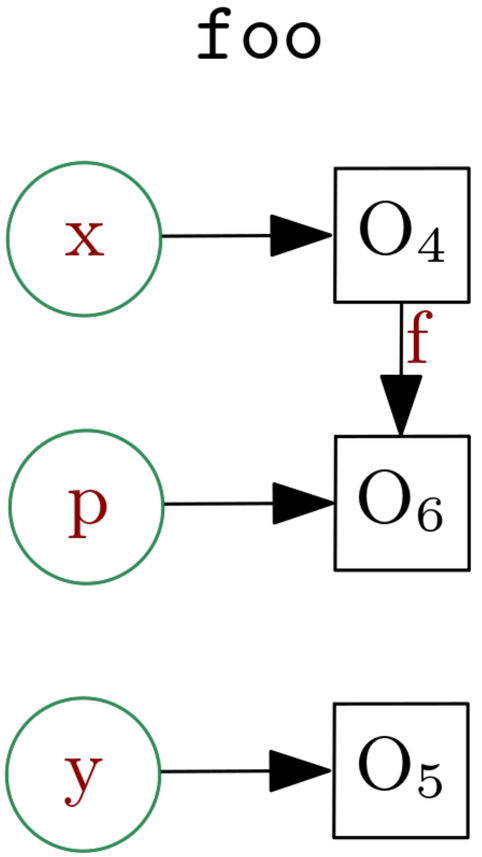
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Motivating Example

```
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 */
```

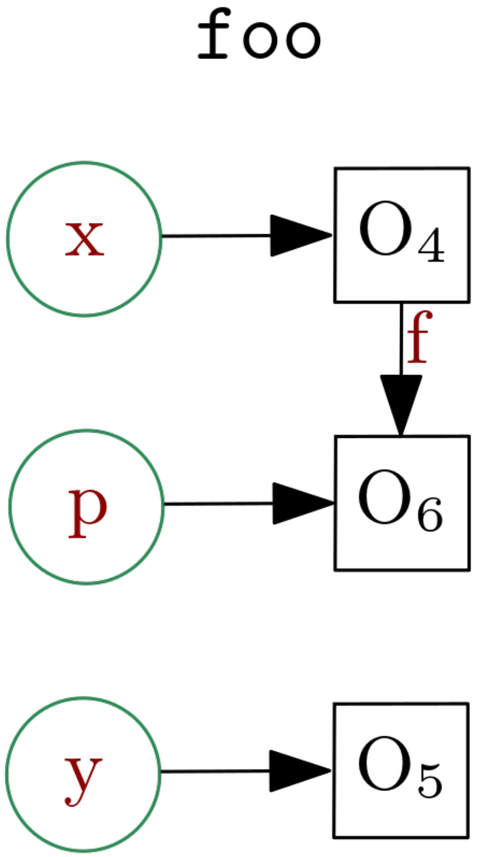
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Motivating Example

```
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 */
```

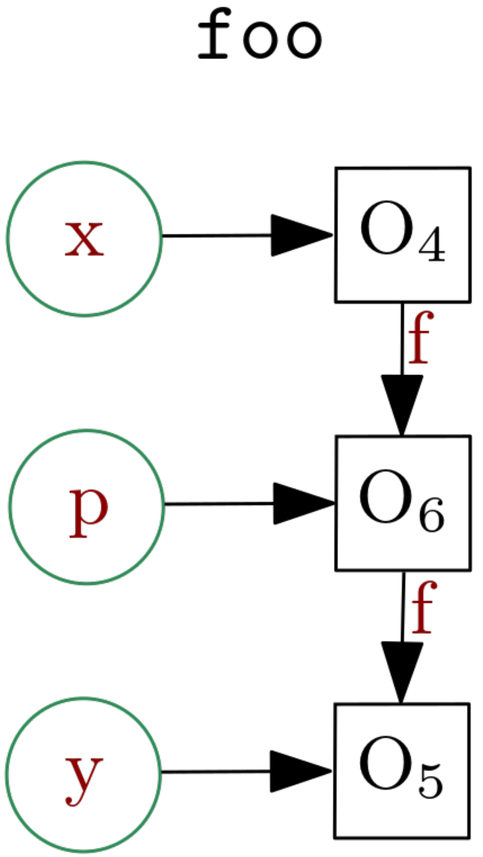
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Motivating Example

```
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 */
```

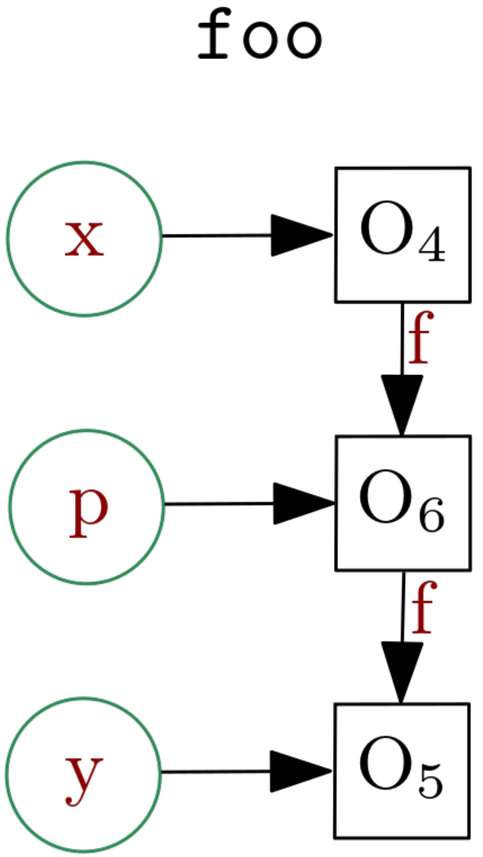


Motivating Example

```
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



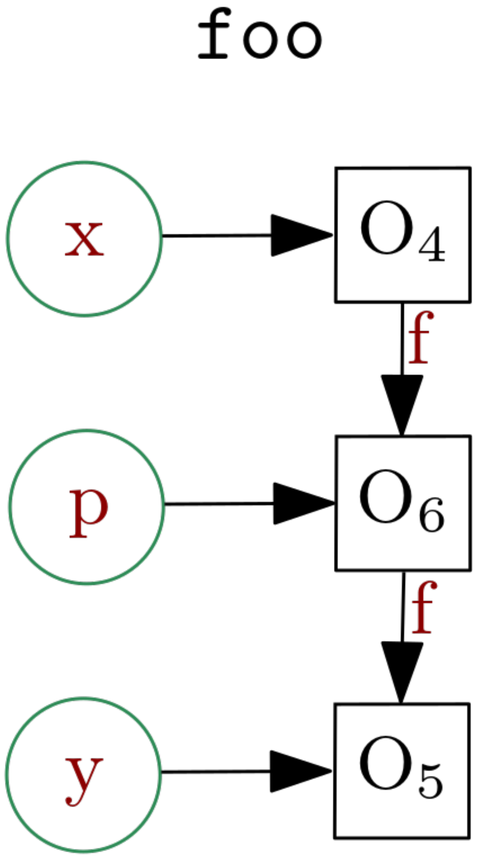
Motivating Example

```
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



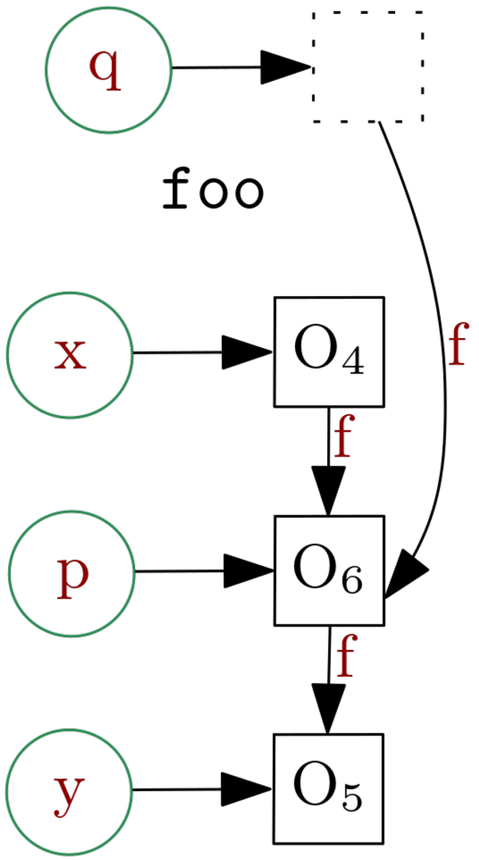
Motivating Example

```
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

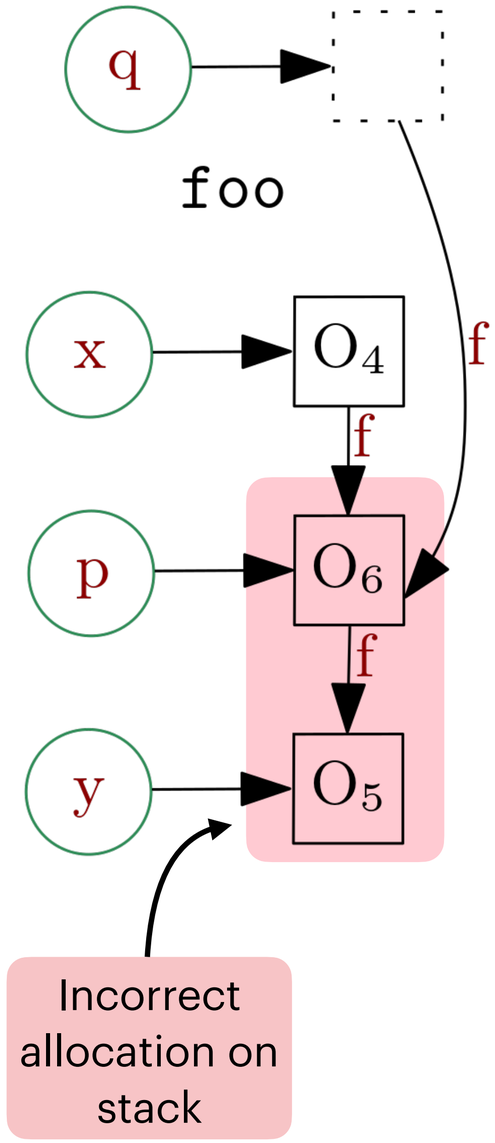


Motivating Example

