Combining Static Analysis and Speculations in JIT Compilers

Innovations in Compiler Technology (IICT-25)

Aditya Anand

Advisor: Prof. Manas Thakur
Indian Institute of Technology Bombay

28th September 2025

Brief Introduction

Brief Introduction

- Program Analysis and JIT (Just-In-Time) compilers for Java like languages.

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.

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.

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.

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

Staged Analysis

- Optimization: Object Allocation

Staged Analysis

- Optimization: Object Allocation
  - By default, objects in Java are allocated on the heap.

Staged Analysis

- Optimization: Object Allocation
  - By default, objects in Java are allocated on the heap.
  - Stack Allocation: Allocate method local objects on the stack frame.

Staged Analysis

- Optimization: Object Allocation
  - By default, objects in Java are allocated on the heap.
  - Stack Allocation: Allocate method local objects on the stack frame.
- Perform precise (context-, flow-, field-sensitive) escape analysis statically.

Staged Analysis

- Optimization: Object Allocation
  - By default, objects in Java are allocated on the heap.
  - Stack Allocation: Allocate method local objects on the stack frame.
- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.

Staged Analysis

- Optimization: Object Allocation
  - By default, objects in Java are allocated on the heap.
  - Stack Allocation: Allocate method local objects on the stack frame.
- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- To handle the functional correctness due to the dynamism offered by the Language/VM — Correctness ensured through run-time checks — Heapification.

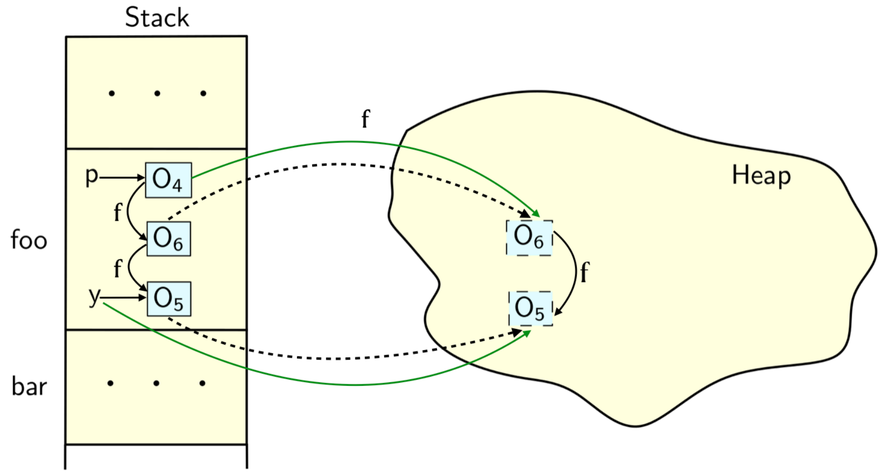
Optimistic Stack Allocation

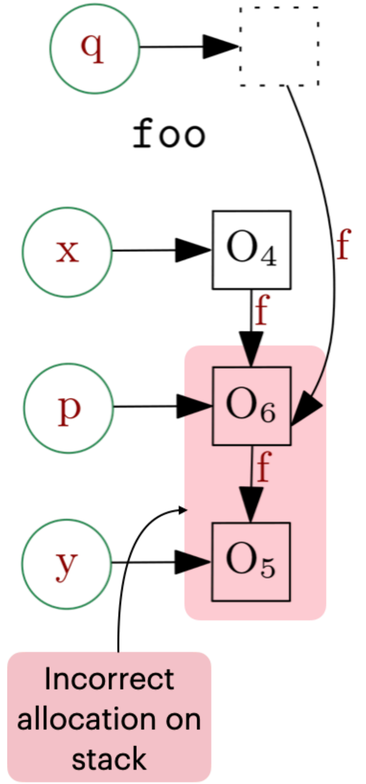
Optimistic Stack Allocation

- Heapification checks whether the static analysis results correspond to the runtime environment, if not:
  - Heapify the object back to the heap from stack.

Optimistic Stack Allocation

- Heapification checks whether the static analysis results correspond to the runtime environment, if not:
  - Heapify the object back to the heap from stack.

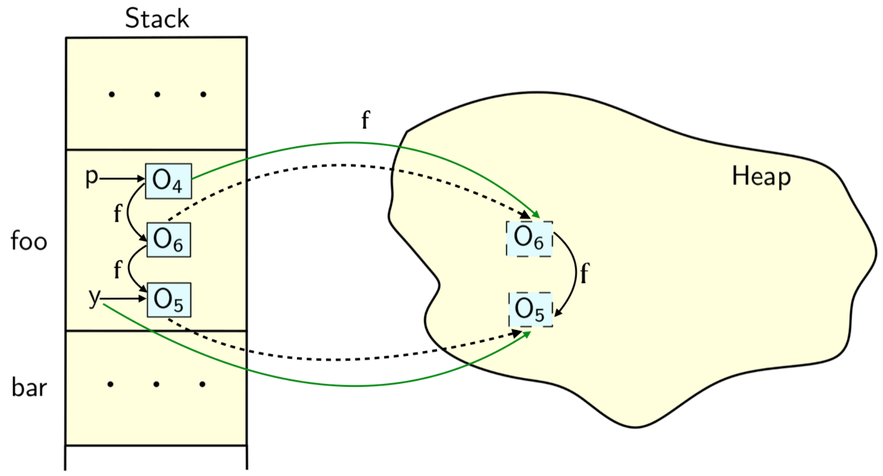


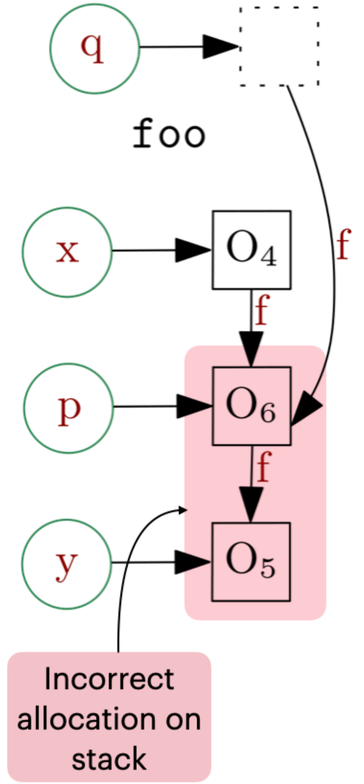


Optimistic Stack Allocation

- Heapification checks whether the static analysis results correspond to the runtime environment, if not:
  - Heapify the object back to the heap from stack.

- Uses the idea of stack ordering to make the checks efficient.

