CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

Research Highlights in Programming Languages (RHPL 2025)

Aditya Anand*, Vijay Sundaresan†, Daryl Maier† and Manas Thakur*

*IIT Bombay, †IBM Canada

18th December 2025

Brief Introduction

- Program Analysis and JIT (Just-In-Time) compilers for Java like languages.
- JITs prioritize fast decisions over precise analysis, resulting in conservative optimization.

- Idea: Staged Static+Dynamic analysis for Managed Runtimes.
    - Offload the costly analysis to static time.
    - Use analysis results in the JIT to enable additional optimizations.

Staged Analysis

- Optimization: Object Allocation

Escape Analysis

- By default, objects in Java are allocated on the heap.

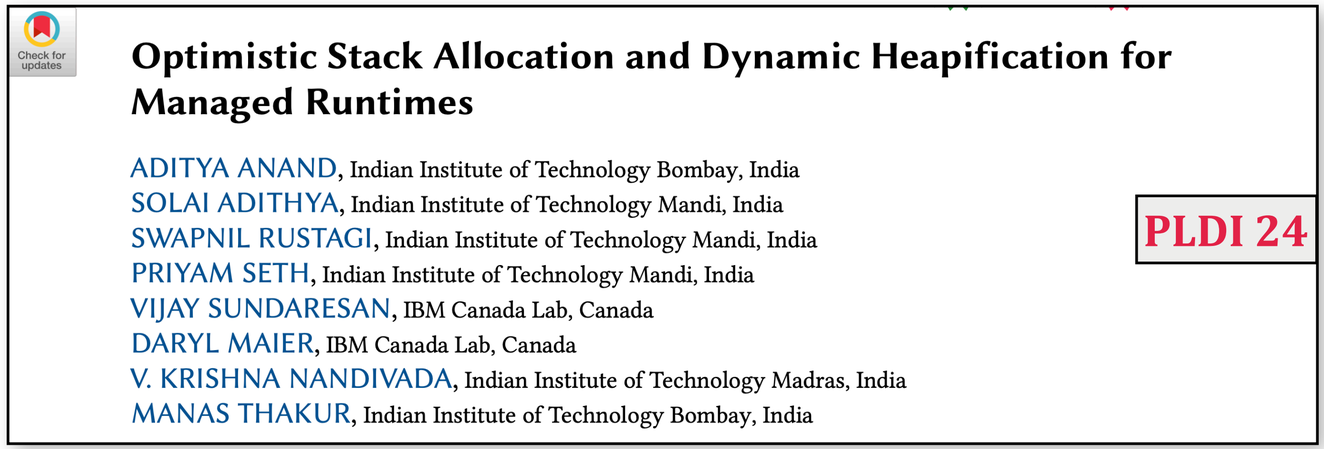
Indirection, Need GC

- Stack Allocation: Allocate method local objects on the stack frame.

Faster access, No GC

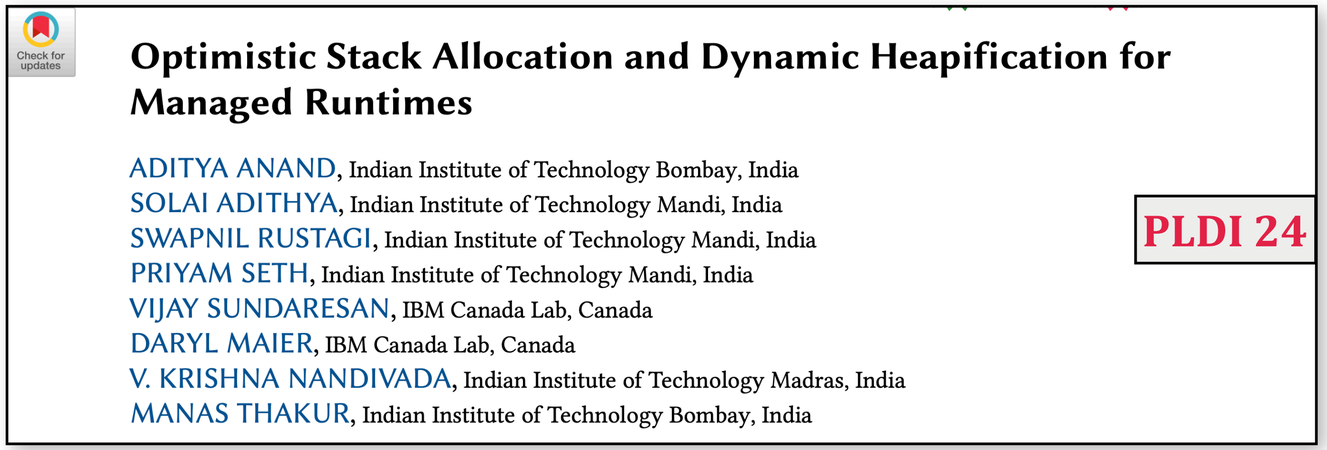
- Perform precise (context-, flow-, field-sensitive) escape analysis statically.

- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.



Optimistic Stack Allocation

- Can we always rely on the static analysis result ??
- Features like Dynamic ClassLoading (DCL), HotCode Replacement (HCR), Callbacks can make your static analysis result go unsound.
- We need to have a fallback mechanism to ensure functional correctness.



Example for Dynamic ClassLoading

Dynamic ClassLoading Example

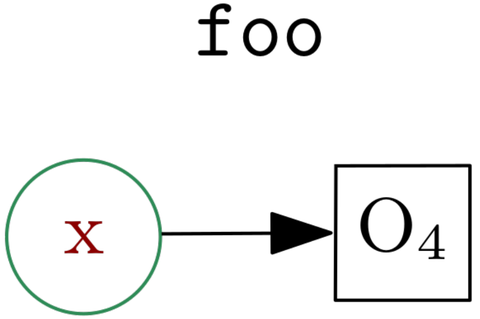
```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

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

Dynamic ClassLoading 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 */
```

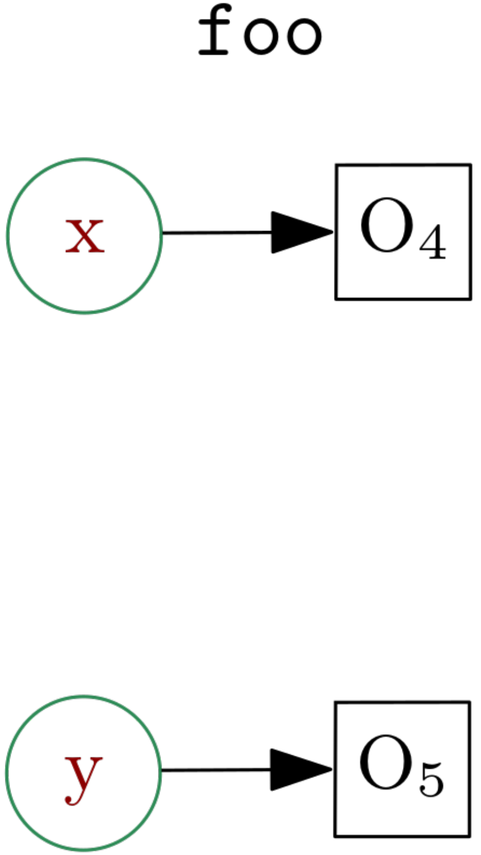
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

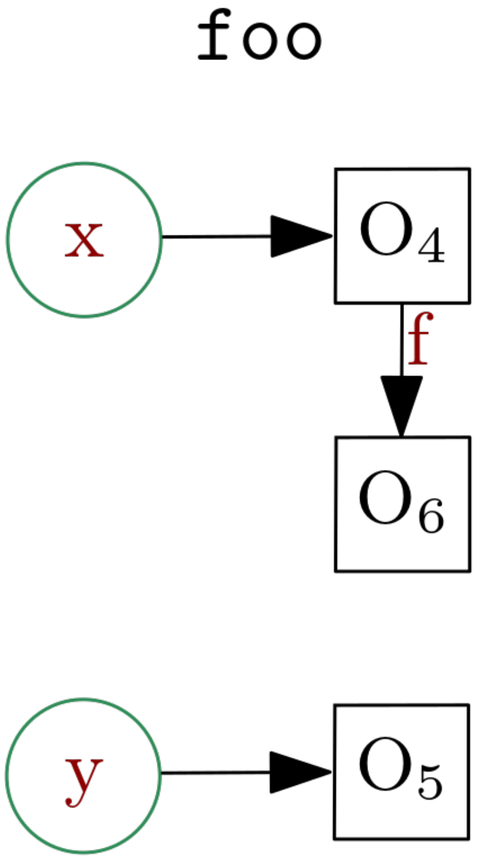
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

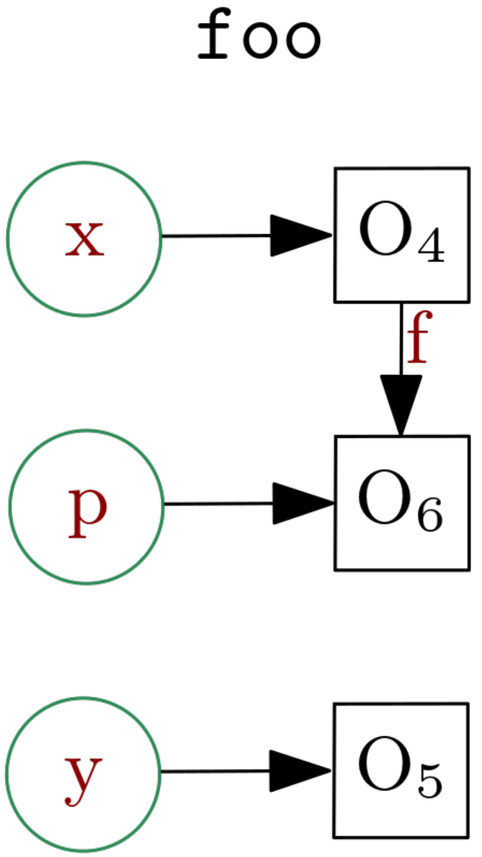
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

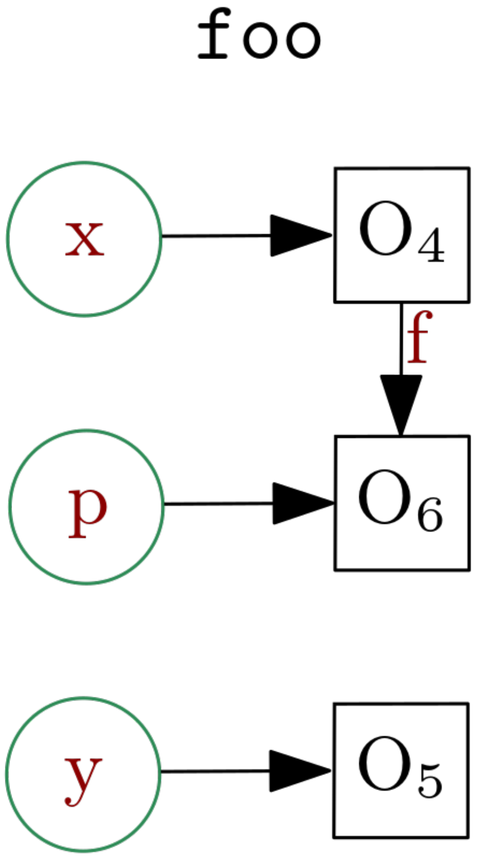
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

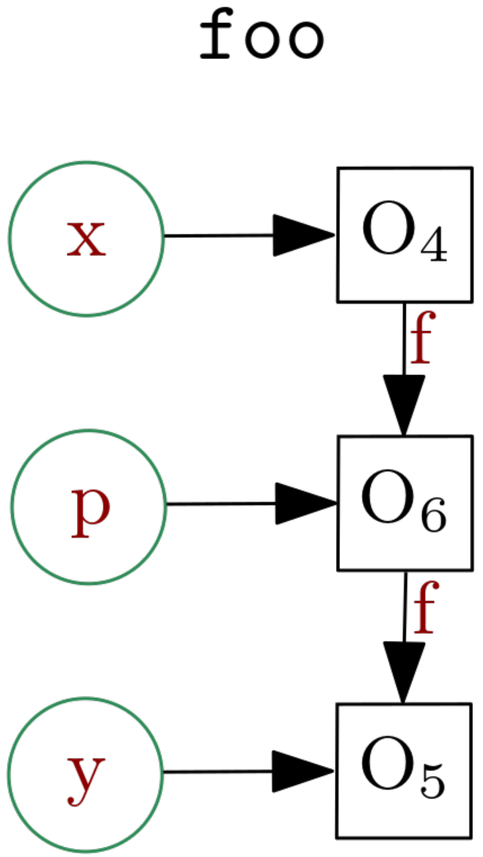
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

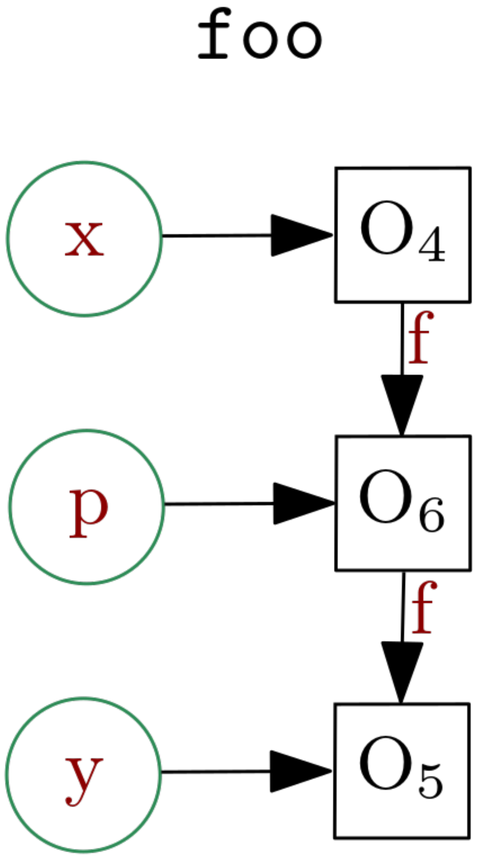
```
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.        p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



Dynamic ClassLoading 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 */
```