```
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 */
```

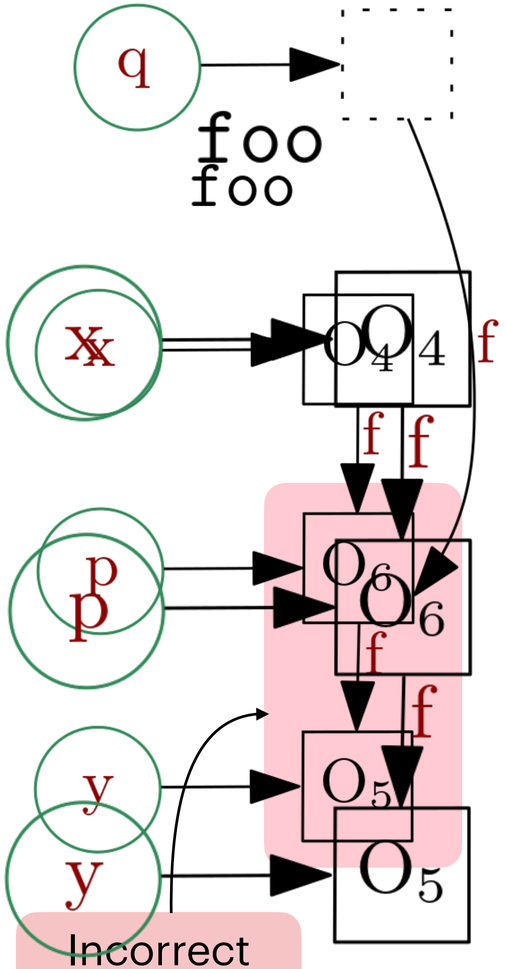
```
16. class B extends A
17.   void zar(A p, A q) {
18.      q.f = p;
19.   } /* method zar */
20. } /* class B */
```



HCR Example

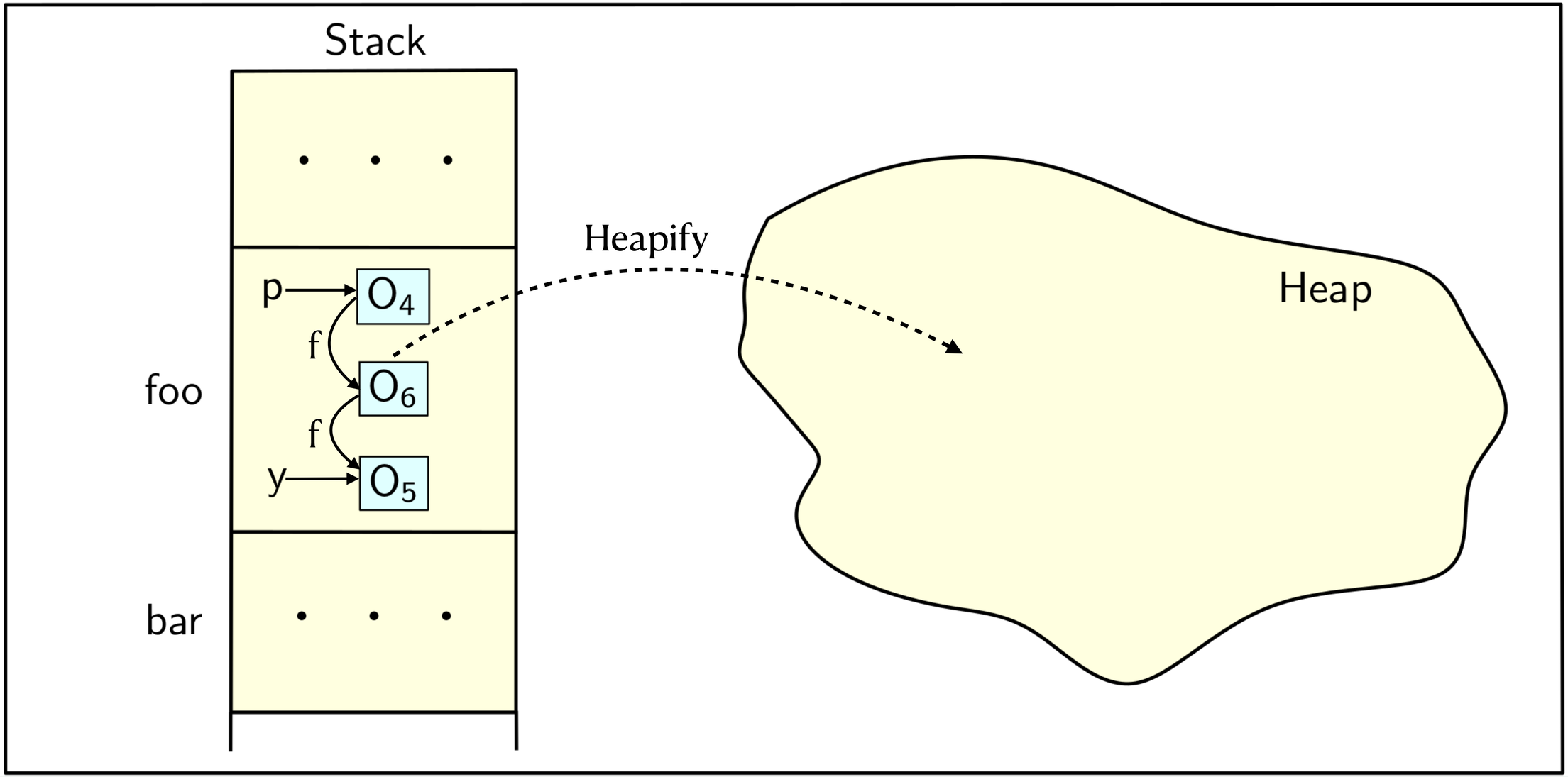
```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