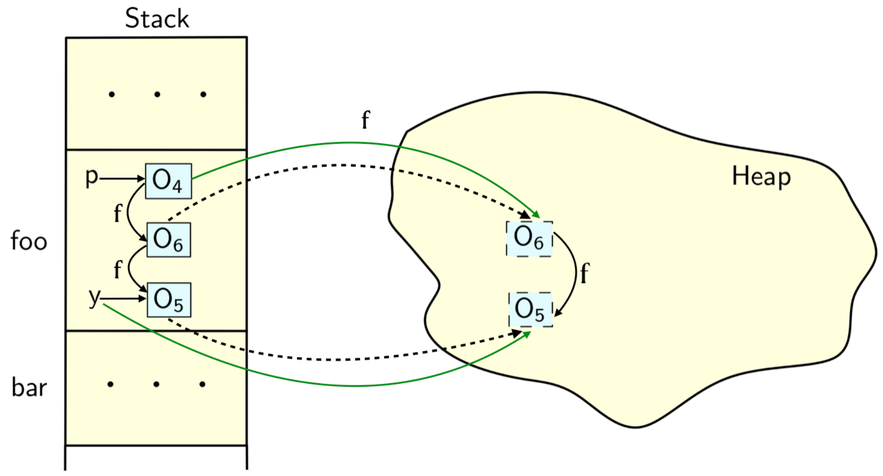


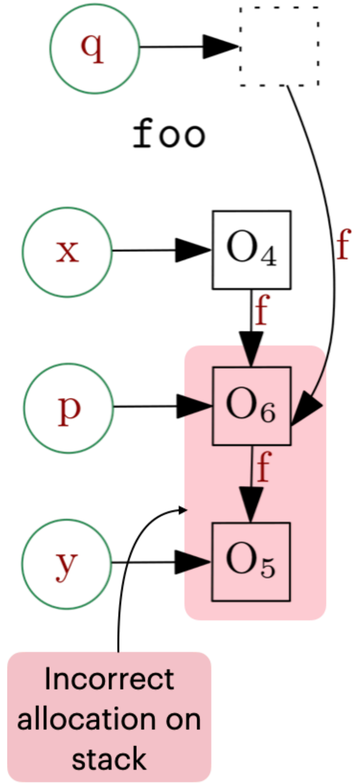


Optimistic Stack Allocation

- Heapification checks whether the static analysis results correspond to the runtime environment, if not:
  - Heapify the object back to the heap from stack.

- Uses the idea of stack ordering to make the checks efficient.





- The first approaches to soundly and efficiently use static (offline) analysis results in a JIT compiler!

Recent Works

Recent Works

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

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

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

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

Recent Works

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

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

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

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

Static Analysis for JIT Compilers

Profiling in JIT Compilers

Profiling in JIT Compilers

- Profile Information:

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.

Profiling in JIT Compilers

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

Profiling in JIT Compilers

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

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.
- Inlining table:

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.
- Inlining table:
  - Maintains a table — for each callsite a list of methods that got inlined during JIT.

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.
- Inlining table:
  - Maintains a table — for each callsite a list of methods that got inlined during JIT.

Standard Speculative Optimizations

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.
- Inlining table:
  - Maintains a table — for each callsite a list of methods that got inlined during JIT.

Standard Speculative Optimizations

Profiling in JIT Compilers

- Profile Information:
  - Basic invocation, loop iteration counts.
  - Type profiles at type-cast statements and polymorphic callsites.
  - Branch information, instance of checks.
- Dynamic Class Hierarchy:
  - Maintains DCT — loaded subclasses of a given class at any point during execution.
- Inlining table:
  - Maintains a table — for each callsite a list of methods that got inlined during JIT.

Standard Speculative Optimizations

Can we make JIT compiler perform more speculative optimizations ??

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

- Idea:

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

- Idea:
  - Enrich Static Analysis results with possibility of speculation at run-time.

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. 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 */
```

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 */
```

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

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

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

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

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

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 */
```

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 */
```

$\sqcap$: {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 */
```

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 */
```

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:

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.

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 */
```

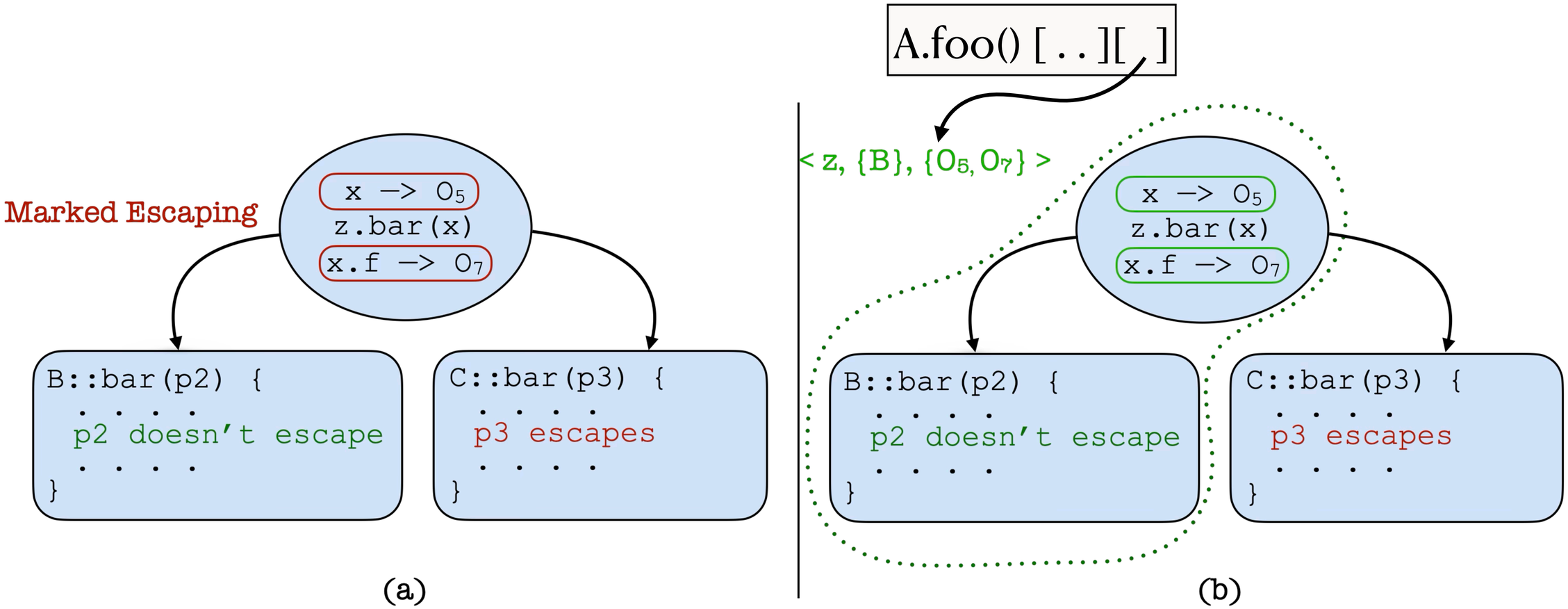
```
17. class B extends A {
18.   void bar(A p2) { . . . }
19. } /* class B */
```

```
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".