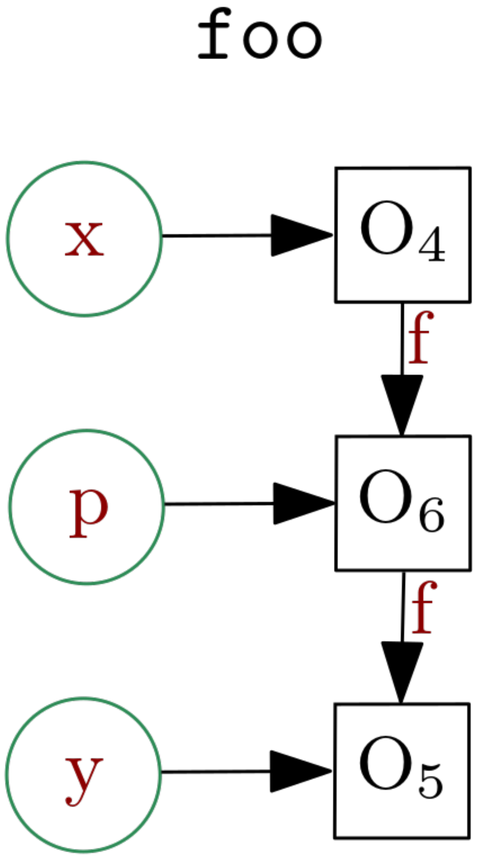


Dynamic ClassLoading Example

```
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



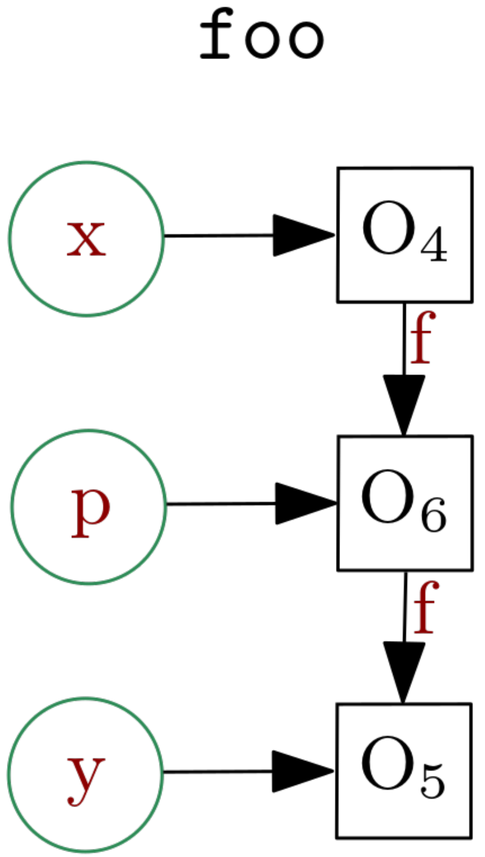
Dynamic ClassLoading Example

```
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

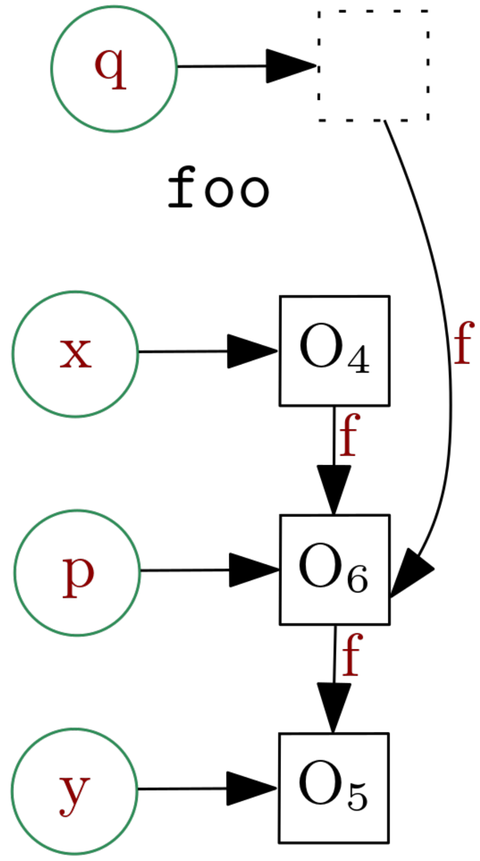


Dynamic ClassLoading Example

```
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 */
```

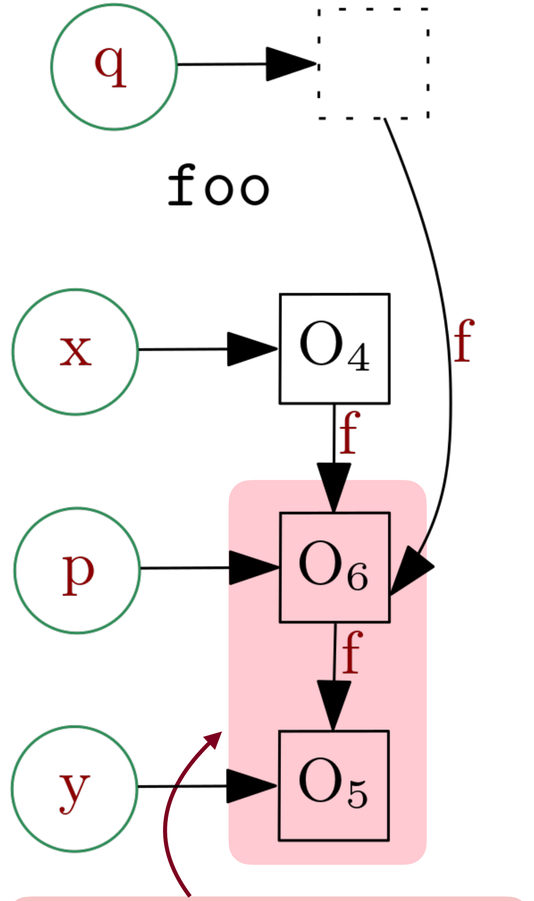


Dynamic ClassLoading Example

```
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 */
```



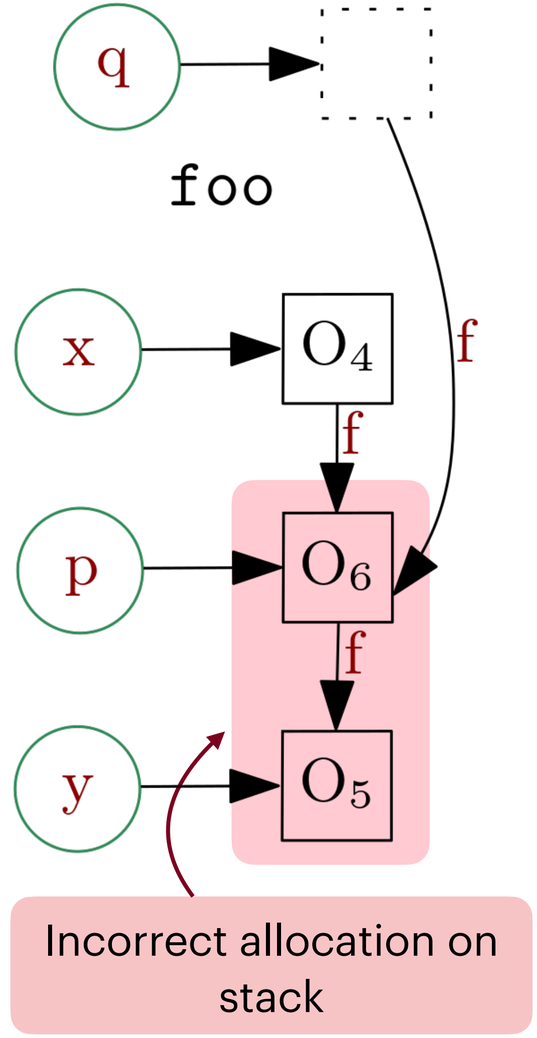
Dynamic Heapification

Dynamic ClassLoading Example

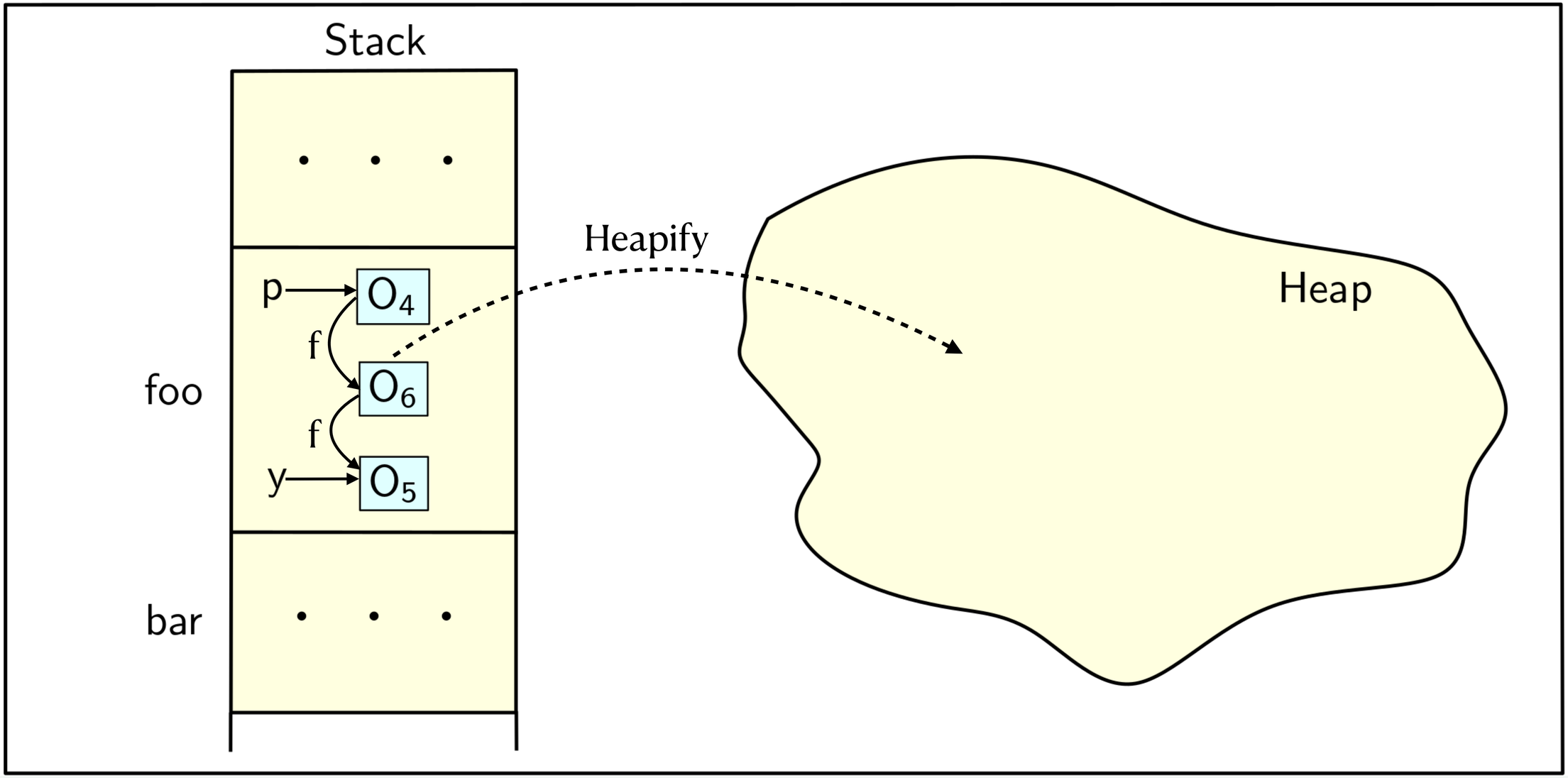
```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.        A x = new A();  // O4
5.        A y = new A();  // O5
6.        x.f = new A();  // O6
7.        A p = x.f;
8.        bar(p, y);
9.        r.zar(p, q);
10.   } /* method foo */
```