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

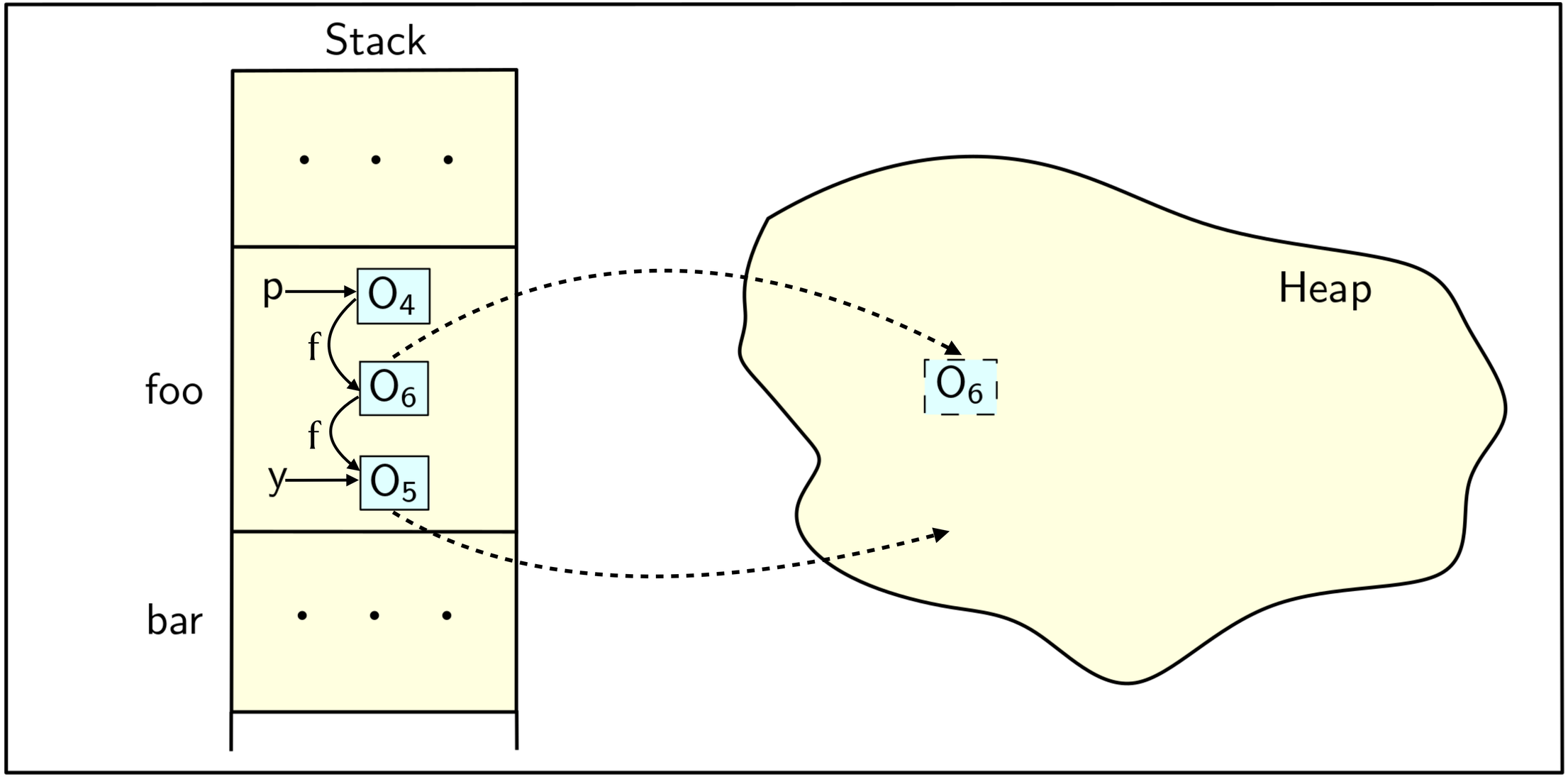


Dynamic Heapification

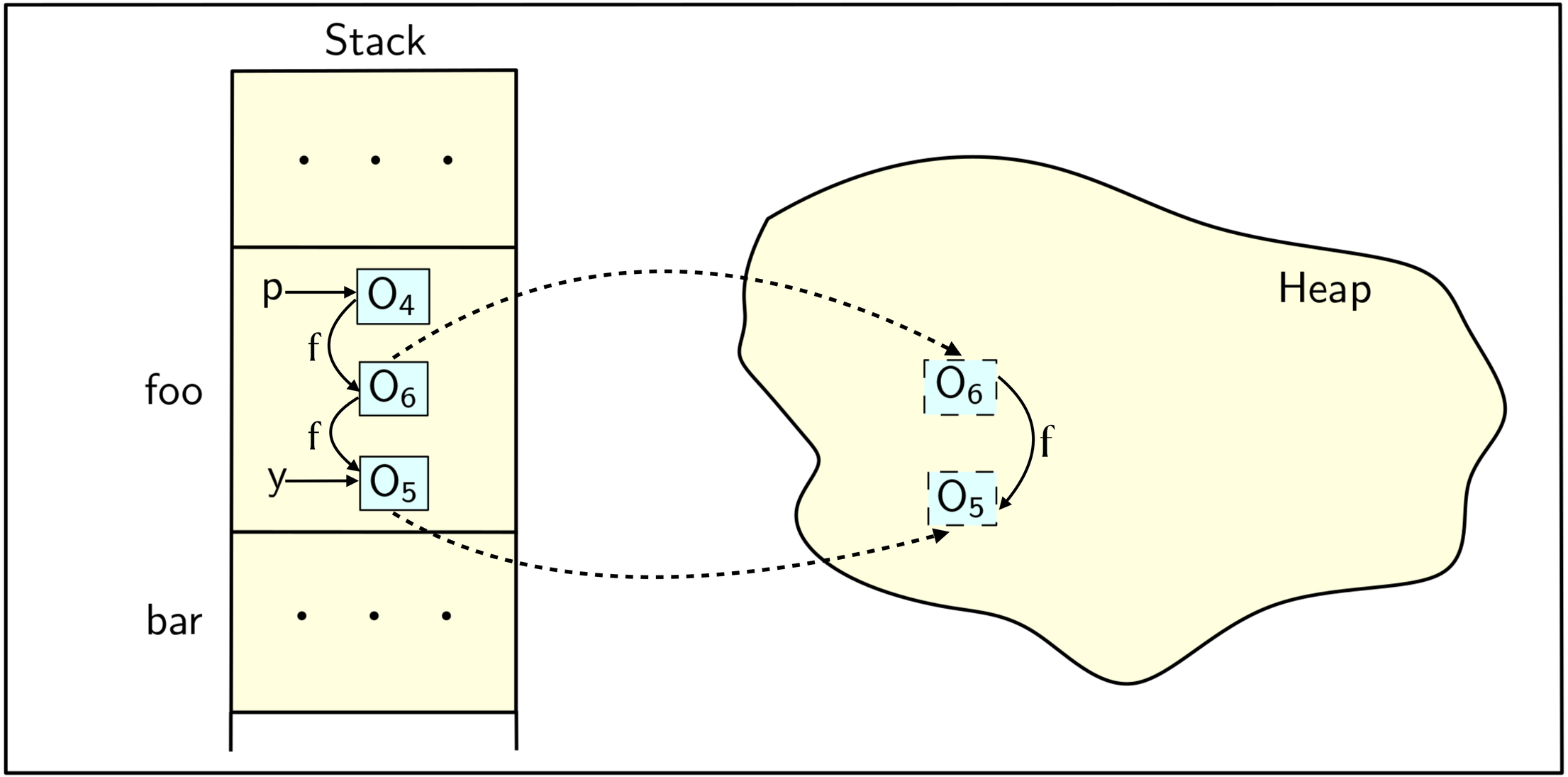
Heapification



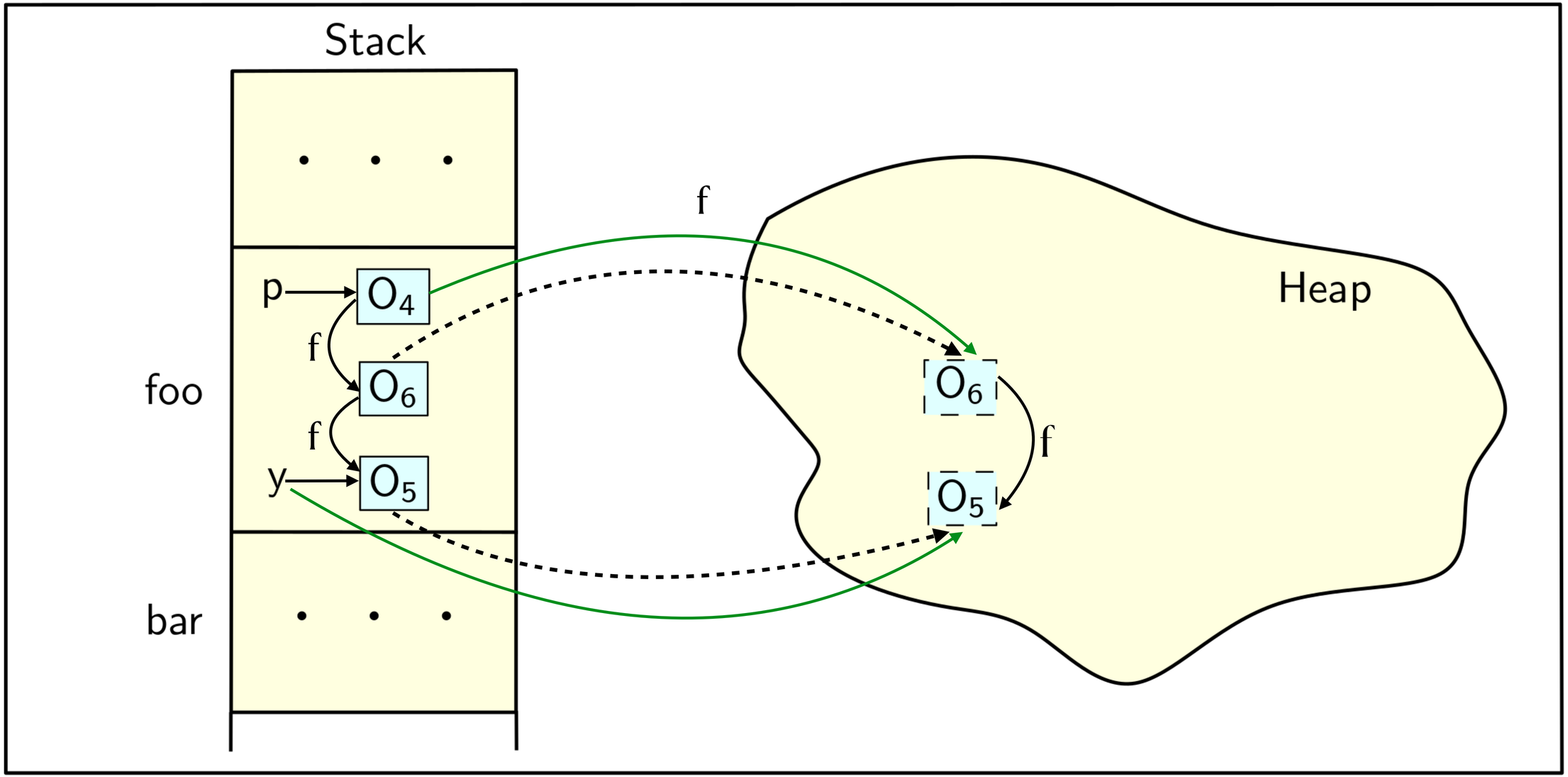
Heapification



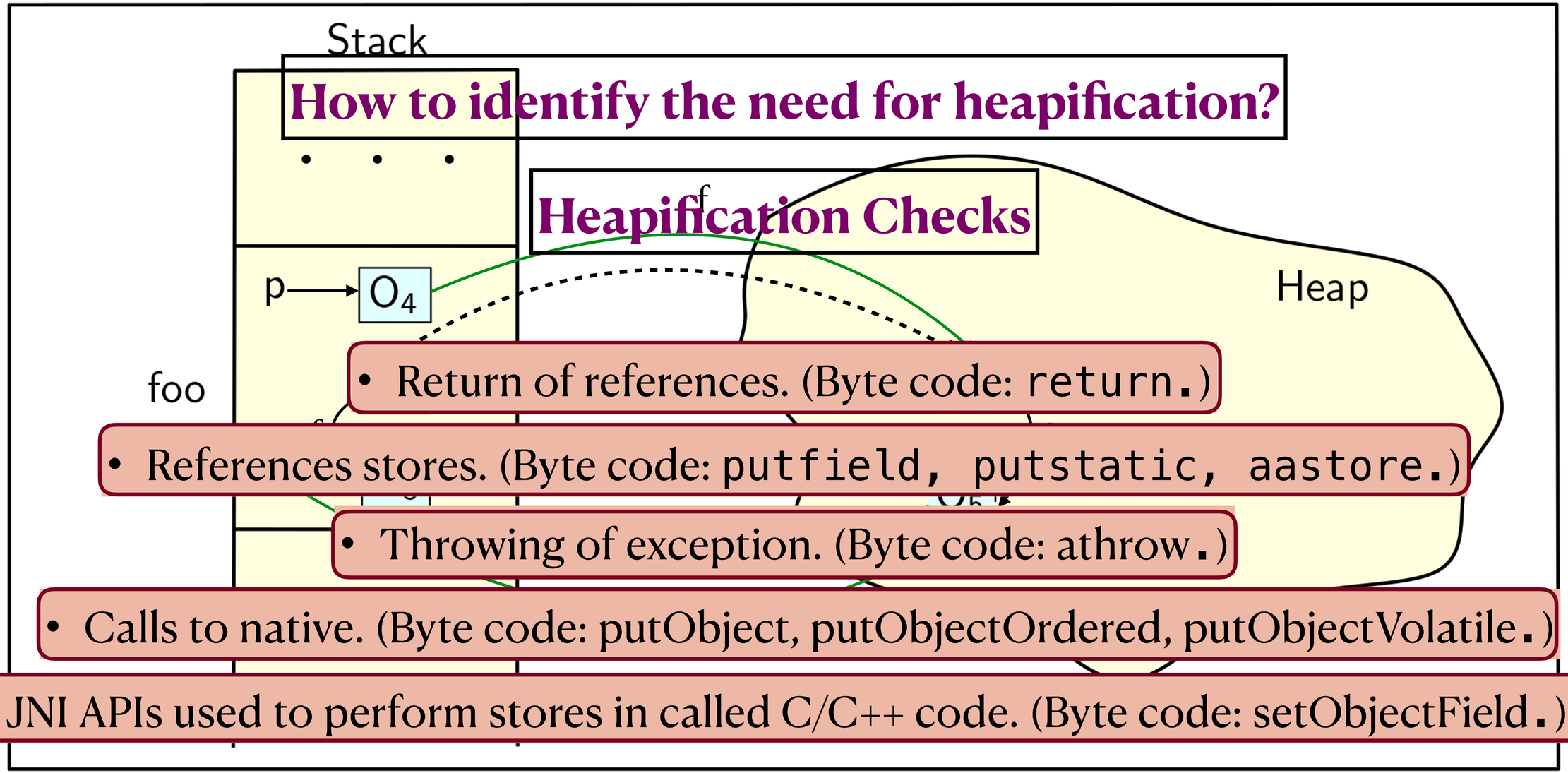
Heapification



Heapification

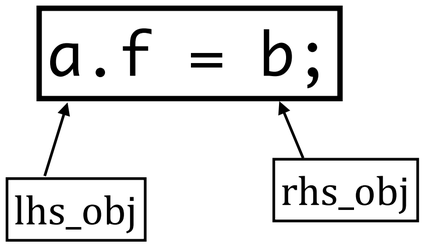


Heapification



Checking the Need for Heapification

```
1 Procedure HeapificationCheckAtStore(lhs, rhs)
2     if rhs object is outside stack bounds then
3         No heapification required.
4     else
          /* The rhs object is present on the stack */
5         if lhs object is outside stack bounds then
6             Heapify starting from the rhs object.
7         else
              /* Both lhs and rhs objects are on the stack */
8             if lhs object has longer life time than rhs object then
9                 Heapify the rhs object.
```