1. PolyMorphic CallSites



2. Branching

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 */
```

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 */
```

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 */
```

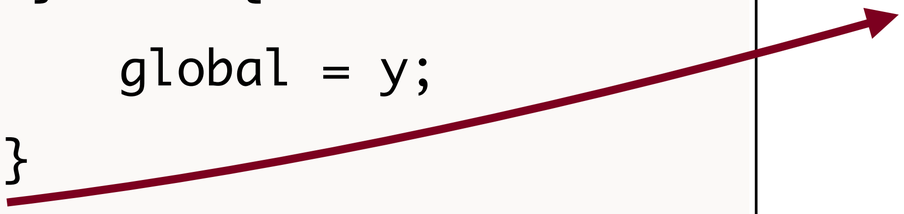
⊓ : —> O_6 (Escaping)

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 */
```

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 */
```



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:

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.

2. Branching

```
A::foo(p1) {
    . . . .
6. Y = new A(); // O6
9. if(*) {
     . . .
   } else {
     // Object O6 escapes.
   }
 }
```

O_6 Marked Escaping

(a)

```
A::foo(p1) {
    . . . .
6. Y = new A(); // O6
9. if(*) {
     . . .
   } else {
     // Object O6 escapes.
   }
 }
```

[< {9}, {O_6} >]

A.foo() [. .][ ]

(b)

3. Inlining Based Results

3. Inlining Based Results

```
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 */
```

3. Inlining Based Results

```
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 { . . . }
```

3. Inlining Based Results

```
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 { . . . }
```

3. Inlining Based Results

```
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 { . . . }
```

3. Inlining Based Results

```
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. Inlining Based Results

```
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 { . . . }
```

3. Inlining Based Results

```
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. Inlining Based Results

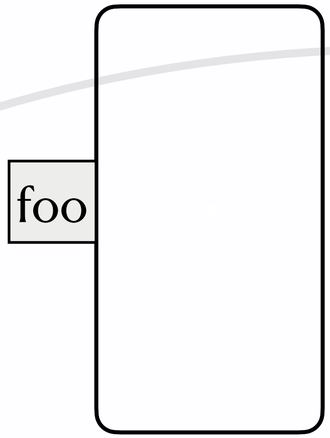
```
1. class A {
2.   void foo() {
3.     . . .
4.     z.bar(x);
5.     r.foobar(p, q);
6.   } /* method foo */
7.   void bar(A p1) { . . . }
8.   void foobar(A p2) { . . . }
9. } /* class A */
```

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. Inlining Based Results

```
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 */
```

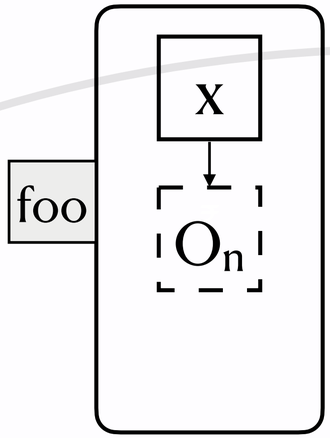


Inline

```
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. Inlining Based Results

```
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 */
```

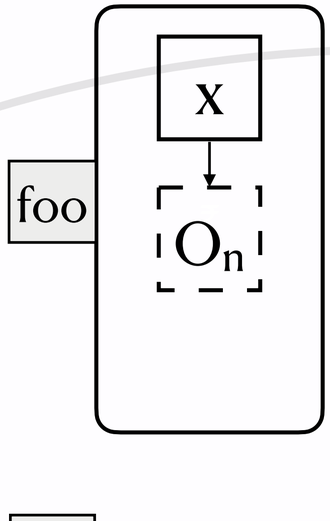


Inline

```
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. Inlining Based Results

```
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 */
```



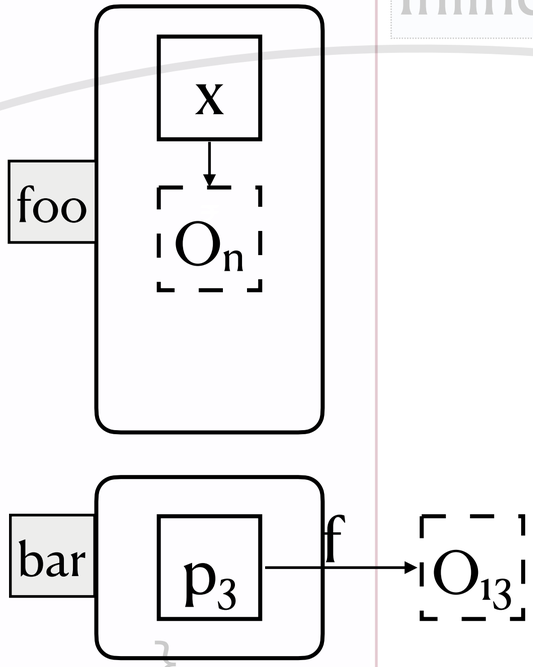
Inline

```
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. Inlining Based Results

```
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 */
```

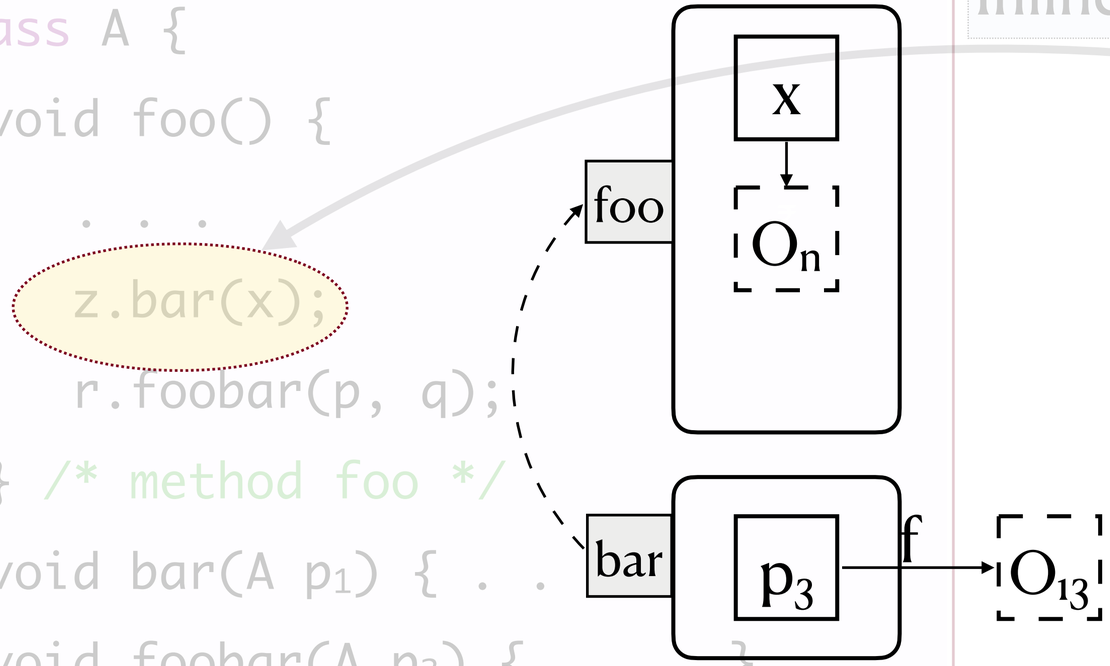
Inline




```
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. Inlining Based Results