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

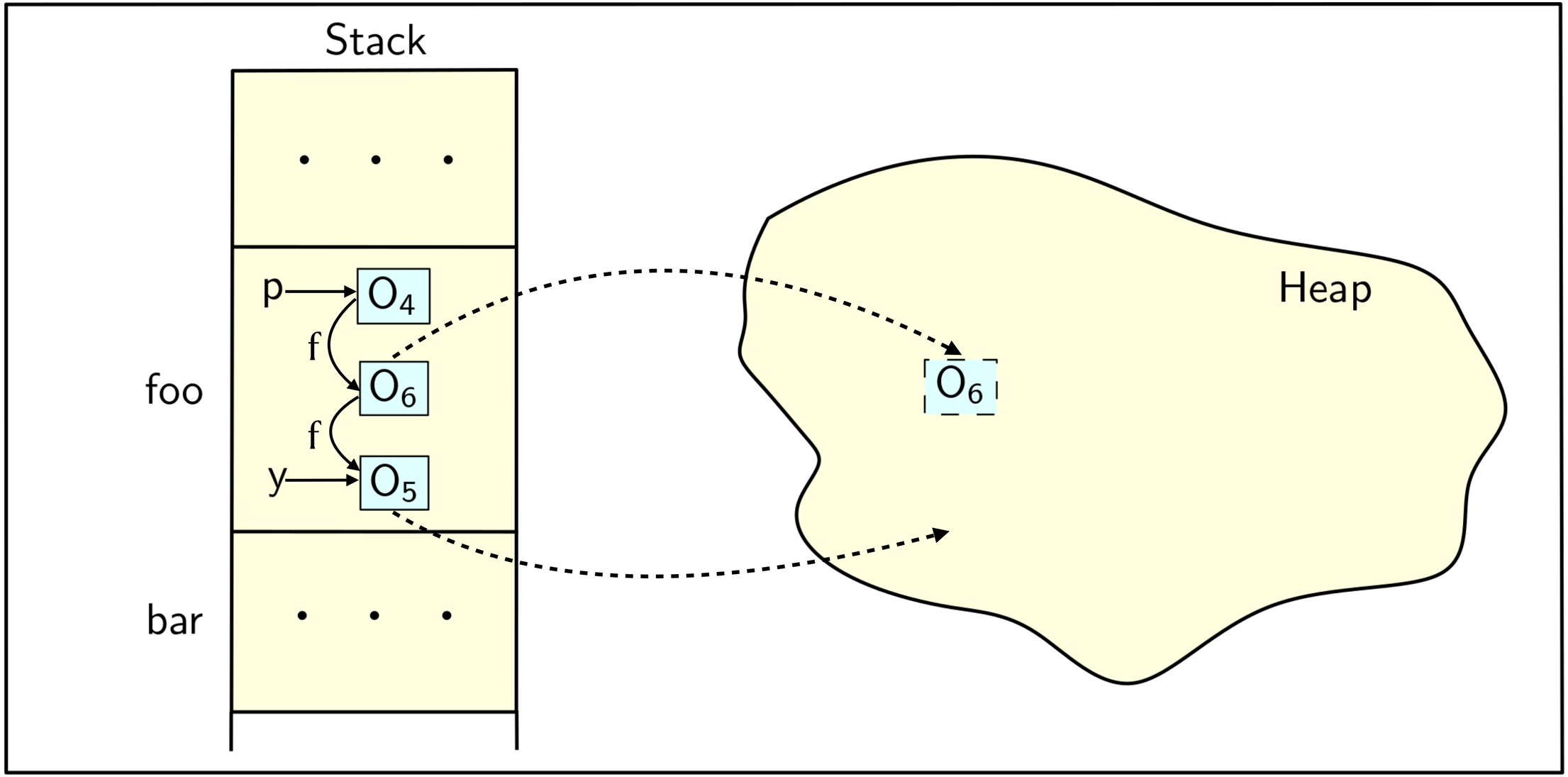
```
16. class B extends A
17.   void zar(A p, A q) {
18.       q.f = p;
19.   } /* method zar */
20. } /* class B */
```



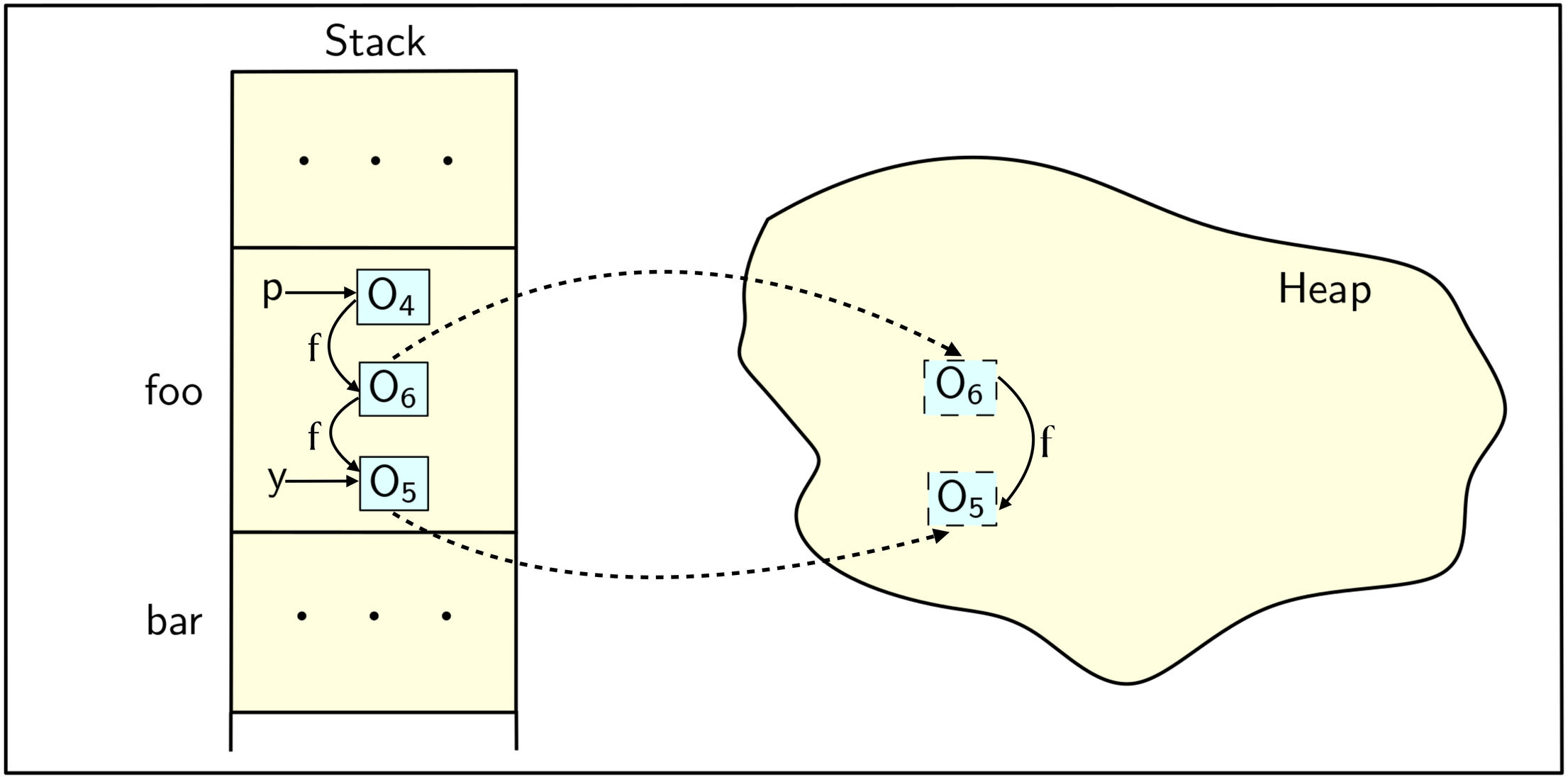
Heapification



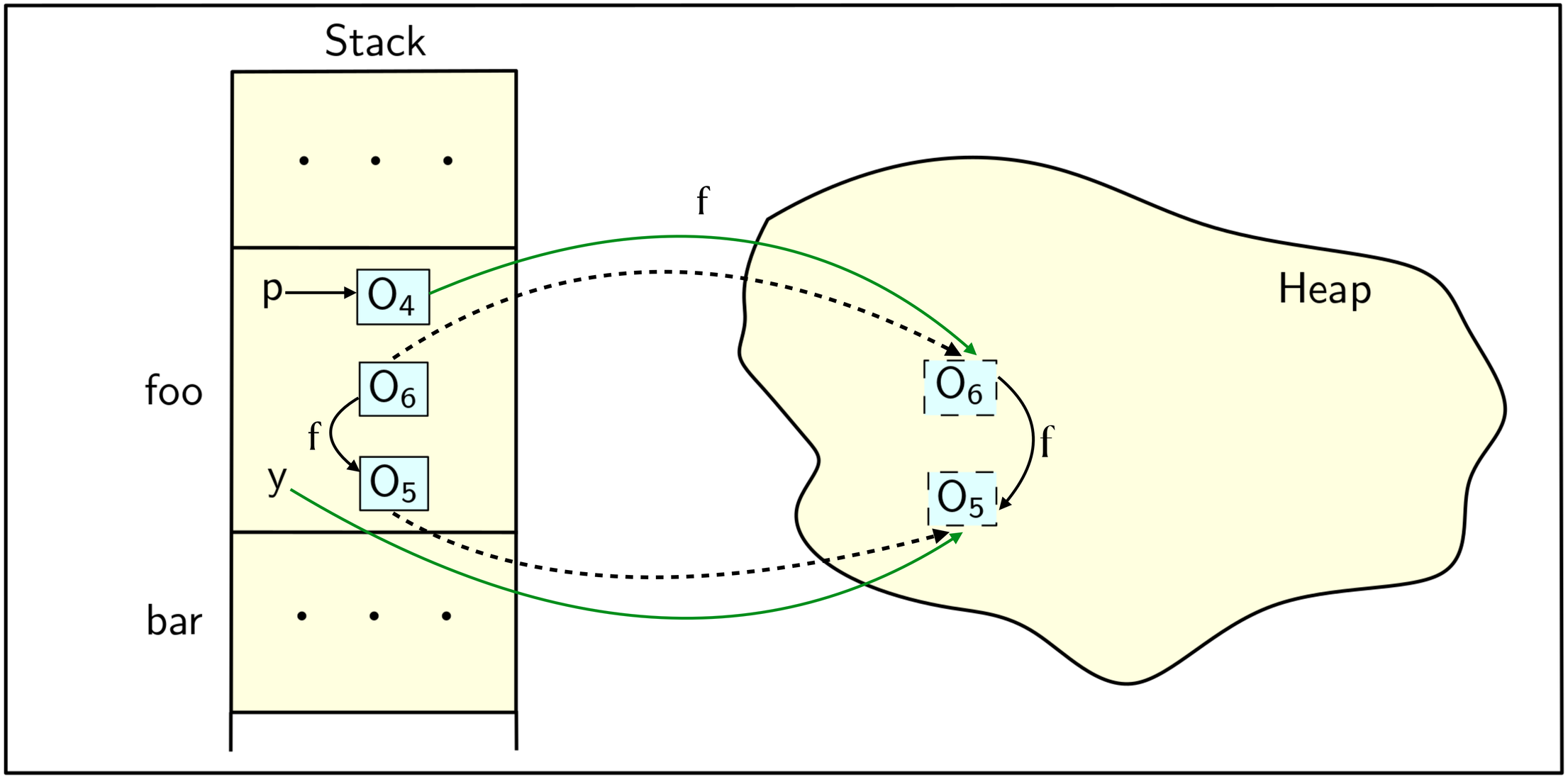
Heapification



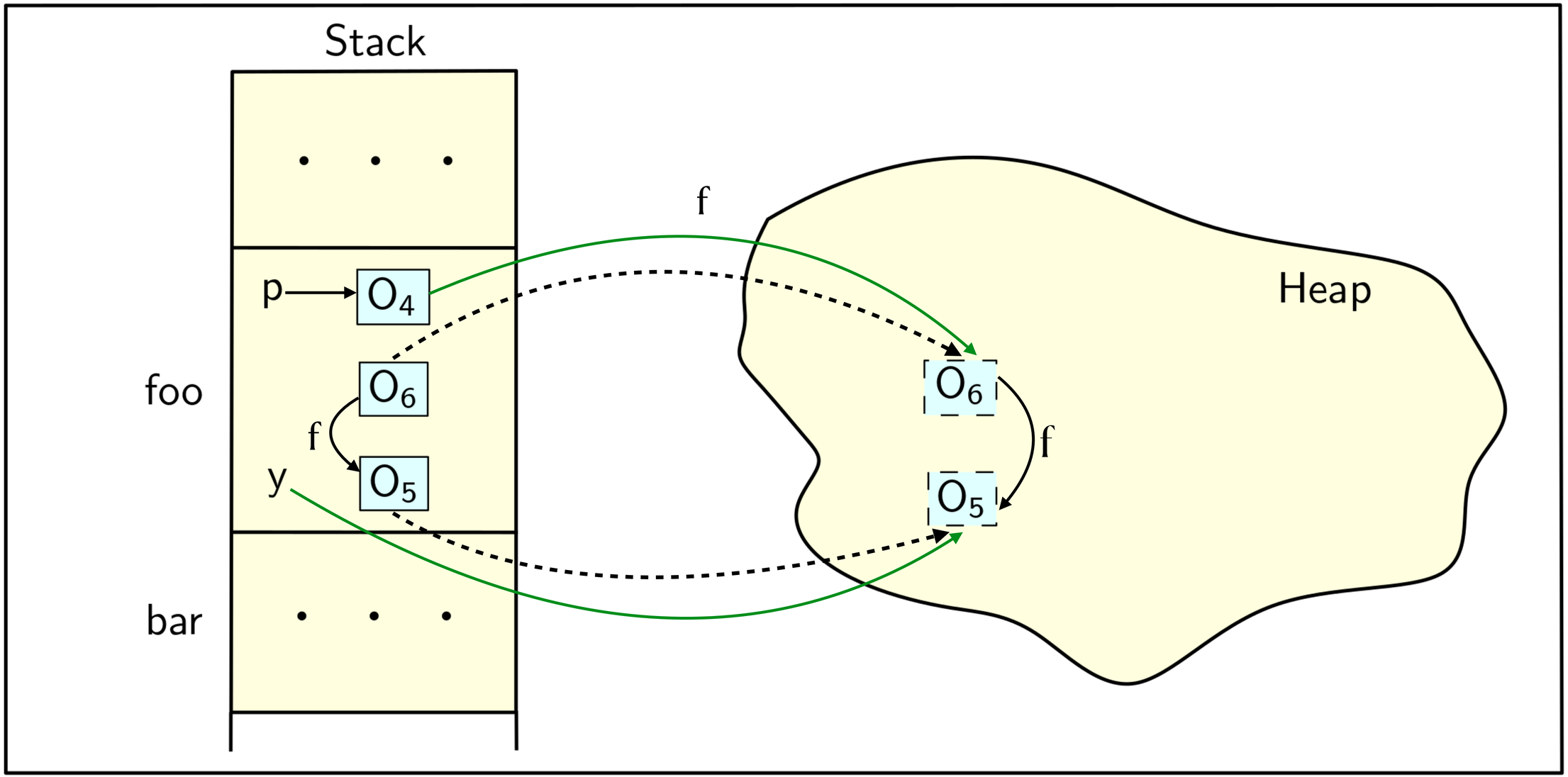
Heapification



Heapification



Heapification



Heapification

How to identify the need for heapification?