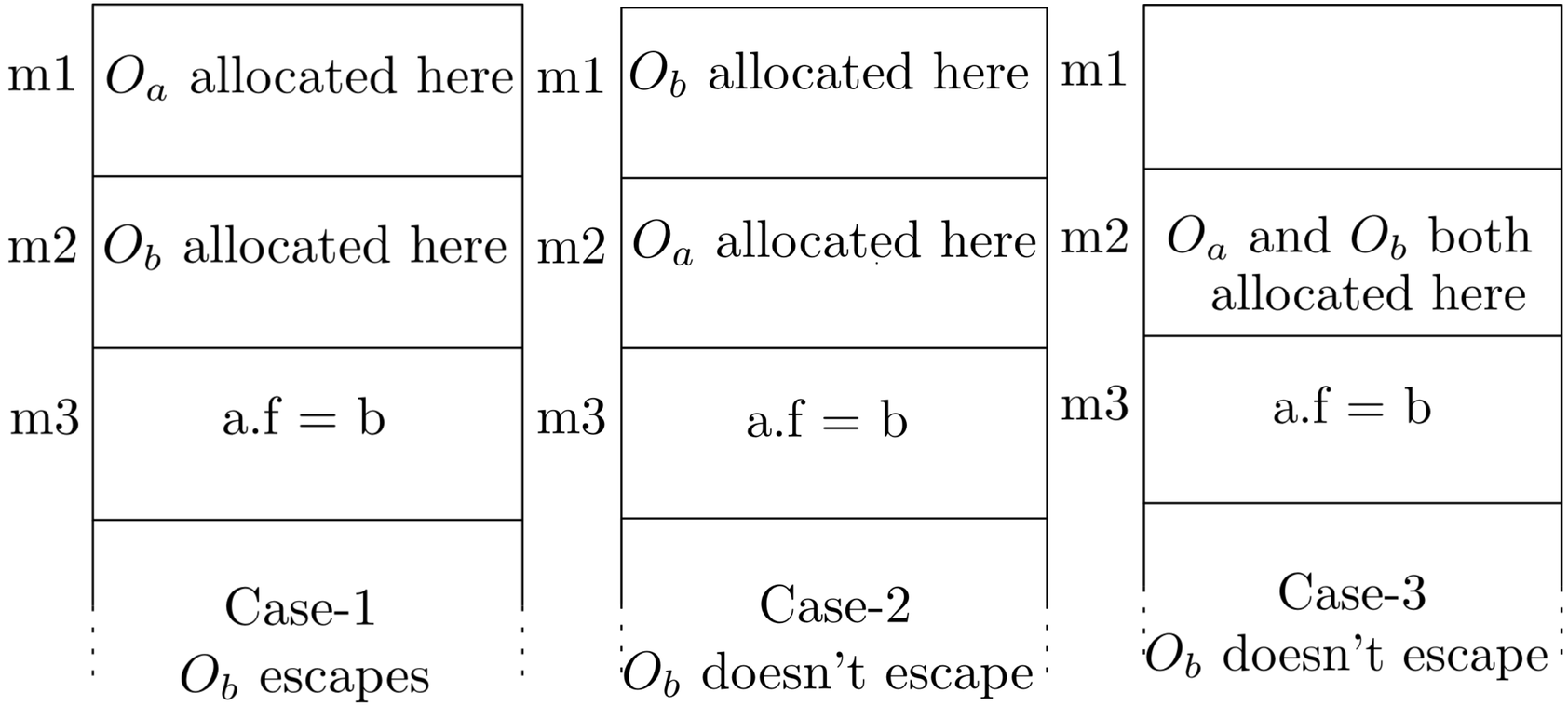


`rhs_obj >= lhs_obj`

Scenarios at Store Statement

```
1. class T {
2.    T f;
3.    void m1() {m2(. . .);}
4.    void m2() {m3(. . .);}
5.    void m3(T a, T b) {
6.        a.f = b;
7.    } /* method m3 */
8.} /* class T */
```

`rhs_obj >= lhs_obj`

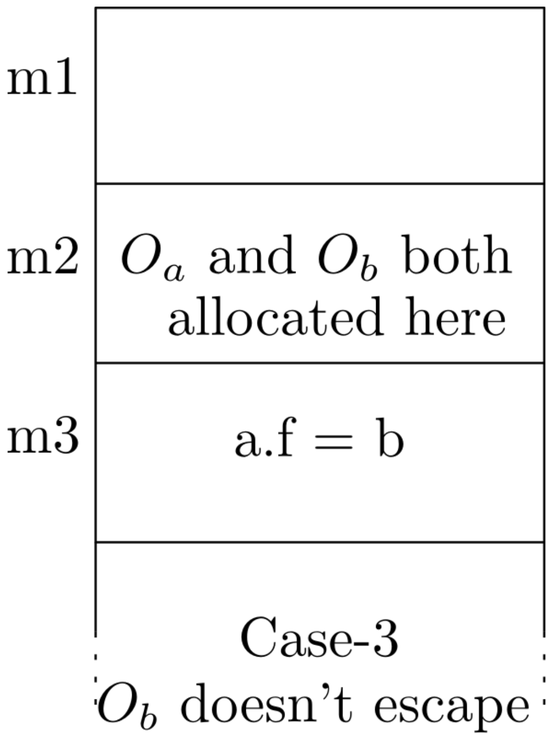


Stack Walk — Costly

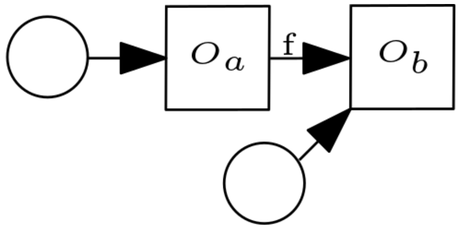
Ordering Objects on Stack

Ordering Objects on Stack

- A simple address-comparison check works majority of times.



- Statically create a partial order of stack-allocatable objects.



$[O_b, O_a]$

- Use the stack-order in VM to re-order the list of stack allocated objects.
- Reduces cost of heapification checks.
- In case of cycles — result will not be valid only for one store statement. Stack Walk

Implementation and Evaluation

- Implementation:

  Static analysis

  Runtime components:

- Benchmarks:
  - **DaCapo benchmark suites:** (23.10-chopin and 9.12 MRI)
  - **SPECjvm** 2008

- Evaluation schemes:
  - **BASE**: Stack allocation with the existing JIT scheme.
  - **OPT**: Stack allocation with our optimistic scheme.

- Compute:
  - Enhancement in stack allocation.
  - Impact on performance and garbage collection.

Evaluation (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

Evaluation (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

Evaluation (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

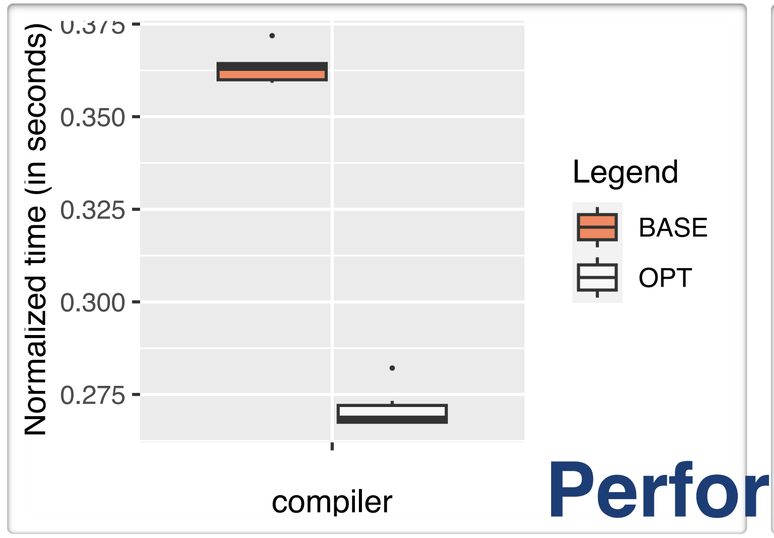
Evaluation (Stack Allocation)

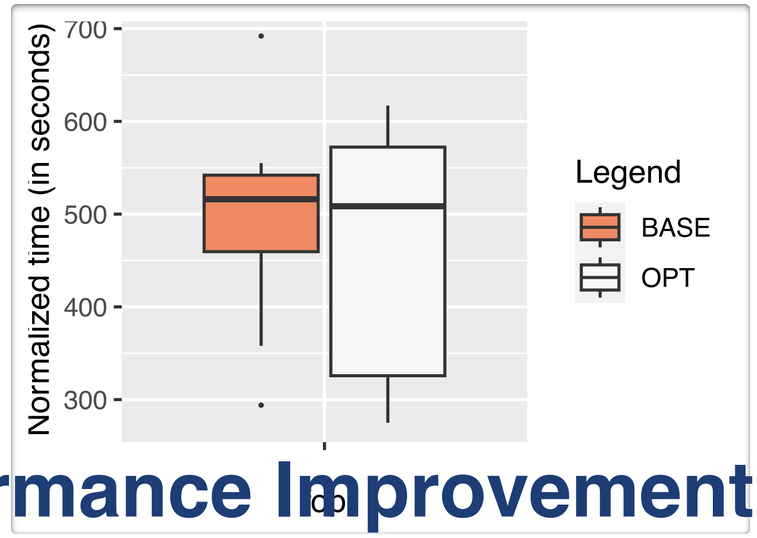
	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	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 (3.09%)	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

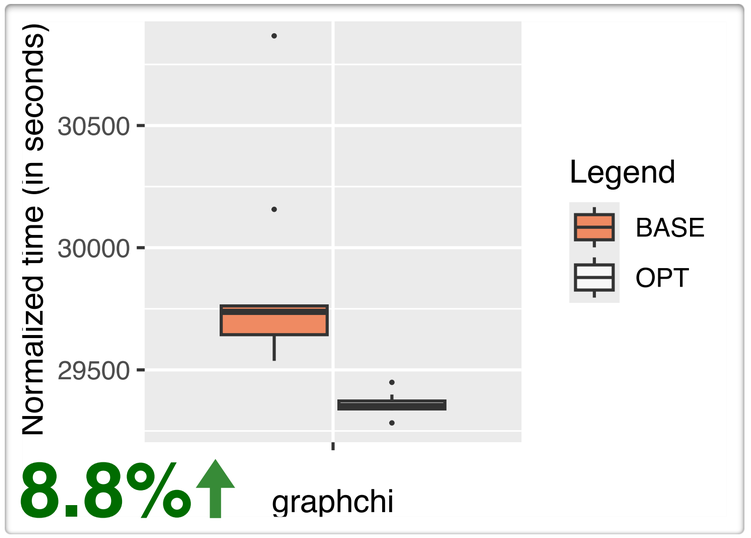
Stack Allocation: 71%↑ Stack Bytes: 54%↑