```
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 */
```

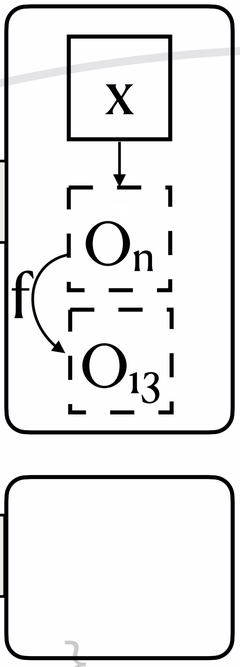


Inline

```
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. Inlining Based Results

```
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 */
```

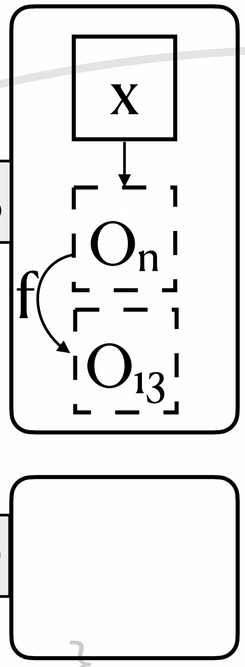


Inline

```
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. Inlining Based Results

```
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 */
```

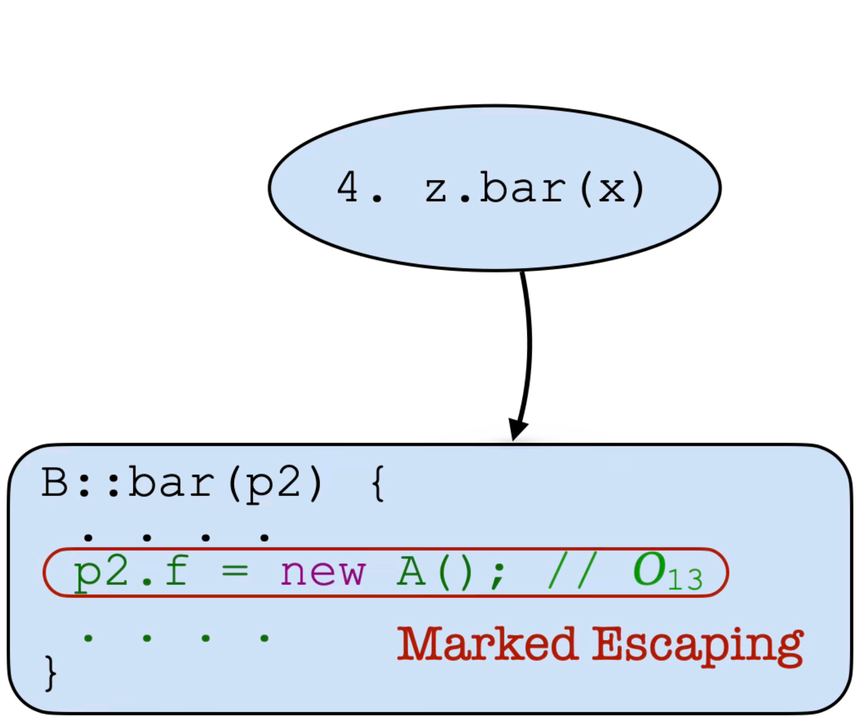


Inline

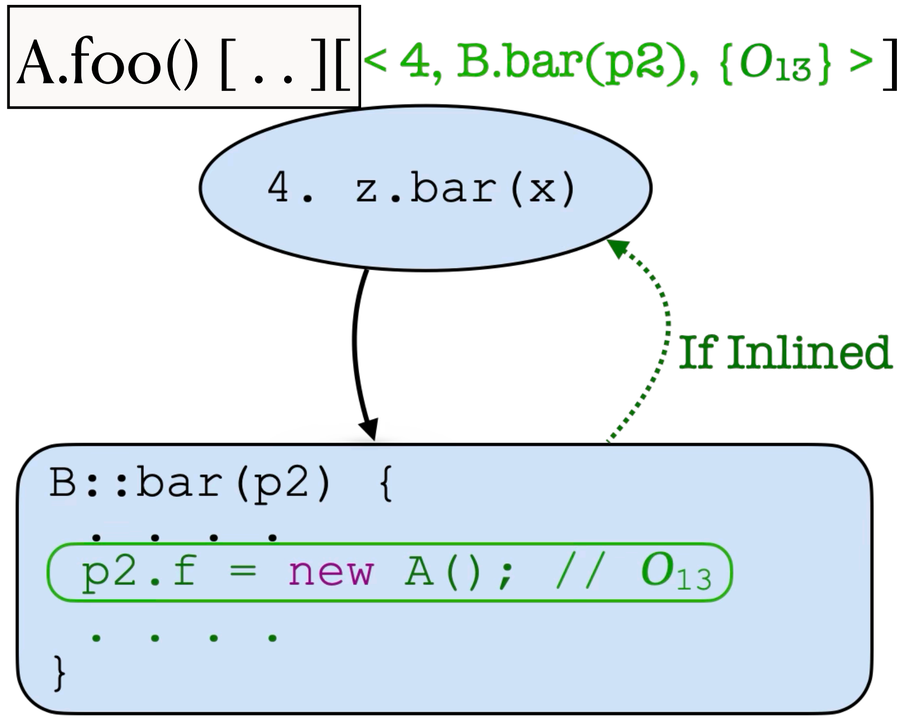
```
10. class B extends A {
11.   void bar(A p3) {
12.     // p3's pointee doesn't escape
13.     p3.f = new A(); // O13
14.     p3.foobar(p3.f);
15.   } /* method bar */
16.   void foobar(A p4) { . . . }
17. } /* class B */
18. class C extends A { . . . }
```

Callee object on caller's stack frame

3. Inlining Based Results



(a)



(b)

Summarize

Summarize

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

Summarize

Direct

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

Summarize

Direct

Conditional

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

Summarize

Direct

Conditional

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

- Statically generated results

Summarize

Direct

Conditional

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

- Statically generated results

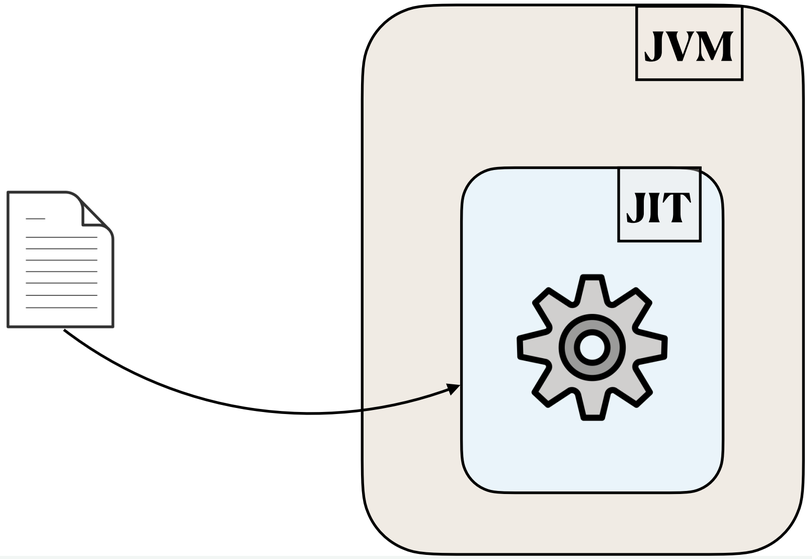
Summarize

Direct

Conditional

- A::foo() [Direct_Allocation] [bci_polymorphic_callsite] [bci_branching][bci_inlining]

- Statically generated results



Algo in JIT for Polymorphic Callsites Results