Heapification Checks

- Return of references. (Byte code: `return.`)
- References stores. (Byte code: `putfield, putstatic, aastore.`)
- Throwing of exception. (Byte code: athrow.)
- Calls to native. (Byte code: putObject, putObjectOrdered, putObjectVolatile.)
- JNI APIs used to perform stores in called C/C++ code. (Byte code: setObjectField.)

Summary 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

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;      Escapes
23.    }
24. } /* class C */
```

$\sqcap$: {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 */
```

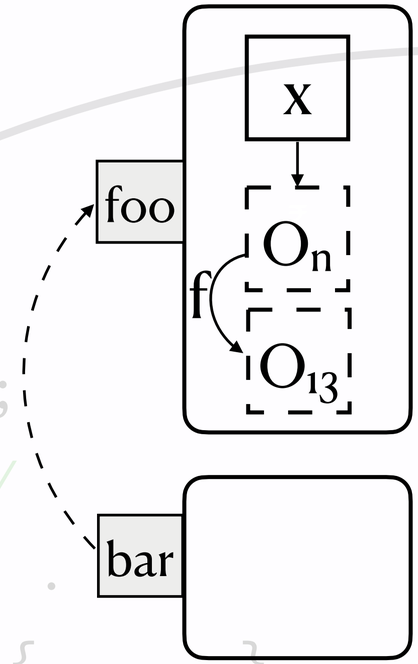
Inline

```
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.   void foobar(A p4) { . . . }
17. } /* class B */
18. class C extends A { . . . }
```

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}}]

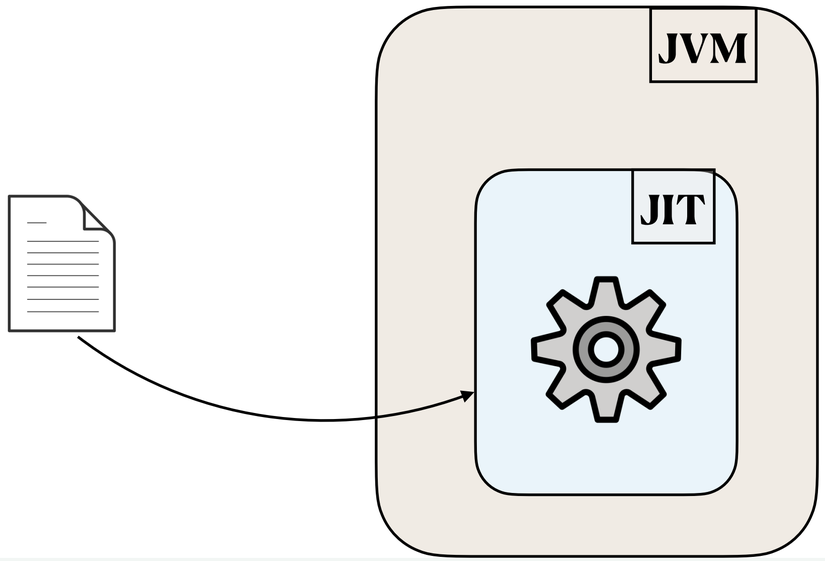
Summary

Direct

Conditional

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

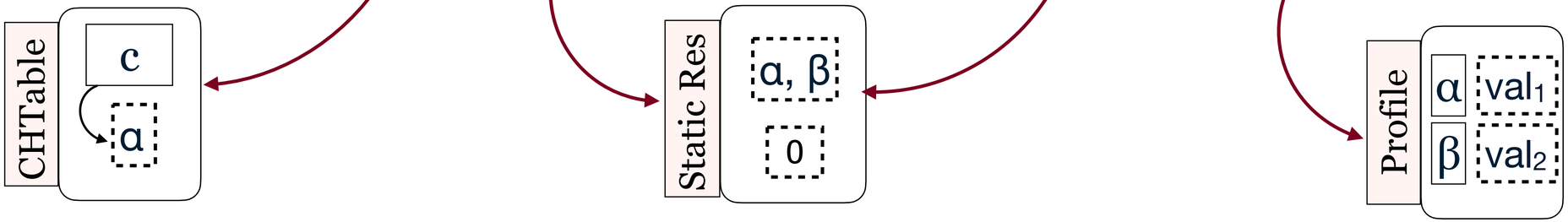
Statically generated results



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)
```

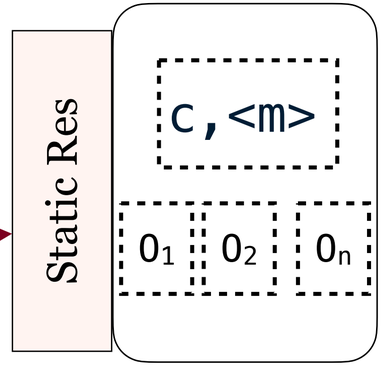


2. Branching Conditions

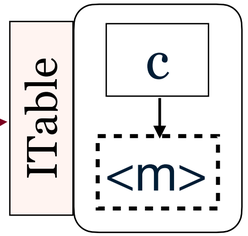
```
for each O in JIT_identified_objects(m):
    (∀ b ∈ SAm[O][br] with Σ BPm(b) > ST)
```

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
```



Evaluation (Setup)

- Implementation:

Static analysis

Runtime components:

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

- Evaluation schemes:
  - **BaseLine**: Stack allocation with the existing JIT scheme.
  - **CoSSJIT**: Stack allocation with our direct+conditional scheme.

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

Evaluation (Stack Allocation)

	Base Scheme (BaseLine)			Conditional Scheme (CoSSJIT)			
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes	Heapify Count
avrora	12	104.2M (38.0%)	3335MB	14	106.5M (39.3%)	3391.4MB	0.4M
compress	8	0.01M (3.29%)	1720MB	18	0.093M (15.5%)	2.75MB	0M
graphchi	12	349M (5.20%)	8327MB	109	1041.1M (14.2%)	20020MB	0.0006M
h2	61	33M (1.02%)	579MB	129	525M (16.2%)	12749MB	6M
luindex	30	4.9M (3.16%)	137MB	84	24.2M (15.4%)	746MB	0.06M
pmd	24	1762M (9.80%)	42295MB	92	1835M (10.2%)	43468MB	0.2M
sunflow	100	1077M (20.0%)	27577MB	243	2286M (34.7%)	56042MB	0.19M
signverify	15	0.24M (0.86%)	6.8MB	40	3.25M (6.34%)	102.2MB	0.5M
zxing	82	24.2M (2.61%)	987.6MB	347	1539M (16.4%)	52796.7MB	0.4M

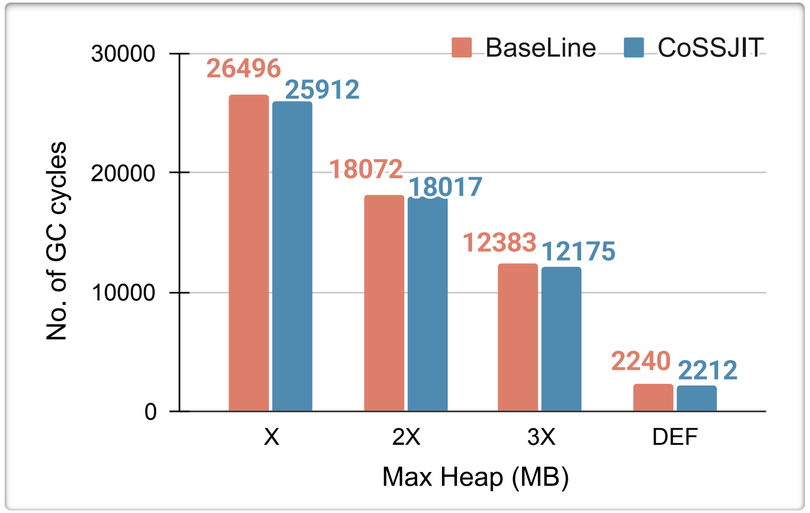
Evaluation (Stack Allocation)

	Base Scheme (BaseLine)			Conditional Scheme (CoSSJIT)			
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes	Heapify Count
avrora	12	104.2M (38.0%)	3335MB	14	106.5M (39.3%)	3391.4MB	0.4M
compress	8	0.01M (3.29%)	1720MB	18	0.093M (15.5%)	2.75MB	0M
graphchi	12	149M (7.20%)	8327MB	10	10[illegible]1M (14.2%)	2[illegible]20MB	0.0006M
h2	61	33M (1.02%)	579MB	129	525M (16.2%)	12749MB	6M
luindex	30	4.9M (3.16%)	137MB	84	24.2M (15.4%)	746MB	0.06M
pmd	24	1762M (9.80%)	42295MB	92	1835M (10.2%)	43468MB	0.2M
sunflow	100	1077M (20.0%)	27577MB	243	2286M (34.7%)	56042MB	0.19M
signverify	15	0.24M (0.86%)	6.8MB	40	3.25M (6.34%)	102.2MB	0.5M
zxing	82	24.2M (2.61%)	987.6MB	347	1539M (16.4%)	52796.7MB	0.4M

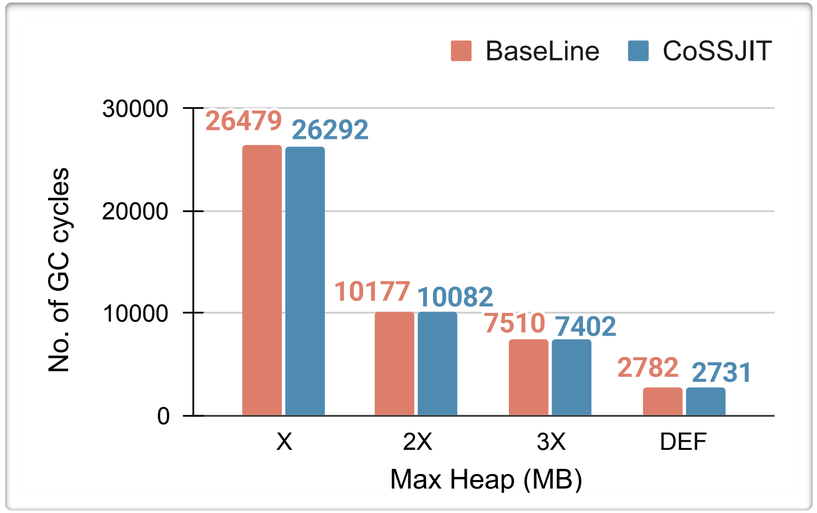
Stack Allocation: 1.4x↑ Stack Bytes: 5.7x↑

(Less Heap Allocation)