(Less Heap Allocation)

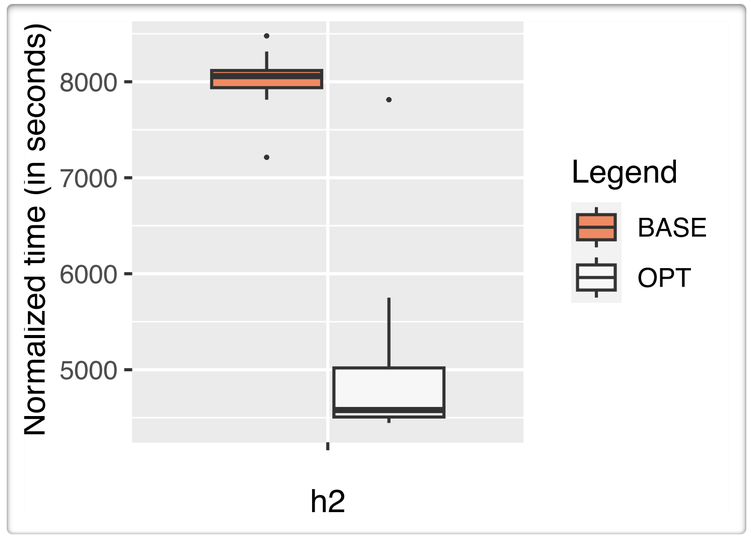
Performance

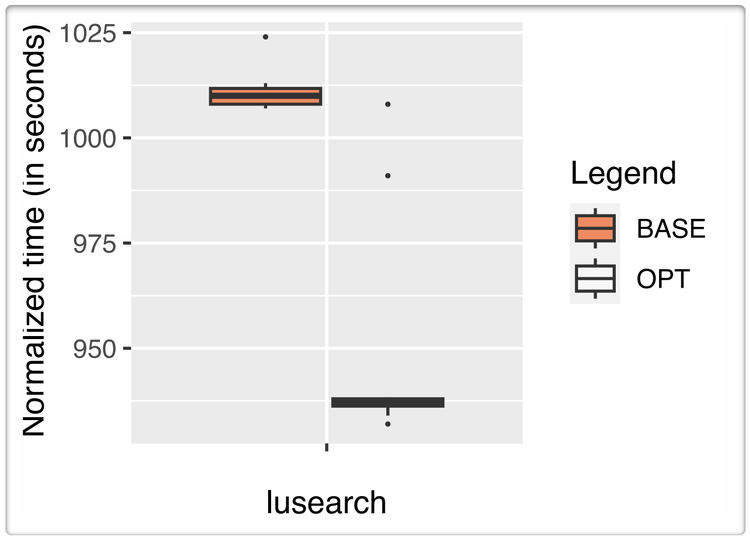


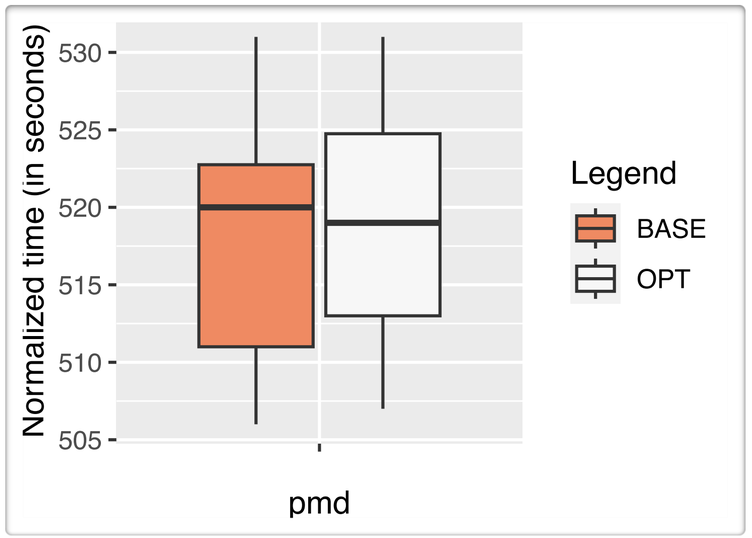




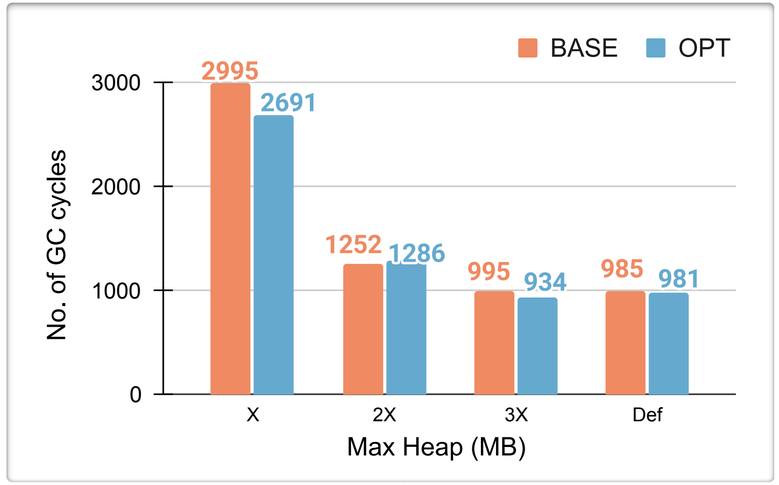
Performance Improvement: 8.8%↑



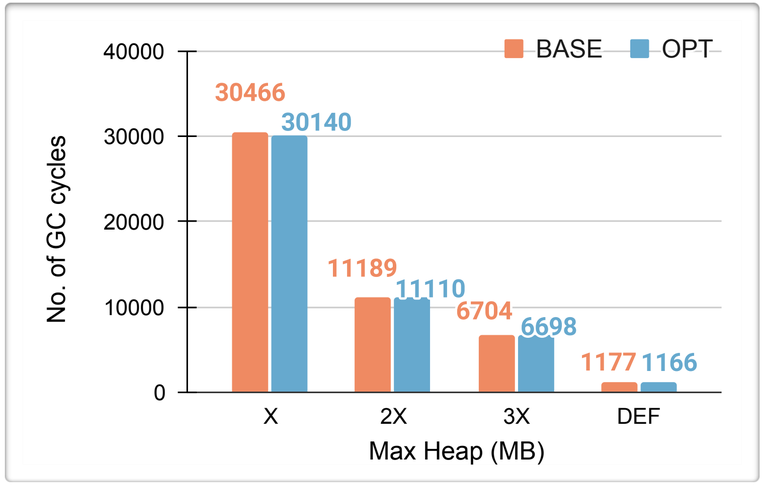




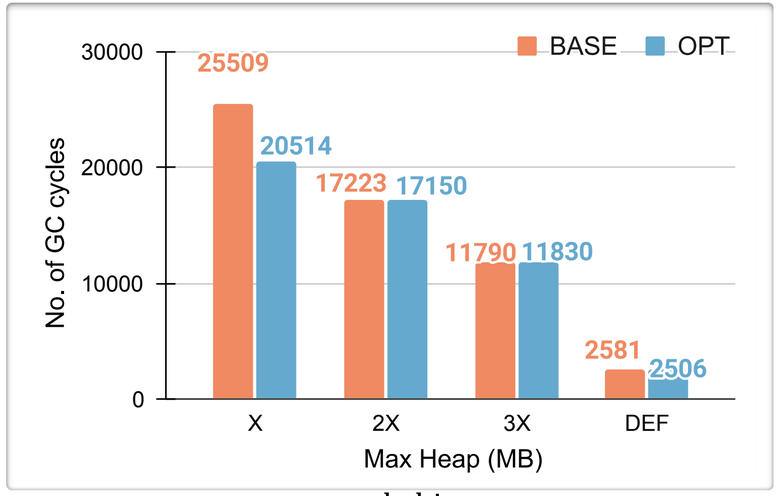
Garbage Collection



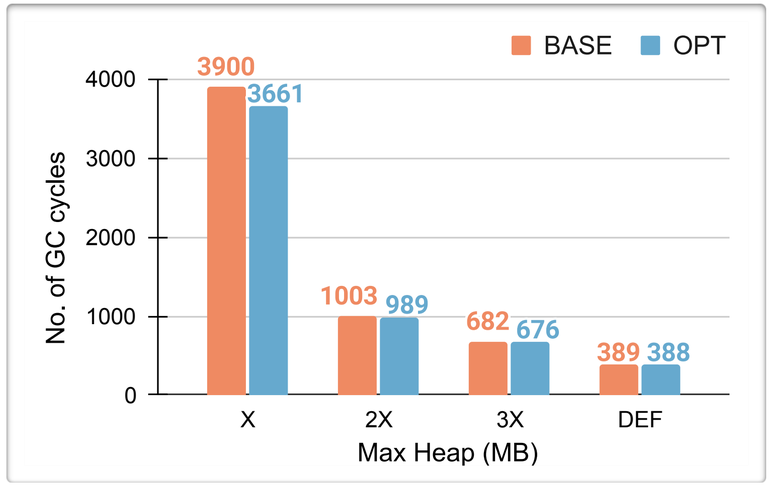
compiler



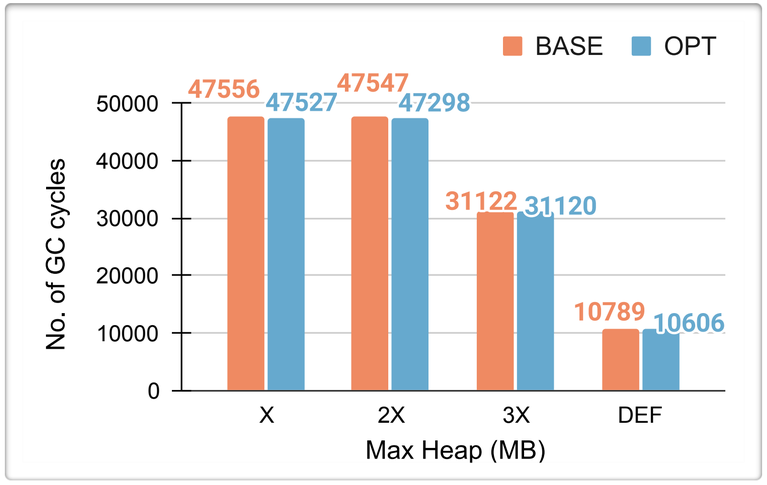
fop



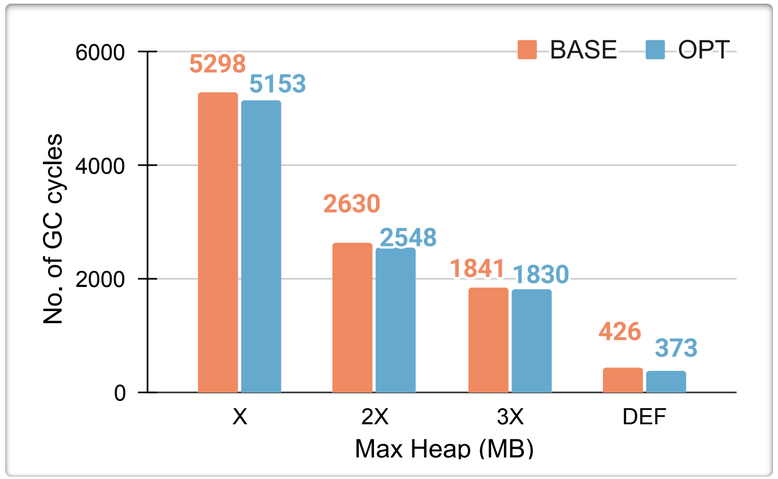
graphchi



h2