```
Input: SAm, CHm, CPm, ST
Output: Objects marked for stack allocation

for each O in JIT_identified_objects(m):
    if O is non-escaping:
        mark O
    else if O in SAm:
        if ∀ callsite c ∈ SAm[O]:
            (CHm[c] ⊆ SAm[O][c]) v ( ∀ t ∈ SAm[O][c] ∑ CPm(t) > ST):
            mark O
```

Algo in JIT for Branching Results

```
Input: SAm, BPm, ST
Output: Additional objects marked for stack allocation

for each O in Candidatesm not marked:
    if ∀ branch br ∈ SAm[O]:
         (∀ b ∈ SAm[O][br] with Σ BPm(b) > ST):
        mark O
```

Algo in JIT for Inlining Based Results

```
Input: SAm, ITm
Output: Additional objects marked for stack allocation

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)

Evaluation (Setup)

- Implementation:
  - Static analysis: **Soot**
  - Runtime components: **OpenJ9 VM**

Evaluation (Setup)

- Implementation:
  - Static analysis: **Soot**
  - Runtime components: **OpenJ9 VM**

- Benchmarks:
  - **DaCapo suites** 23.10-chopin and 9.12 MRI.
  - **SPECjvm** 2008.

Evaluation (Setup)

- Implementation:
  - Static analysis: **Soot**
  - Runtime components: **OpenJ9 VM**

- Benchmarks:
  - **DaCapo suites** 23.10-chopin and 9.12 MRI.
  - **SPECjvm** 2008.

- Evaluation schemes:
  - **BaseLine**: Stack allocation with the existing scheme.
  - **CoSSJIT**: Stack allocation with our scheme.

Evaluation (Setup)

- Implementation:
  - Static analysis: **Soot**
  - Runtime components: **OpenJ9 VM**

- Benchmarks:
  - **DaCapo suites** 23.10-chopin and 9.12 MRI.
  - **SPECjvm** 2008.

- Evaluation schemes:
  - **BaseLine**: Stack allocation with the existing scheme.
  - **CoSSJIT**: Stack allocation with our 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	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	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	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	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

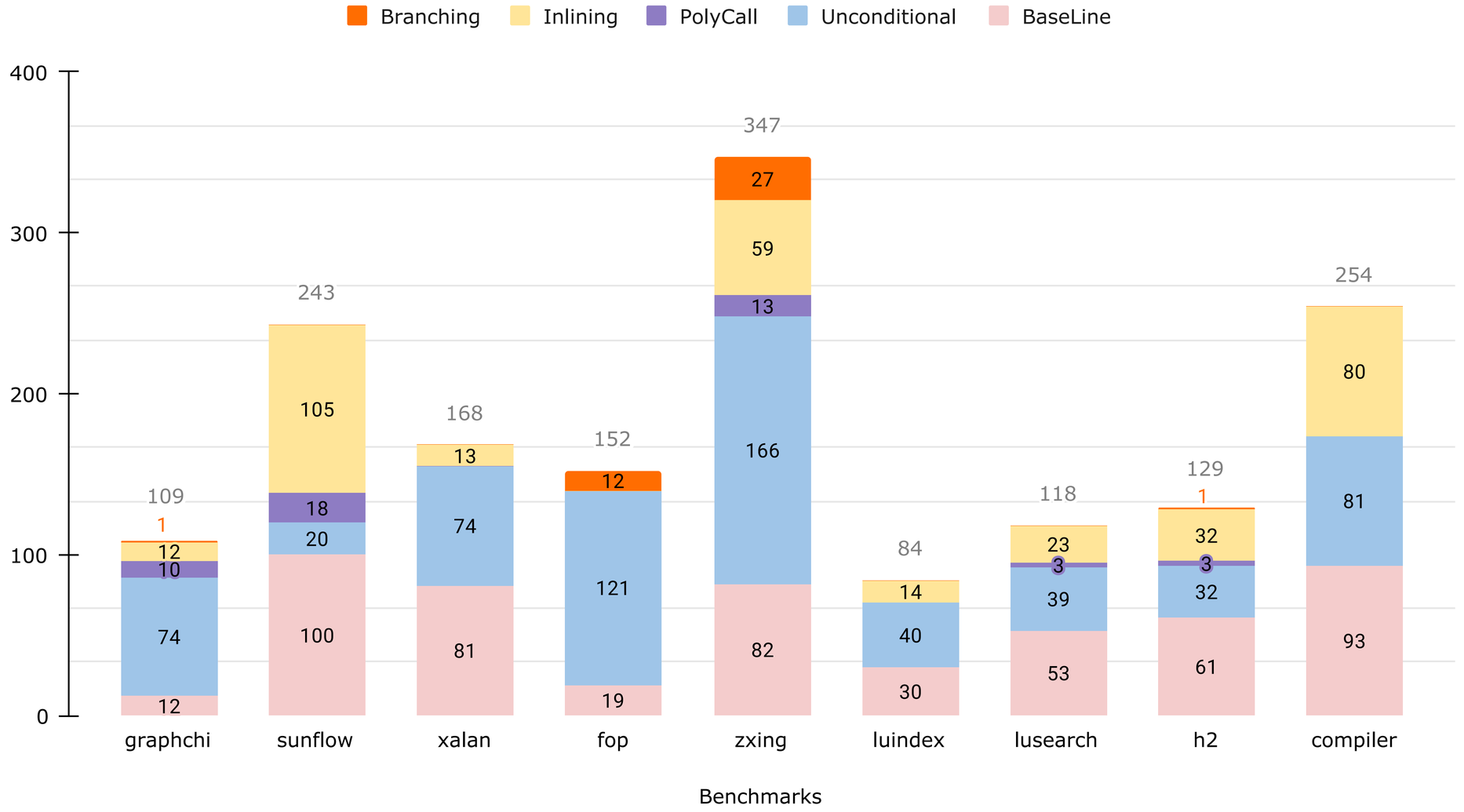
Evaluation (Stack Allocation)