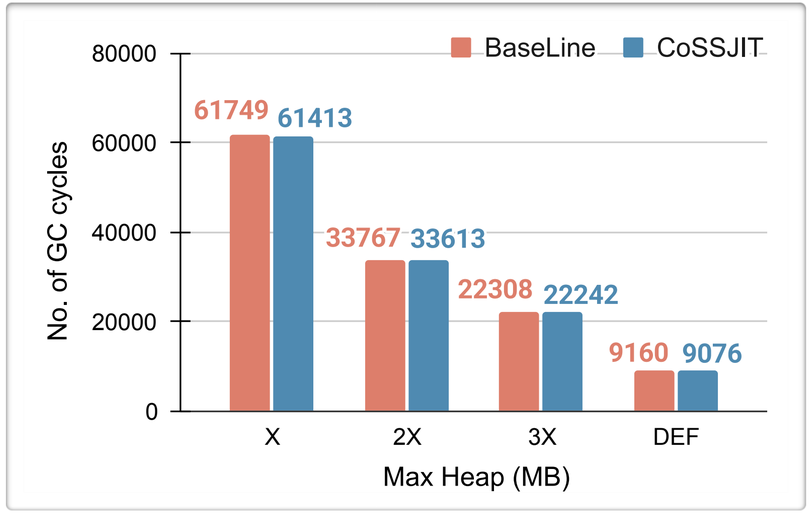
Garbage Collection



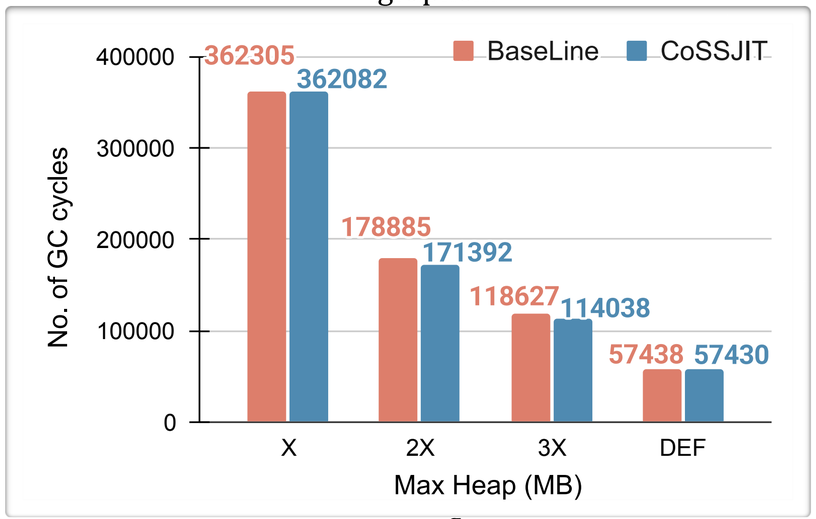
graphchi



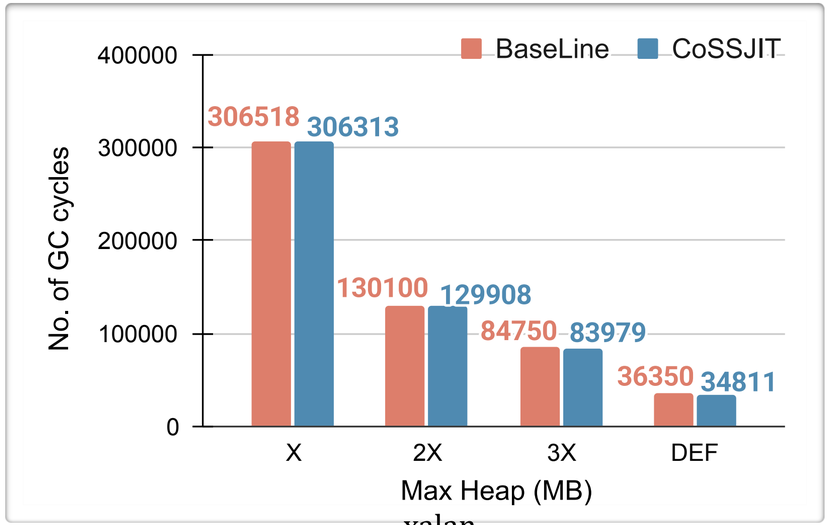
luindex



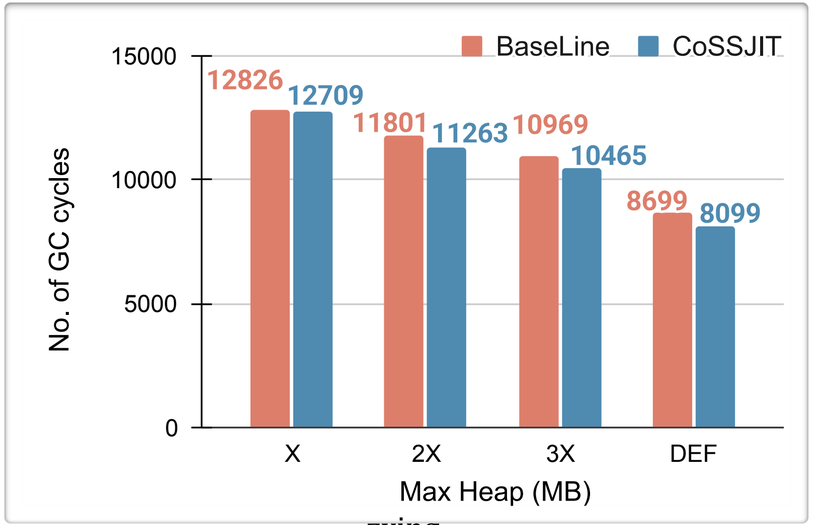
lusearch



sunflow

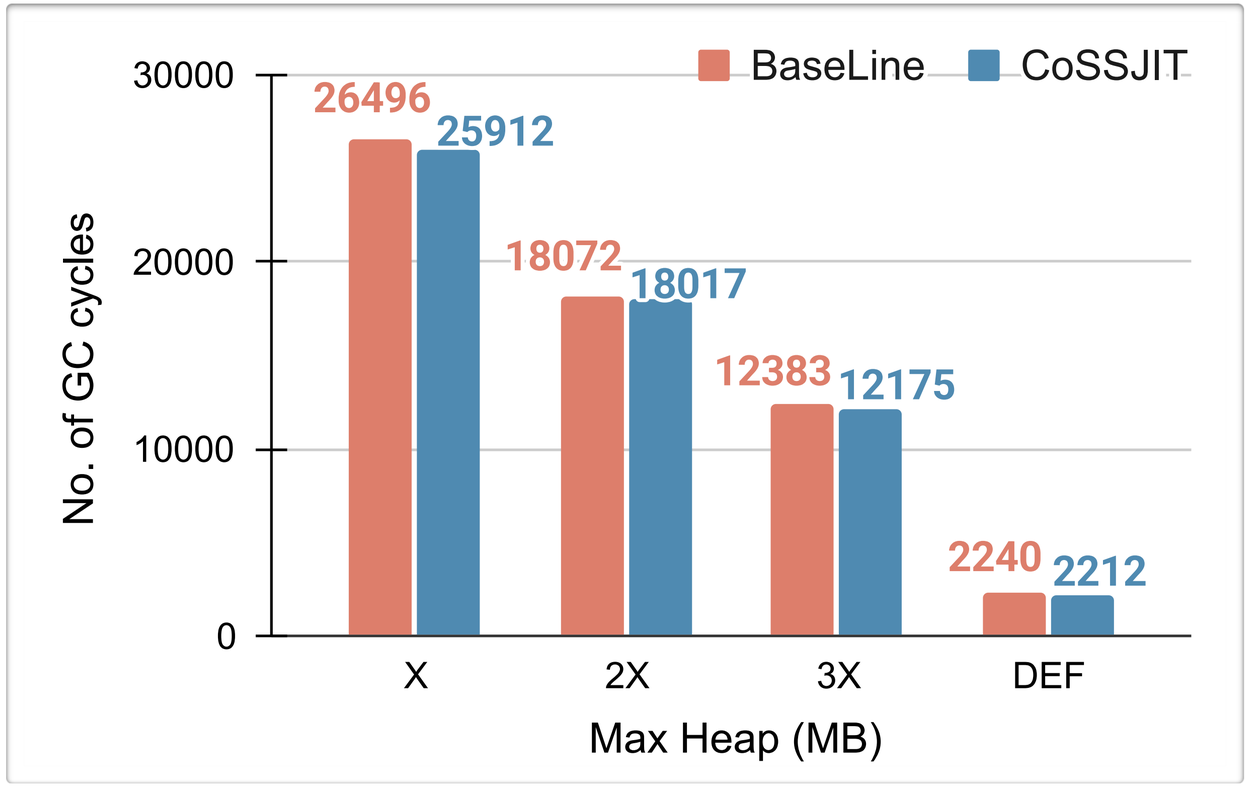


xalan



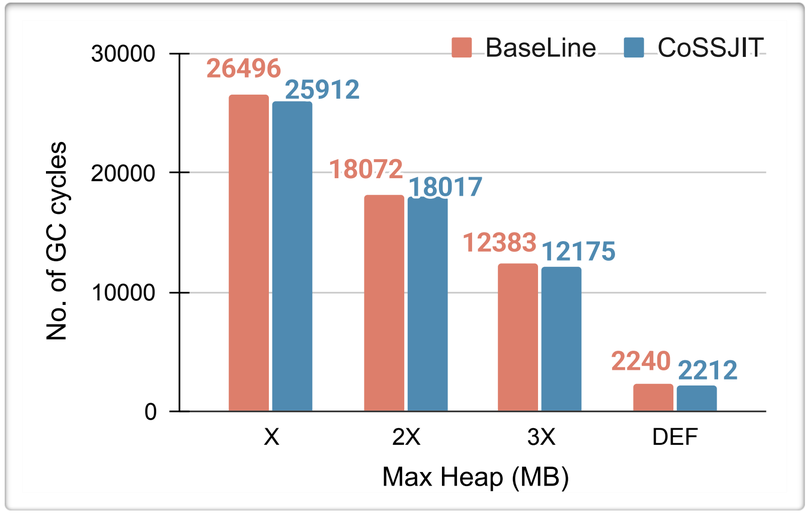
zxing

Garbage Collection

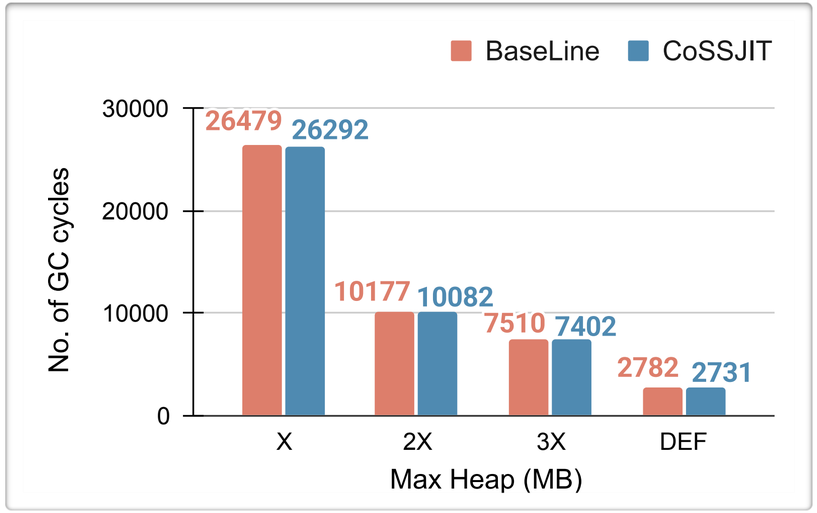


graphchi

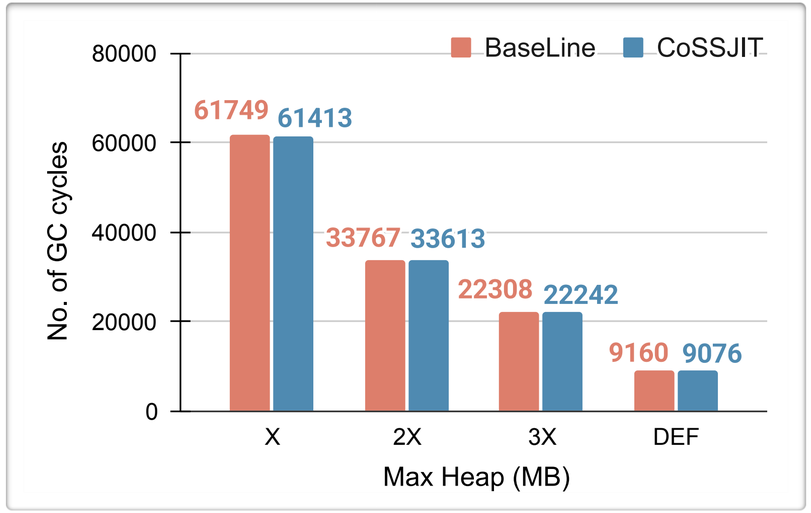
Garbage Collection



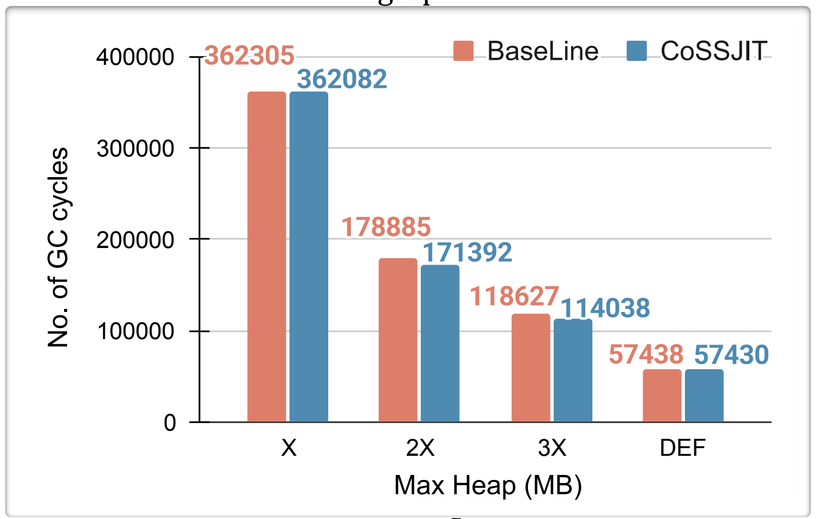
graphchi



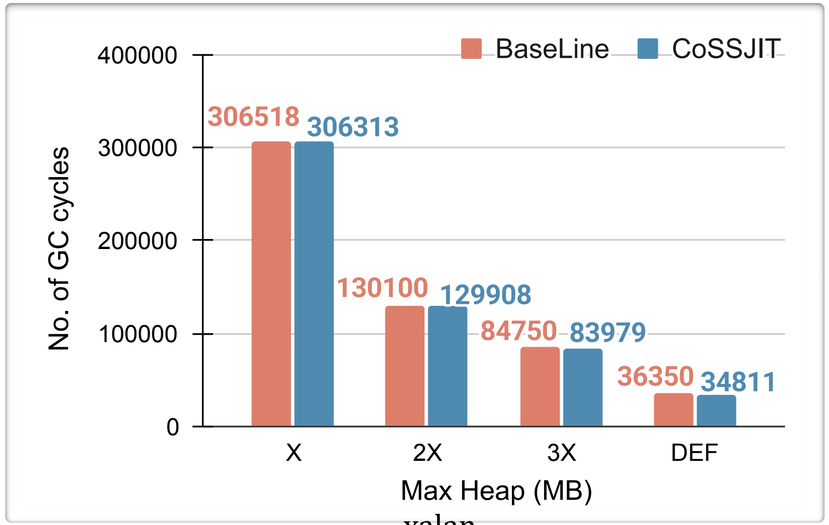