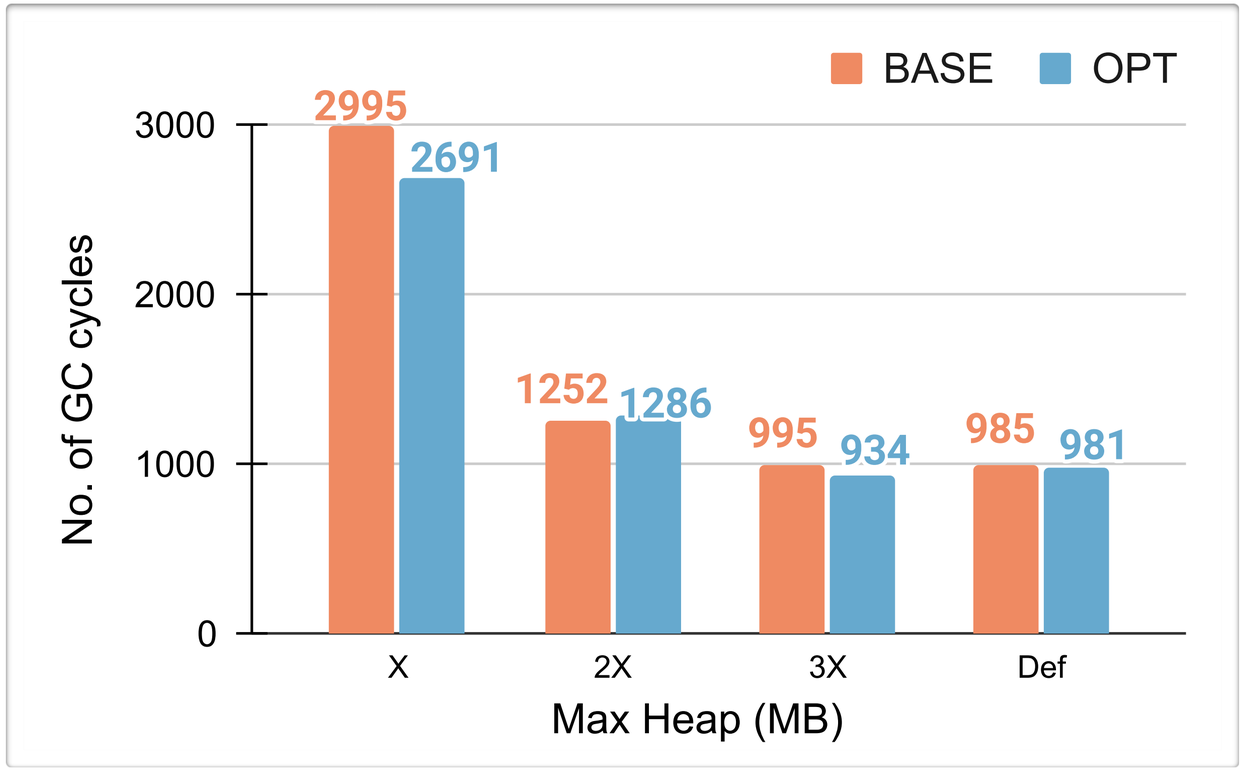


lusearch



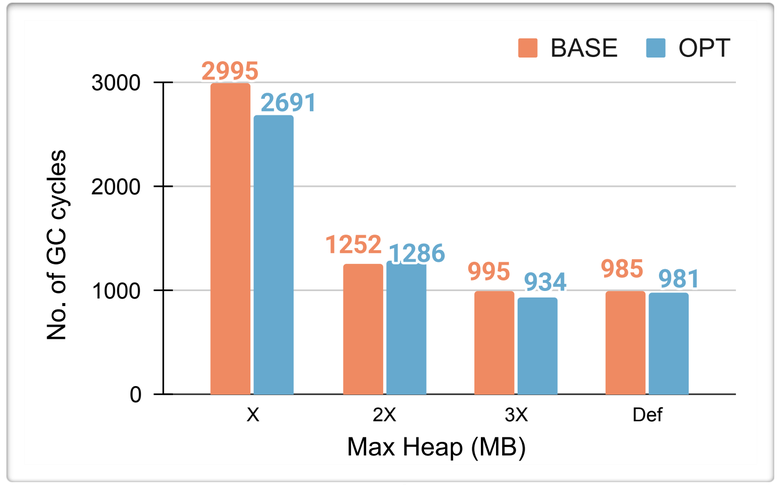
pmd

Garbage Collection

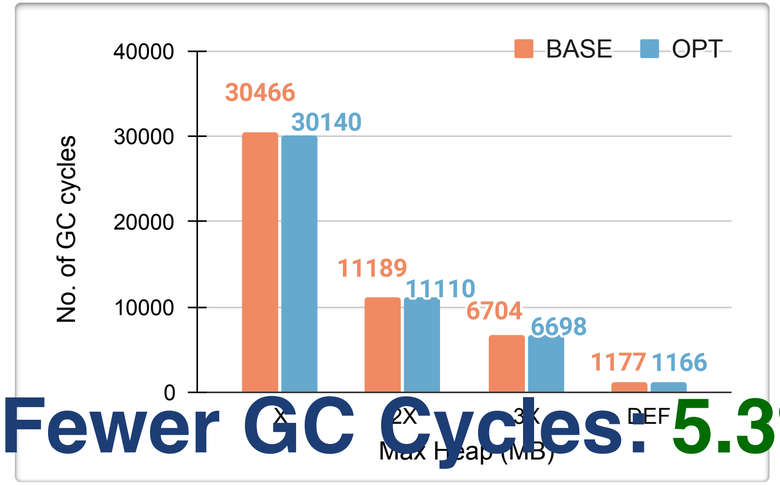


compiler

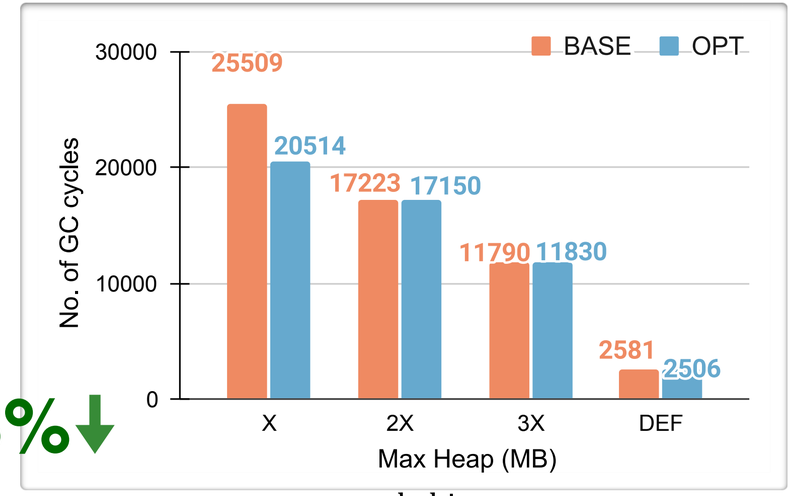
Garbage Collection



compiler

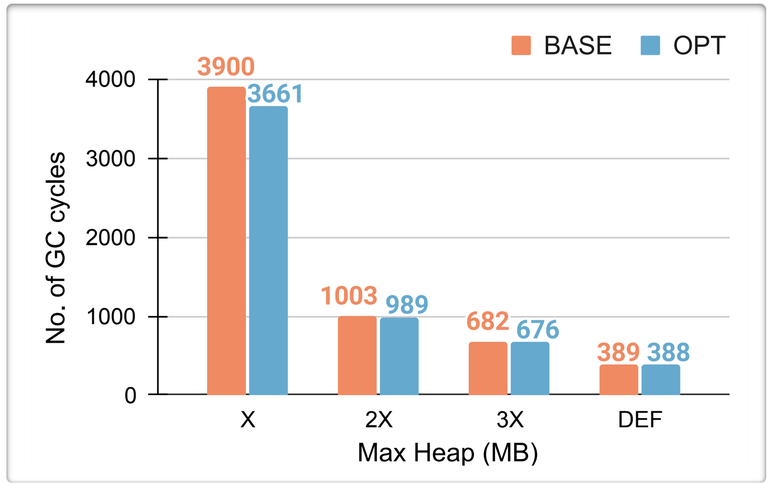


fop

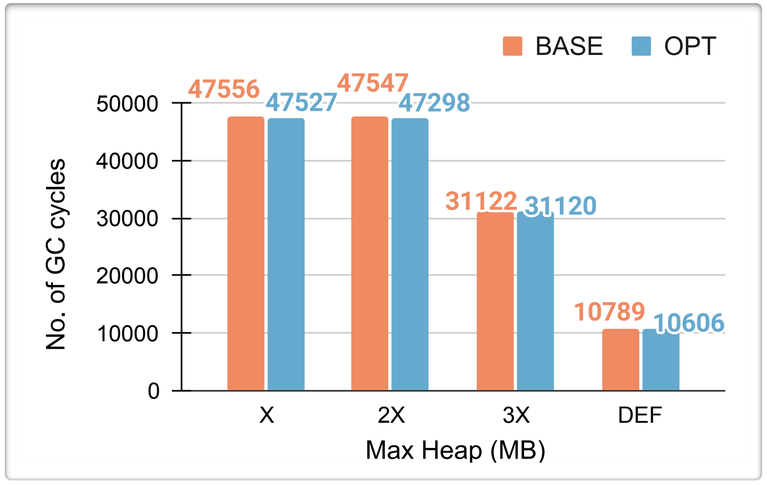


graphchi

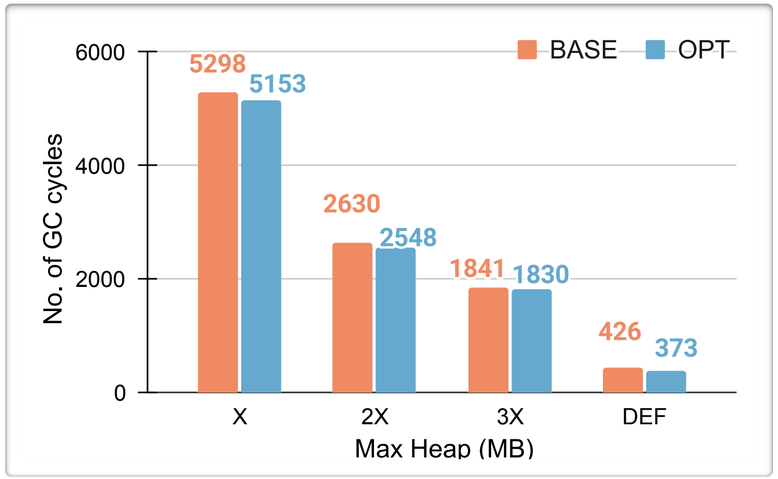
Fewer GC Cycles: 5.3%↓



h2



lusearch



pmd

More in Paper

PLDI 24

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.