	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 (1.20%)	8327MB	10	1M (14.2%)	20820MB	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

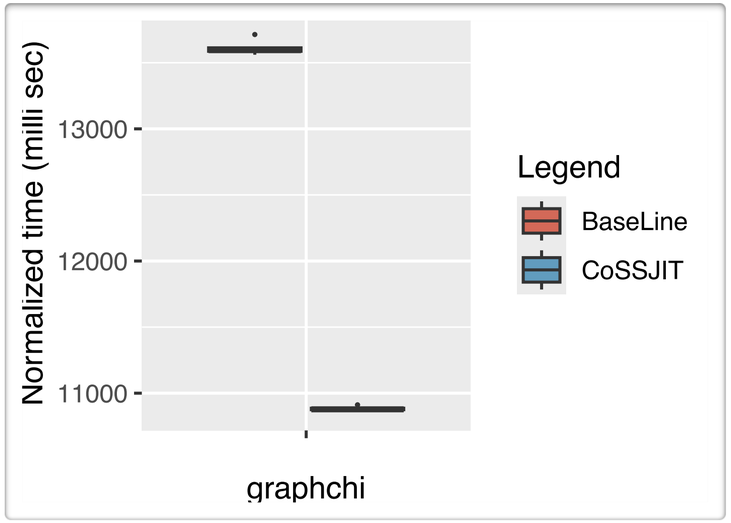
Stack Allocation: 1.4x↑ Stack Bytes: 5.7x↑

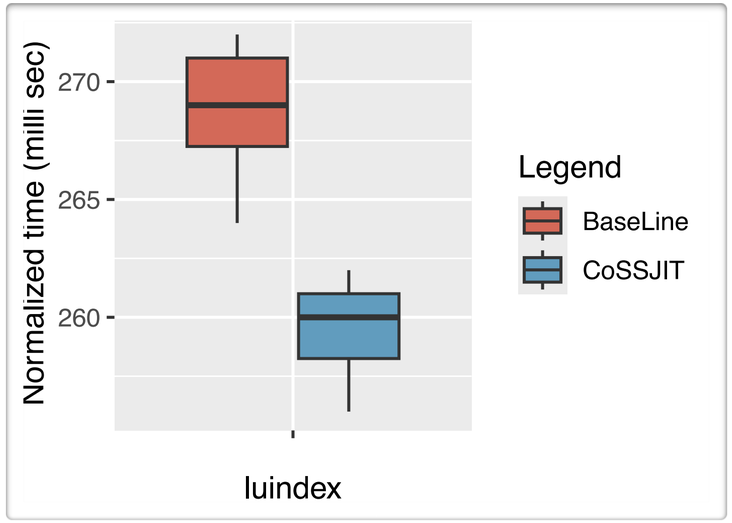
(Less Heap Allocation)