luindex



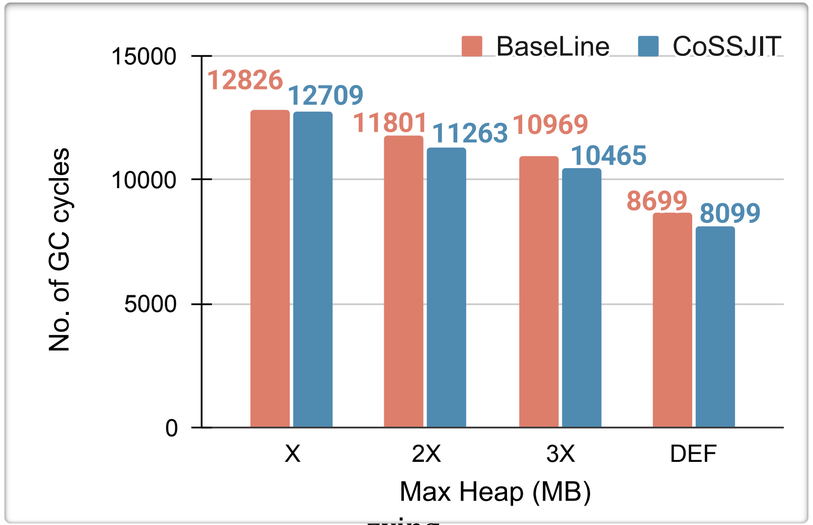
lusearch



sunflow

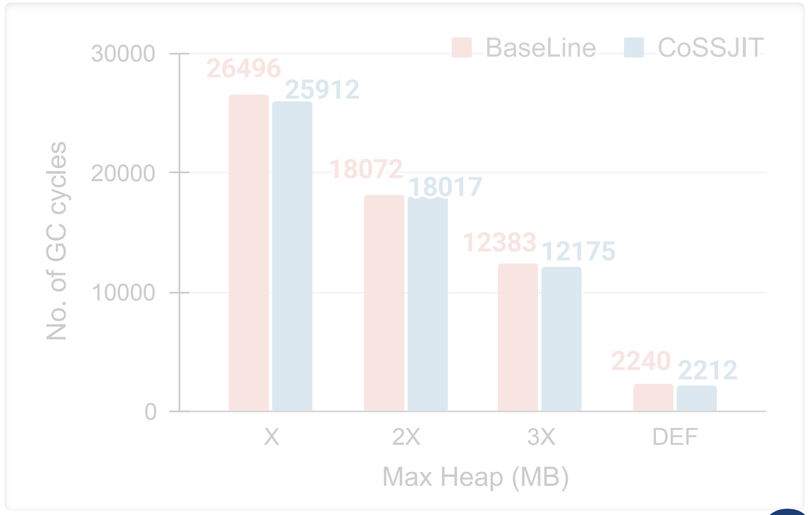


xalan

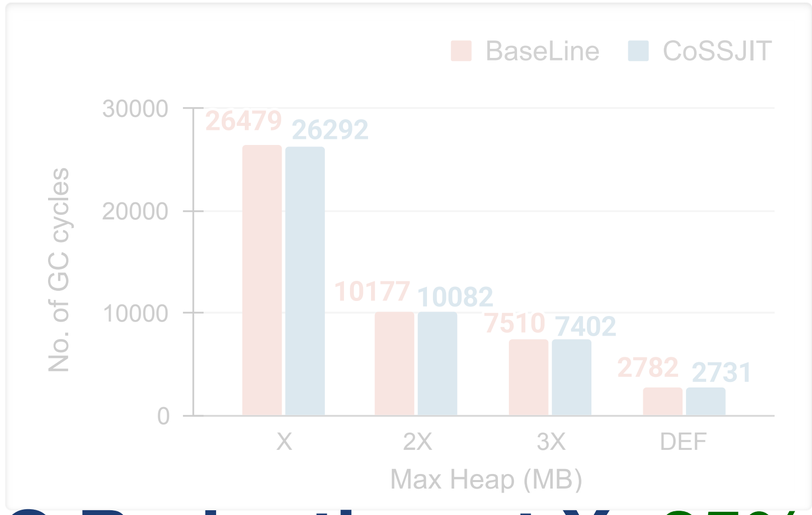


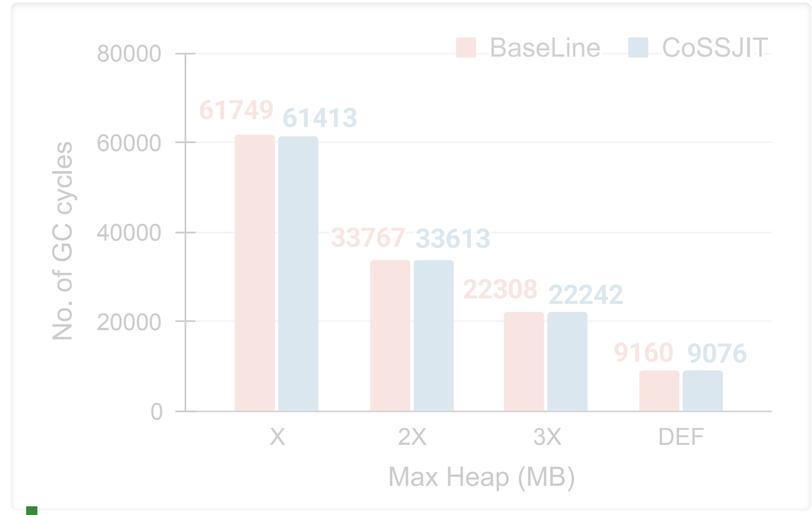
zxing

Garbage Collection



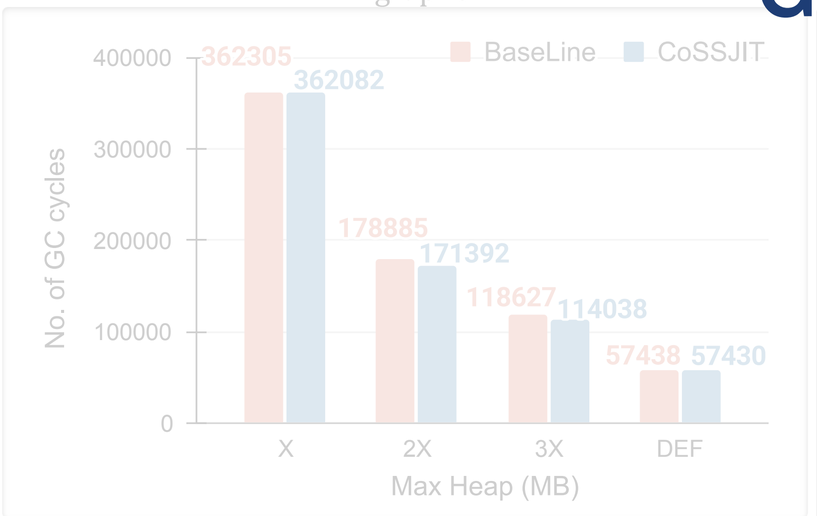
graphchi



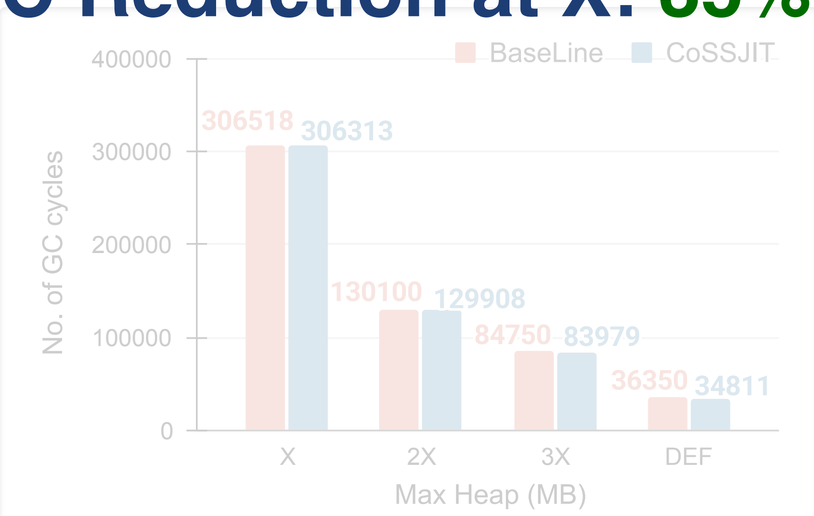


lusearch

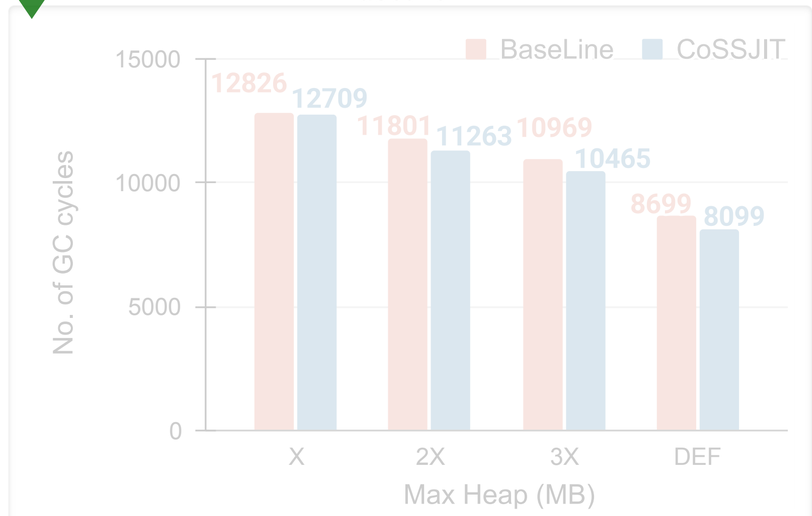
GC Reduction at X: 35% ↓



sunflow

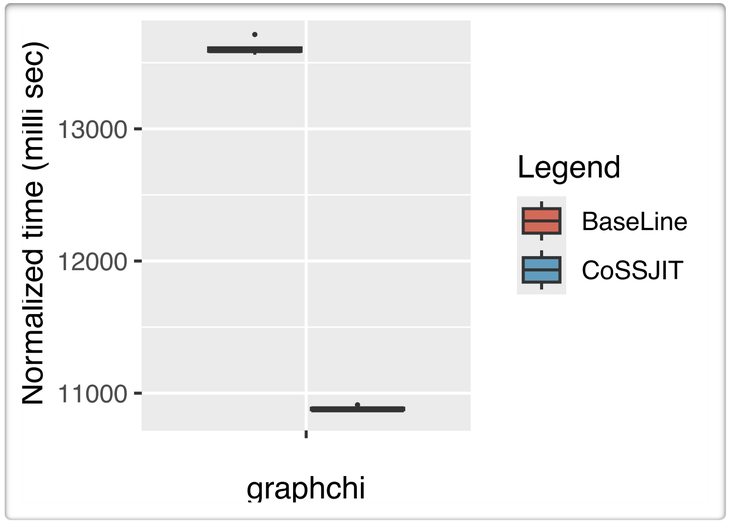


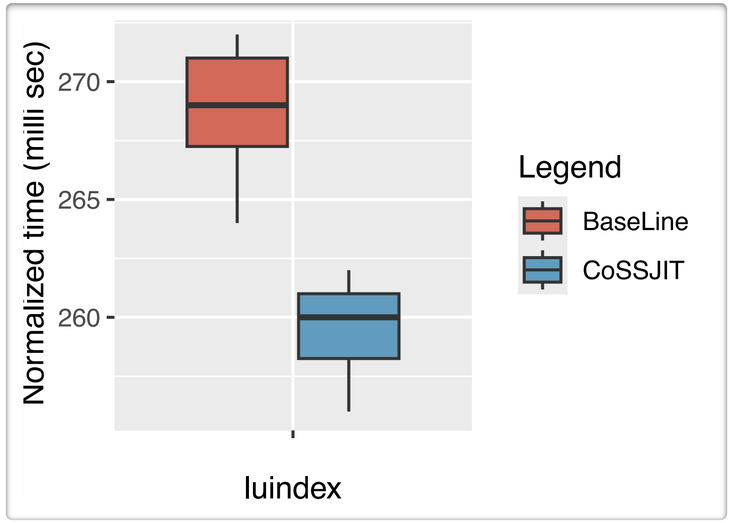
xalan

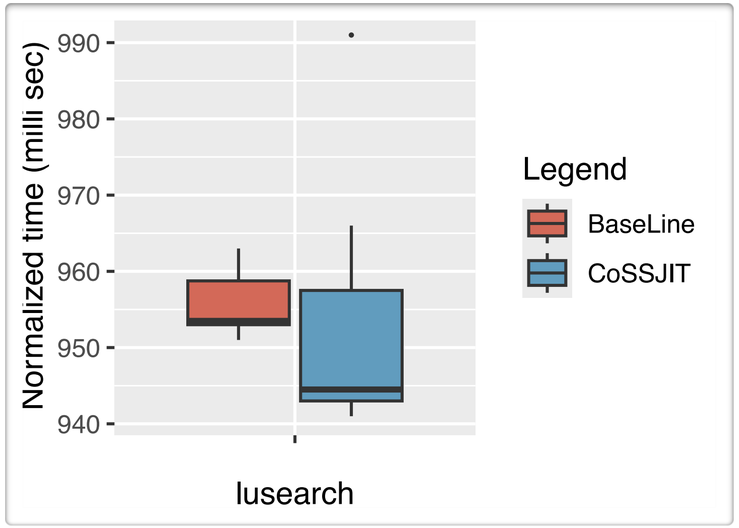