- Implementation of opcodes for statements that can cause an object to escape, across JIT & interpreter.
- Simulating longer runs of benchmarks with forced JIT compilation.
- Analyzing allocation sites that lead to high number of allocations.
- Cost of heapification.
- Offline cost.

To summarize and Moving Ahead

- Fallback as heapification allowed us to maintain functional correctness due to the dynamism offered by the Language/VM
- Overall, one of the first approaches to soundly and efficiently use static (offline) analysis results in a JIT compiler!

Is this the best we can do for stack allocation?

Can we go more aggressive?

Recent Works on Static + Dynamic 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 use AOT-analysis results to perform optimizations in JIT Compilers.

Speculative Optimization in JIT Compilers

- Profile Information:
  - Basic invocation, loop invariant values.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Information about loaded subclasses of a given class during execution.
- Inlining Table:
  - Information about the list of methods inlined at various callsites.

Standard Speculative Optimizations

Conservative Fallback

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

OOPSLA 25

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

- Idea:
  - Enrich Static Analysis results with possibility of speculation at run-time.
  - Enable the JIT Compiler to perform speculative optimization based on the static analysis results.

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 */
```

Escapes

⊓ : {B, C } —> [O_5],[O_7] [Escaping]

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 */
```

At runtime:

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

polymorphic_cond

A.foo() [..] [z_14, {B},{O5, O7}]

2. Branching

```
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)

2. Branching

```
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 */
```

At runtime:

- The “if” branch is taken most number of times.

branching_cond

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

3. Method Inlining

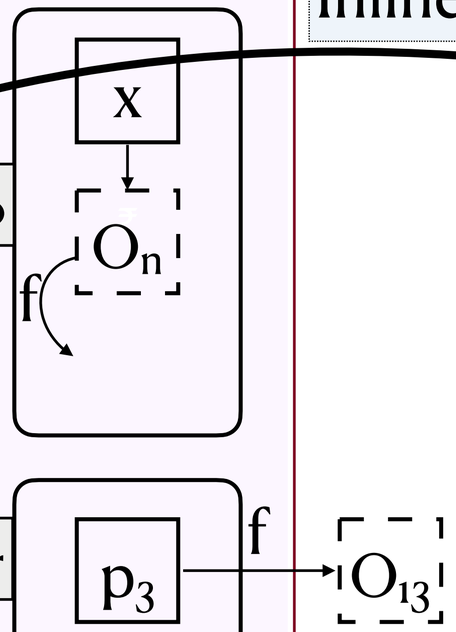
```
1. class A {
2.   void foo() {
3.       . . .
4.       z.bar(x);
5.       r.foobar(p, q);
6.   } /* method foo */
7.   void bar(A p1) { . . . }
8.   void foobar(A p2) { . . . }
9. } /* class A */
```

```
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.

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(p_3), {O_{13}}]

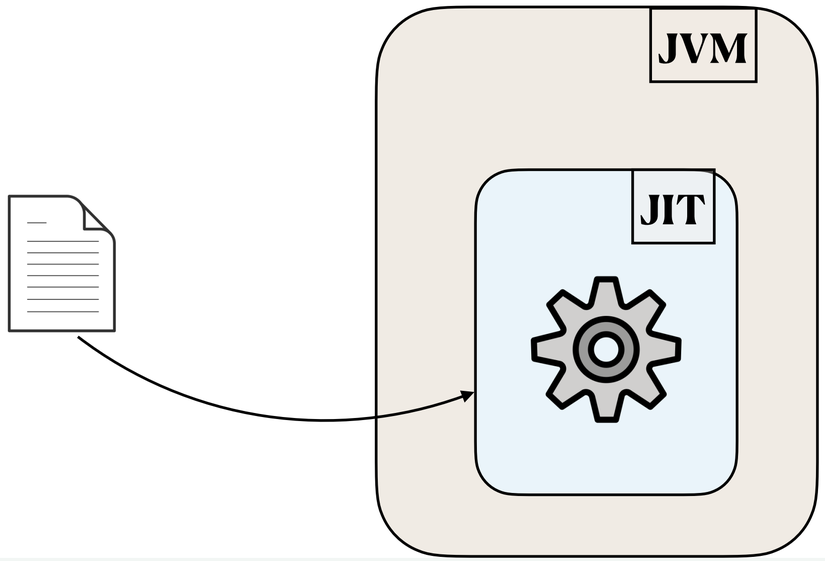
Summary

Direct

Conditional

A::foo() [Direct_Allocation] [polymorphic_cond] [branching_cond] [inlining_cond]

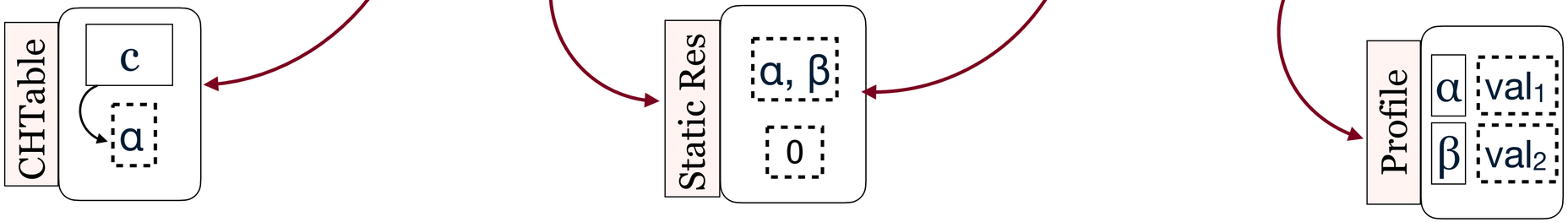
Statically generated results



Algorithm in the JIT Compiler

1. Polymorphic Callsite

```
for each O in JIT_identified_objects(m):
    (CHm[c] ⊆ SAm[O][c]) v ( ∀t ∈ SAm[O][c] Σ CPm(t) > ST)
```

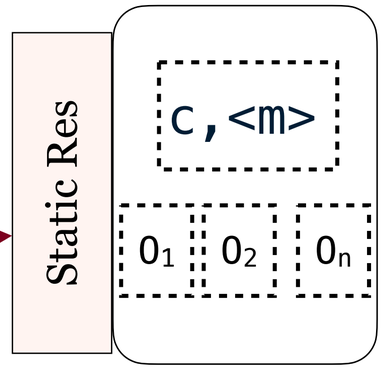


2. Branching Conditions

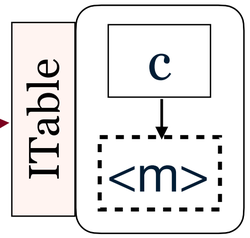
```
for each O in JIT_identified_objects(m):
    (∀ b ∈ SAm[O][br] with Σ BPm(b) > ST)
```

Algorithm in the JIT Compiler

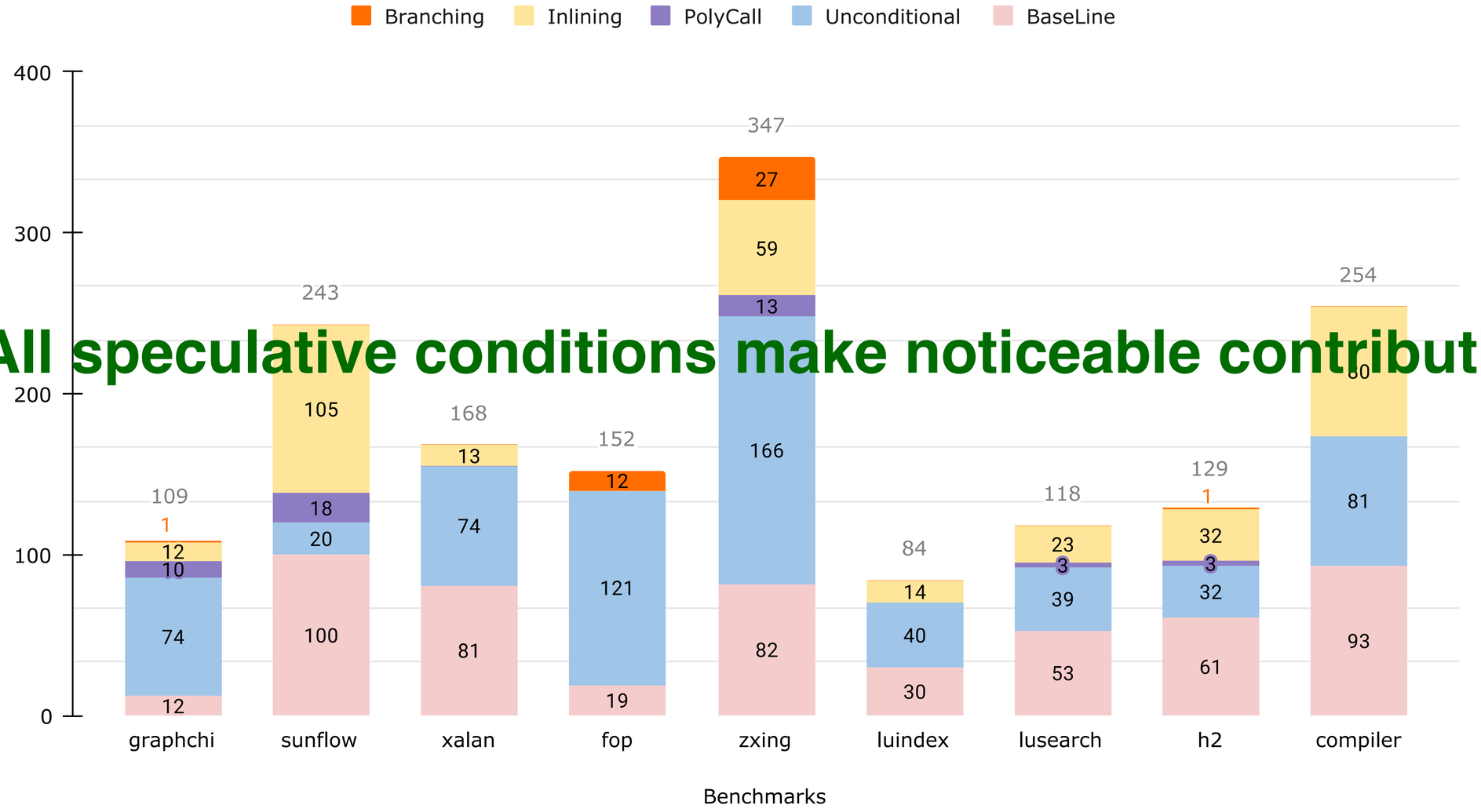
3. Inlining Conditions




```
for each callsite c ∈ CallSitesm:
    Callersc = SAm[c]
    if ∃ n such that (n ∈ ITm[c] ∧ n ∈ Callersc):
        Ostatic = statically_marked_objects(m)
        mark all O ∈ Ostatic
```