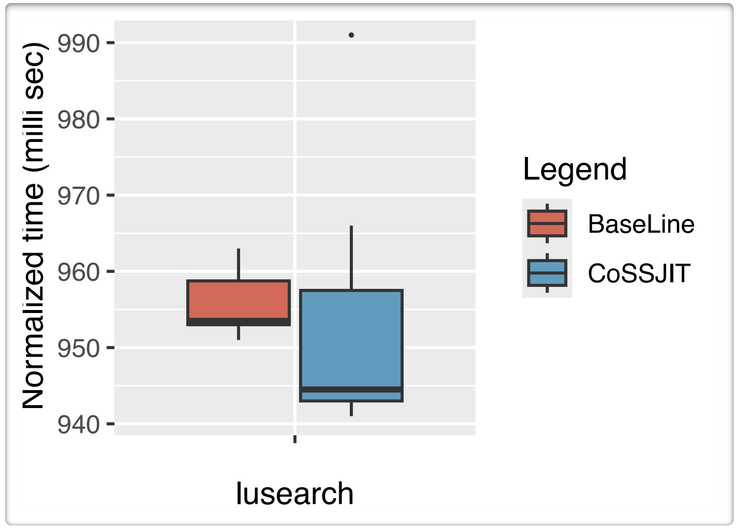
Contributions by different SC

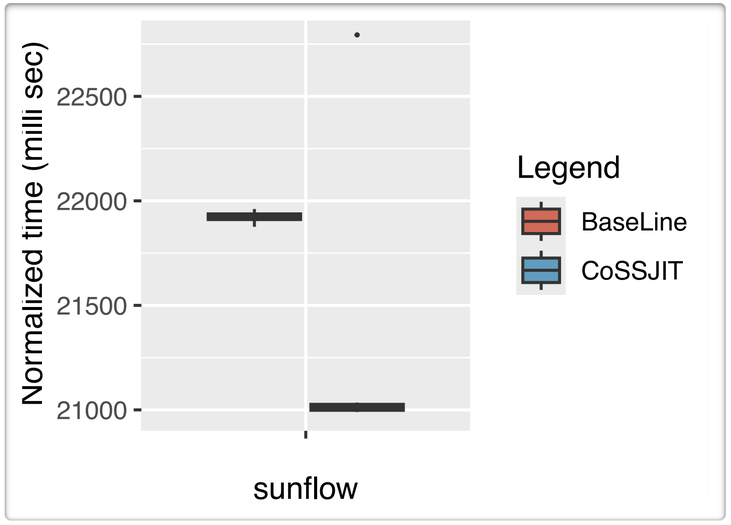


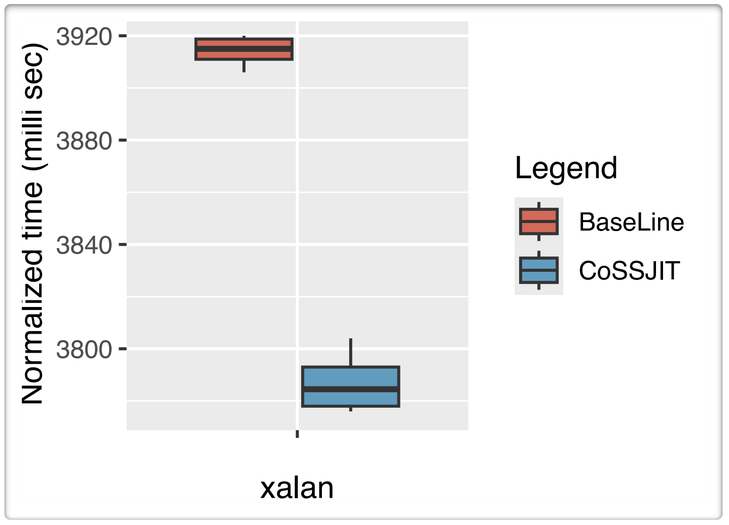
Performance Improvement

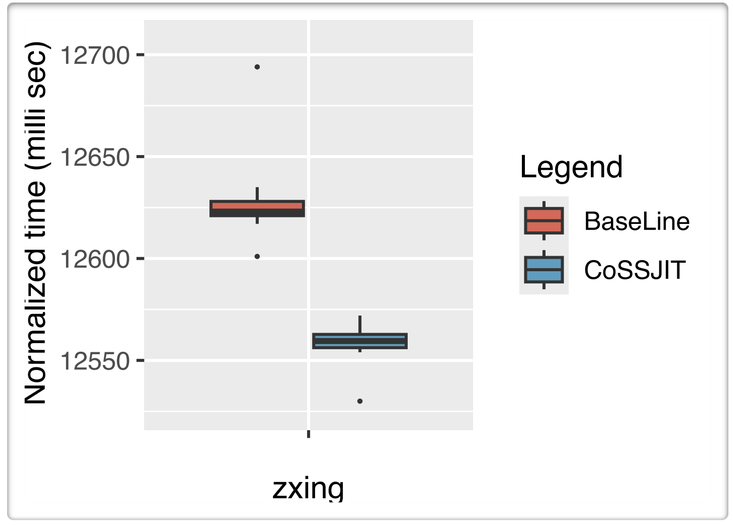




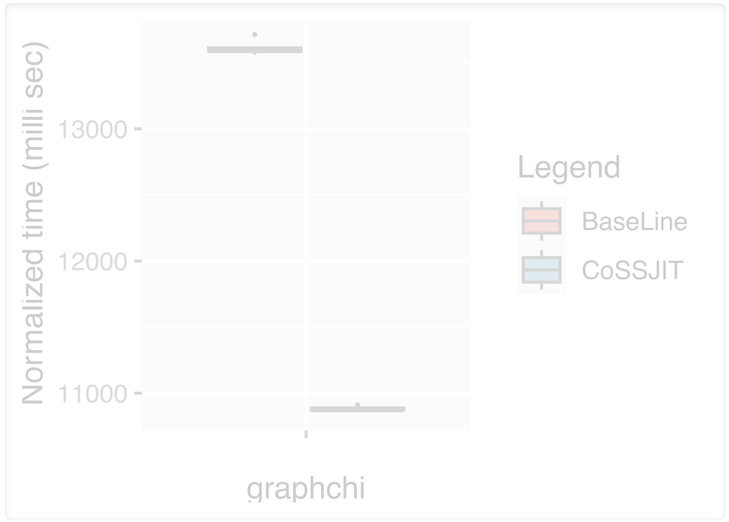


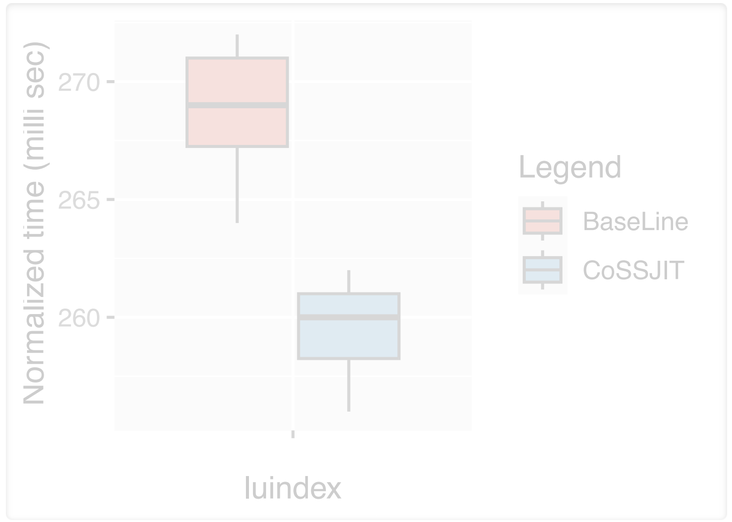