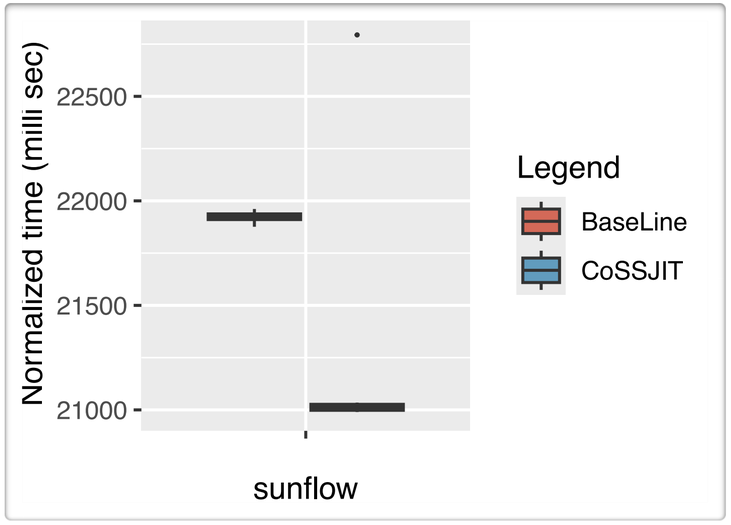
zxing

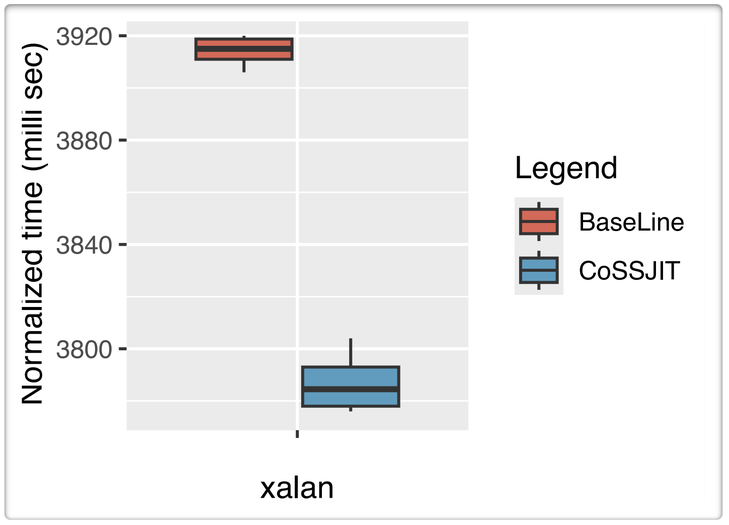
Performance Improvement

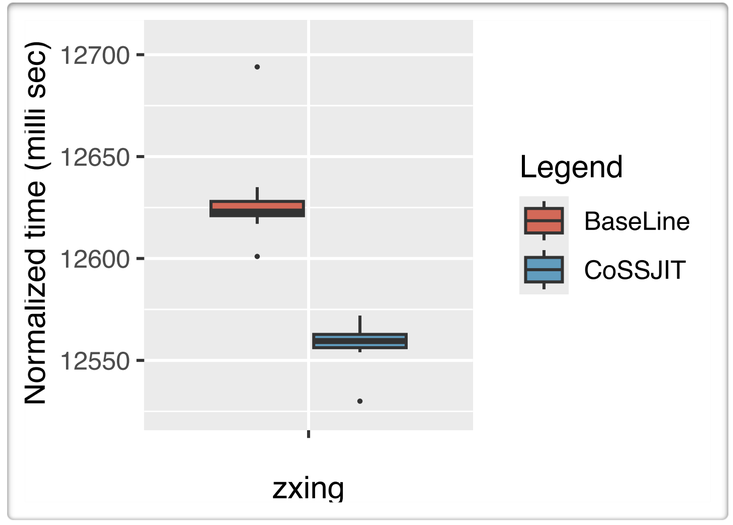




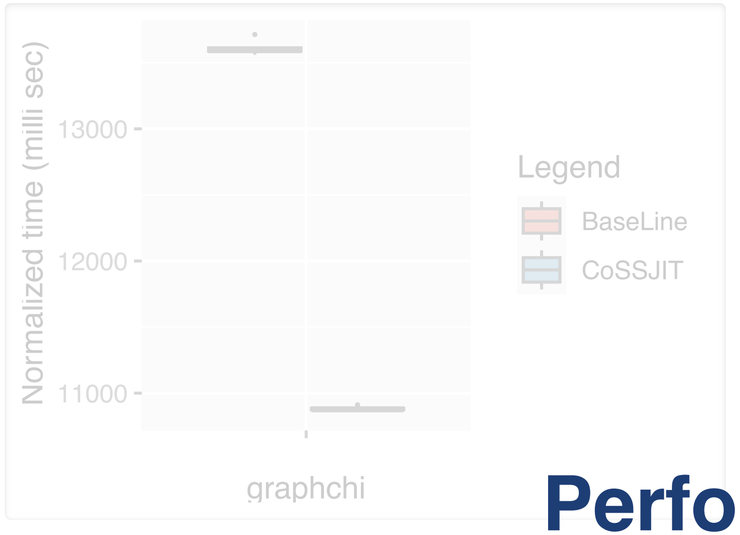


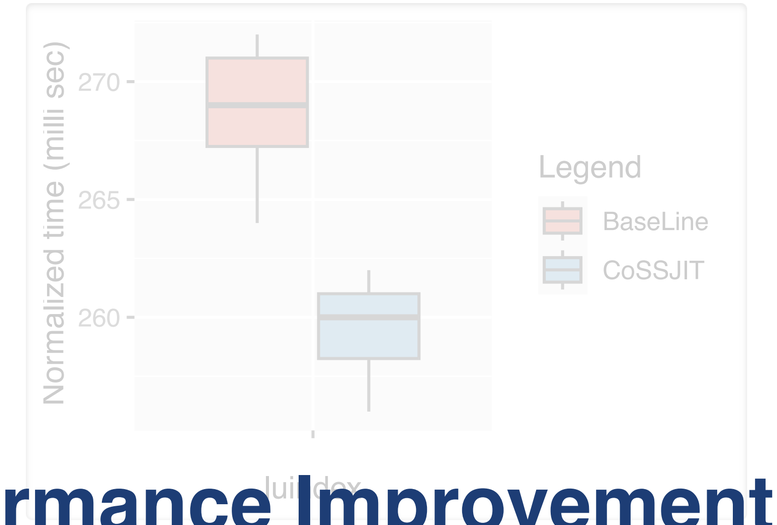


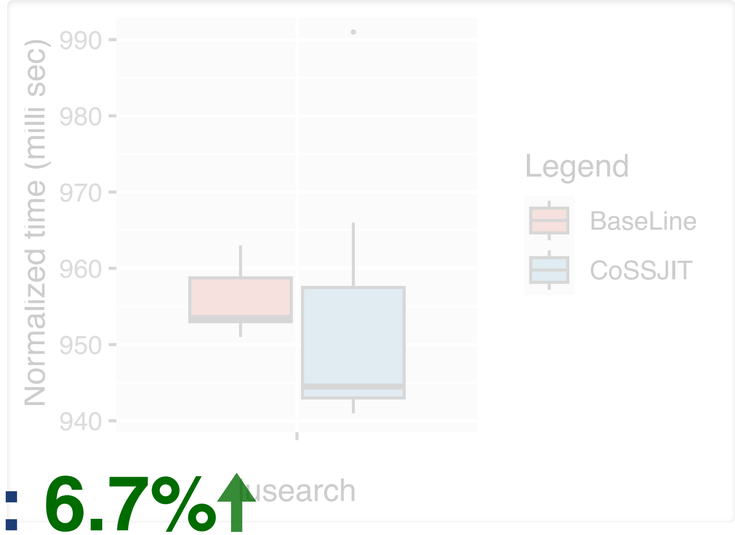




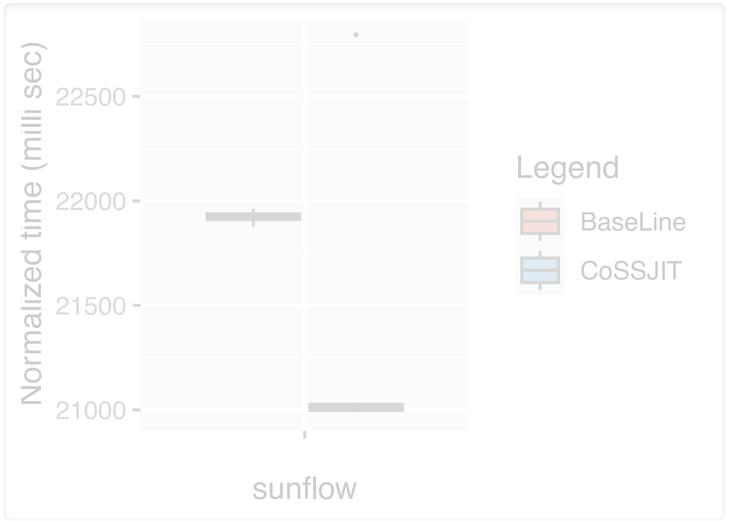
Performance Improvement

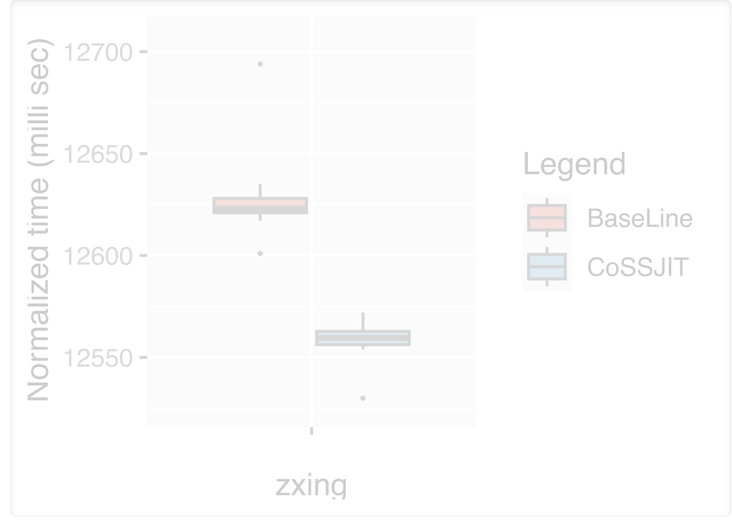




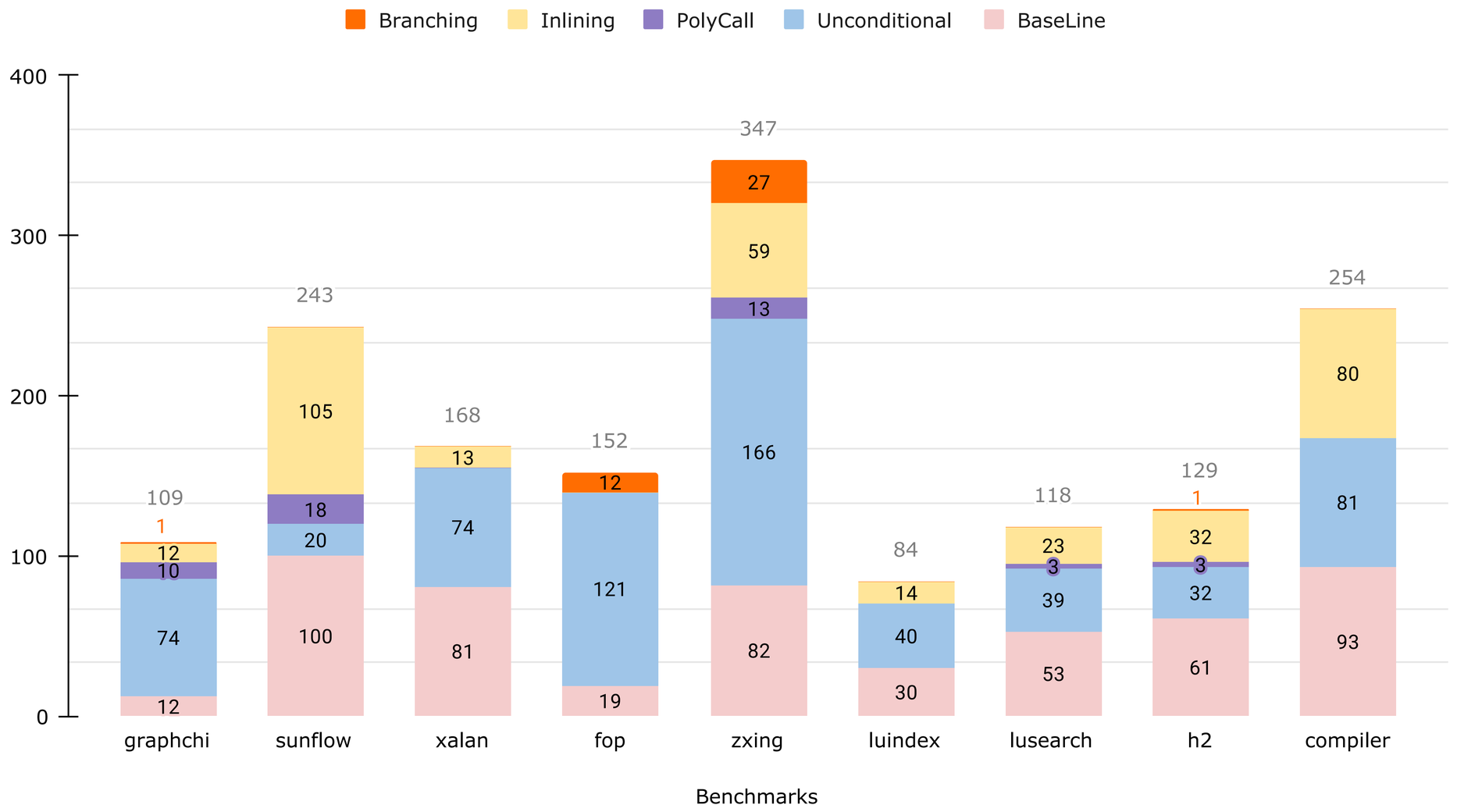


Performance Improvement: 6.7%↑

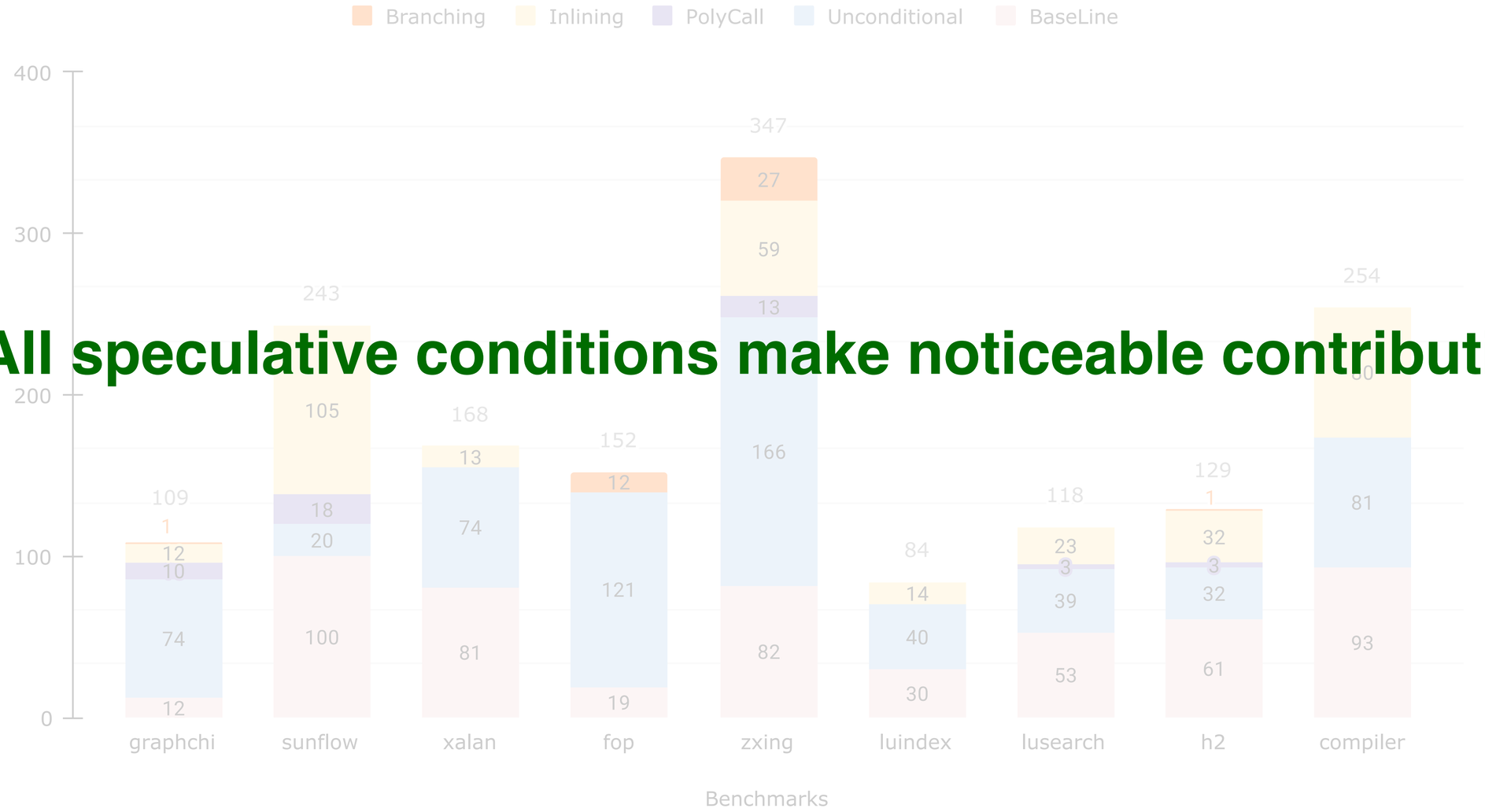