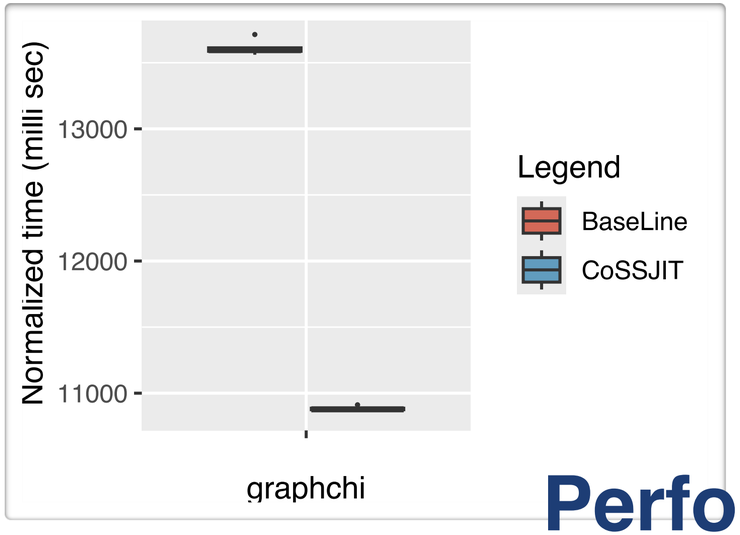


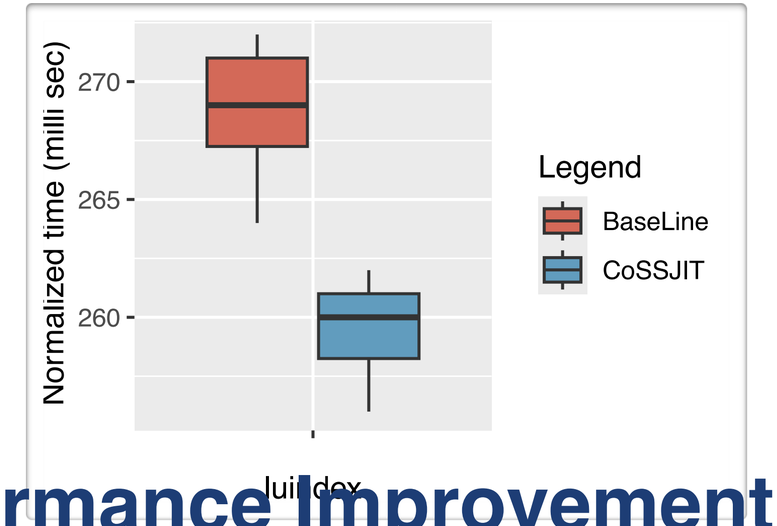
Contributions by different Speculative Conditions

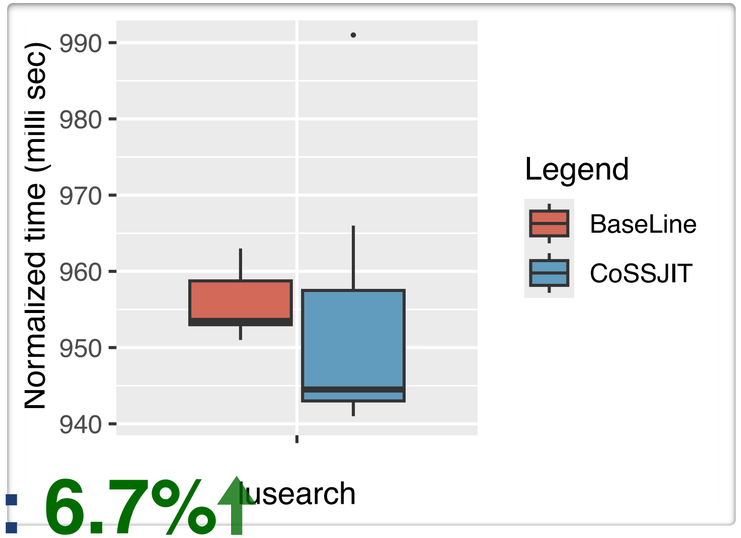


All speculative conditions make noticeable contributions.

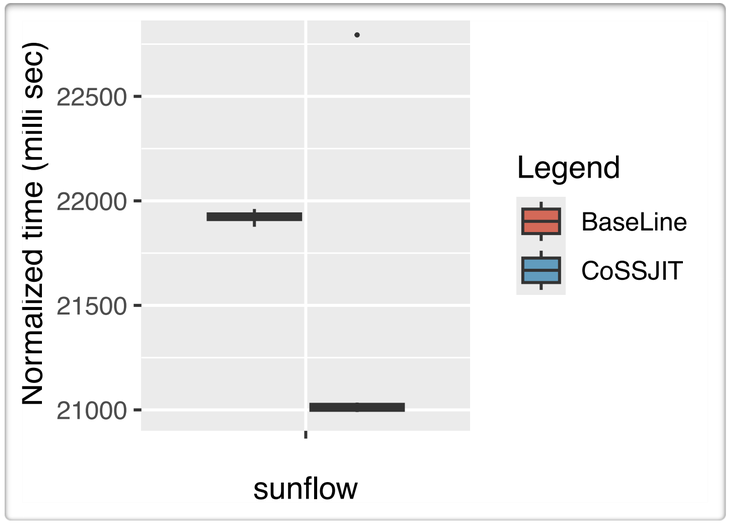
Performance Improvement

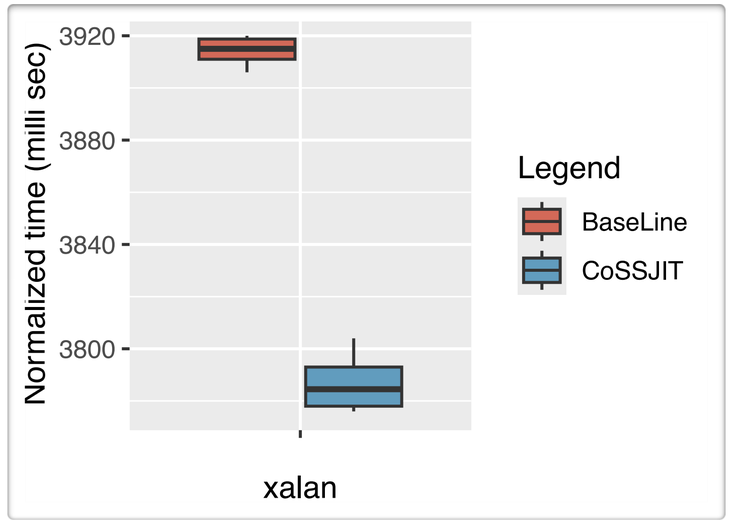


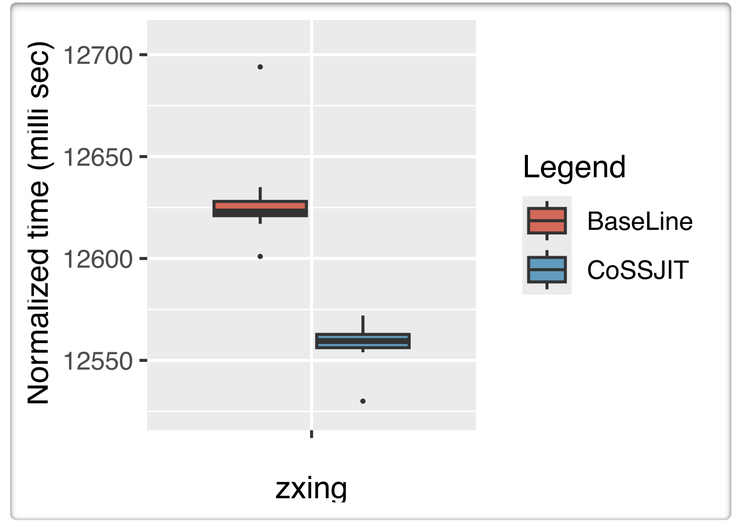




Performance Improvement: 6.7%↑

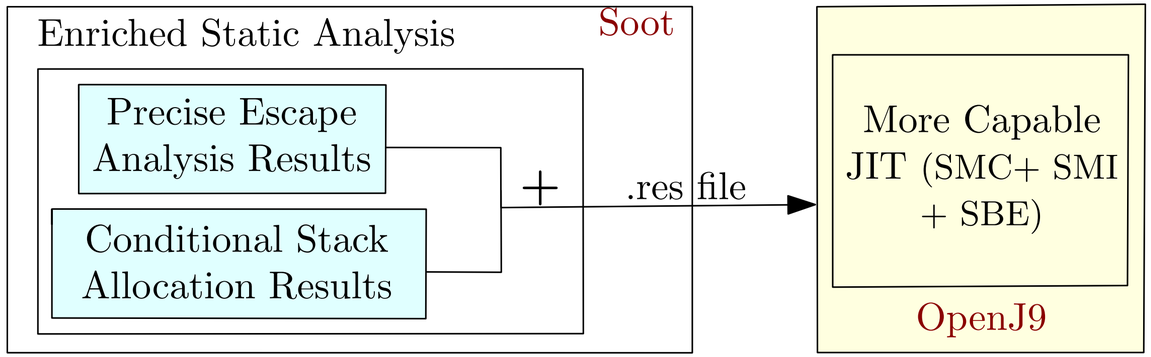






CoSSJIT: Take Aways

- Enriched the static analysis with possibility of speculation at run-time.
- Mechanism in the JIT compiler to incorporate the conditional static analysis results.



- Overall, one of the first approaches that strike a balance between static analysis and JIT speculation, harnessing the best of both the worlds.

Thank You !!