Contributions by different Speculative Conditions



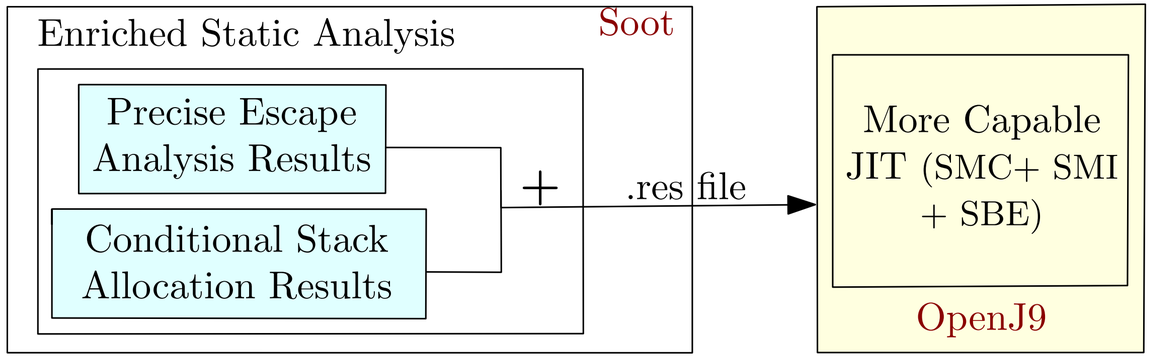
Contributions by different Speculative Conditions



All speculative conditions make noticeable contributions.

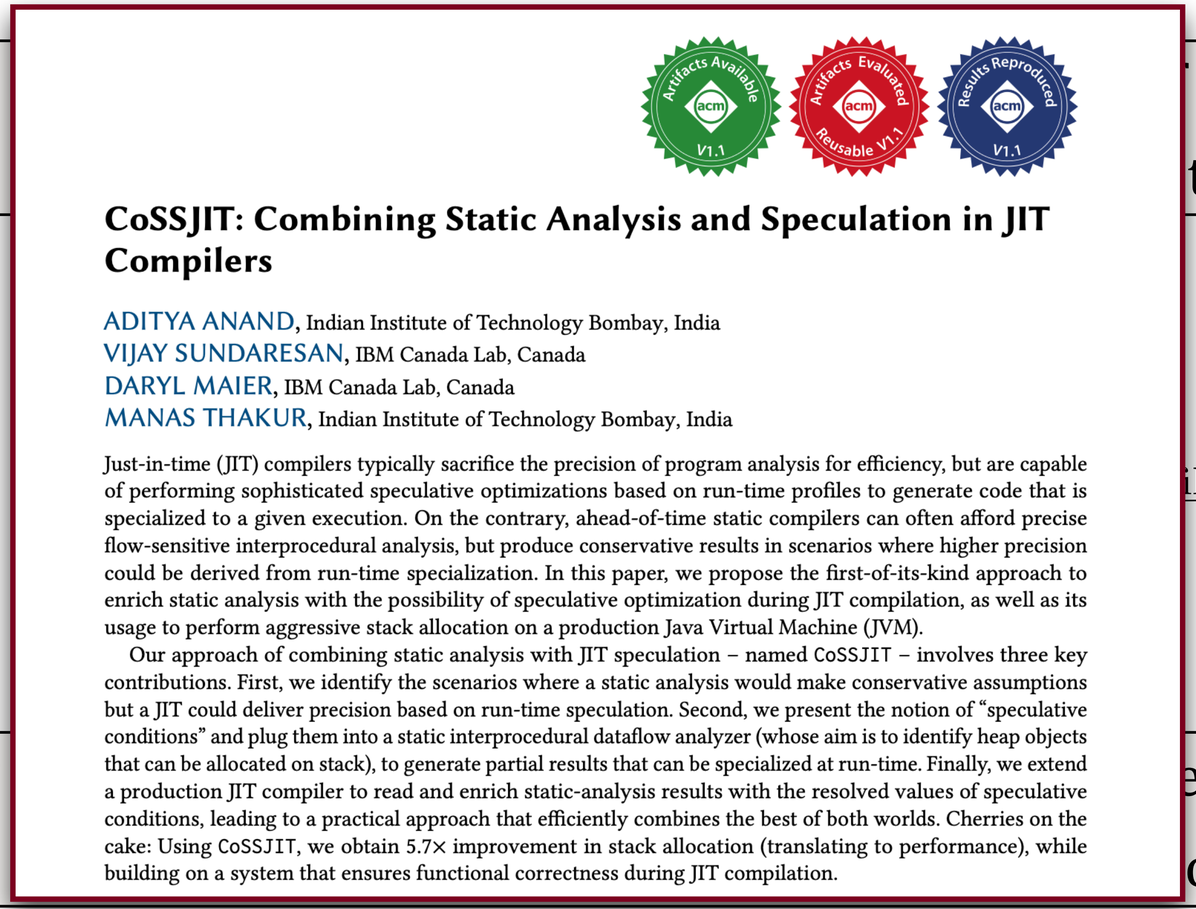
CoSSJIT: Take Aways

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

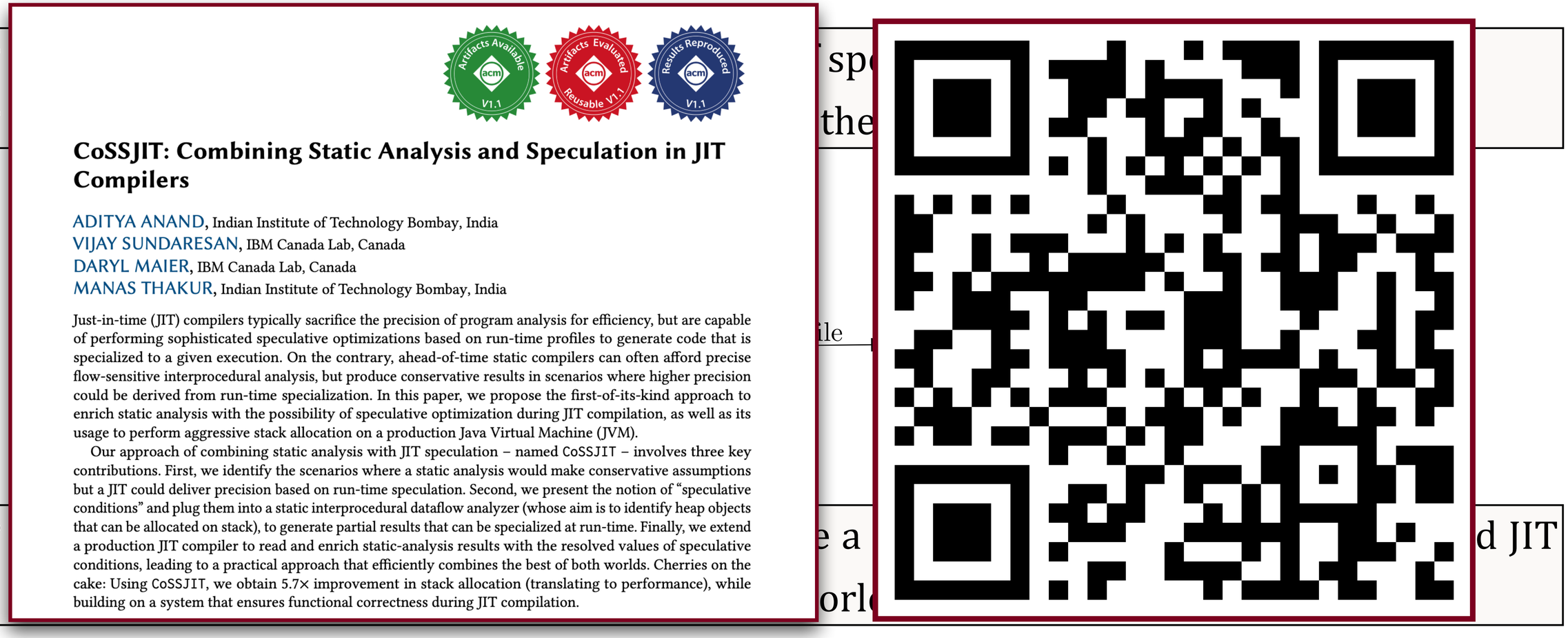


- Overall, one of the first approaches that strike a balance between static analysis and JIT speculation, harnessing the best of both the worlds.

CoSSJIT: Take Aways



CoSSJIT: Take Aways



Thank You !!