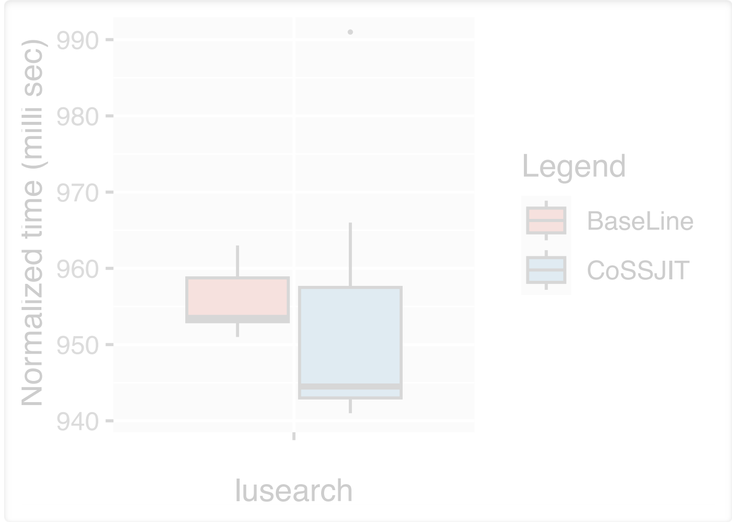


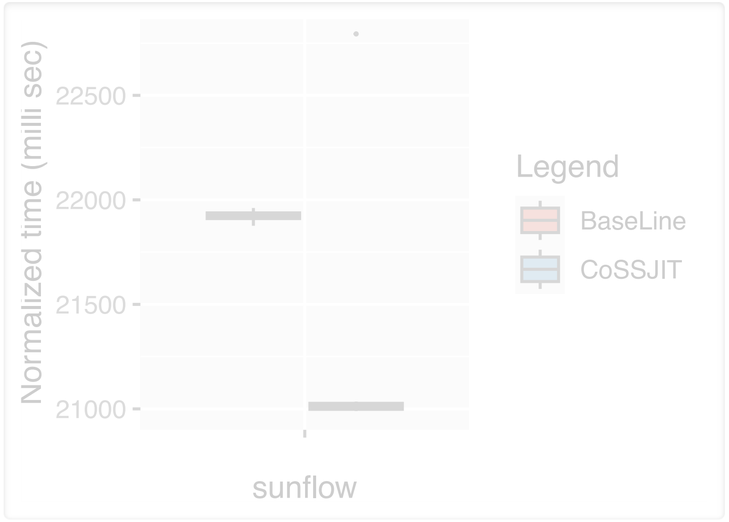


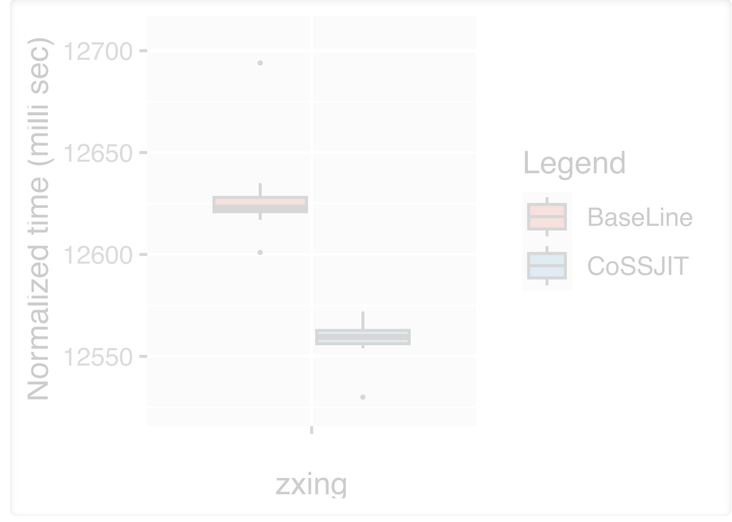
Performance Improvement



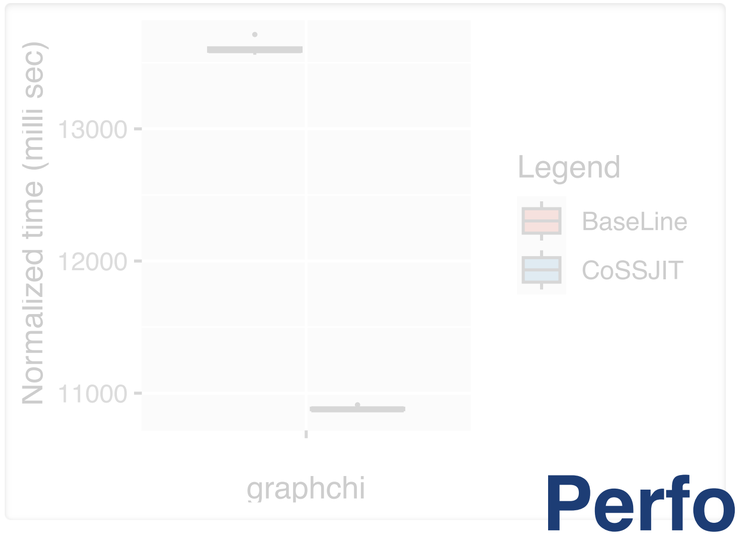


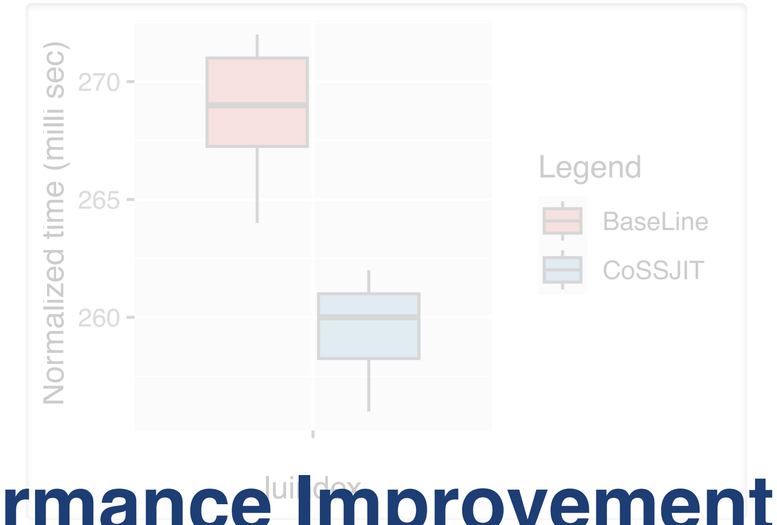


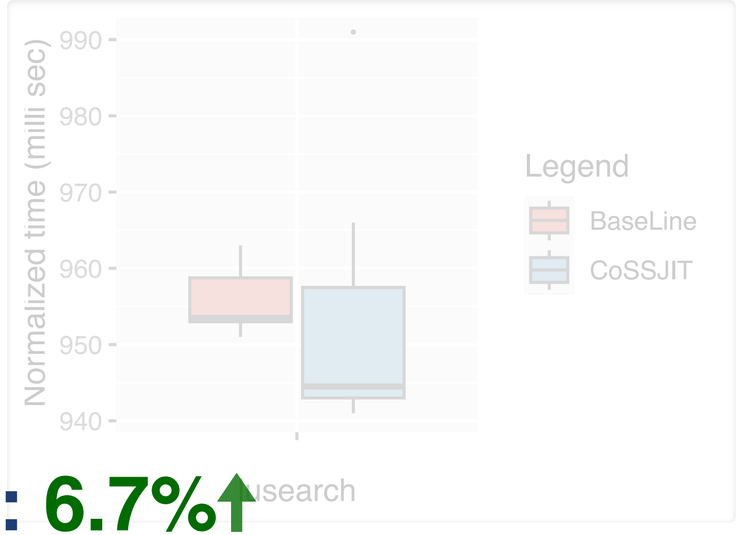




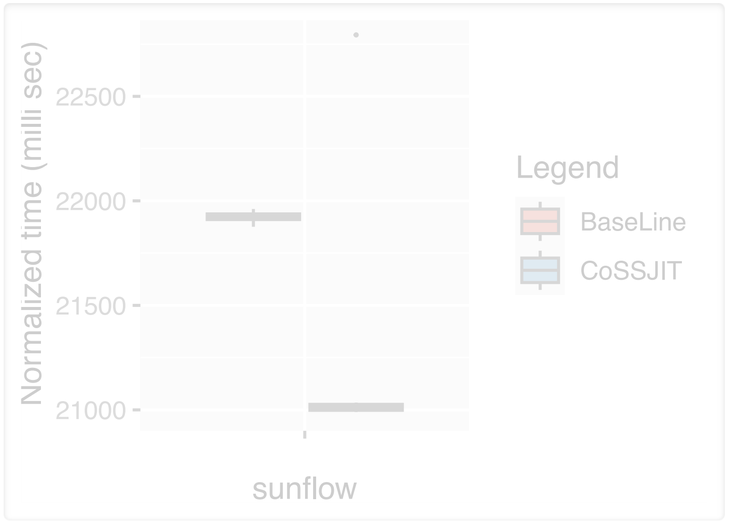
Performance Improvement

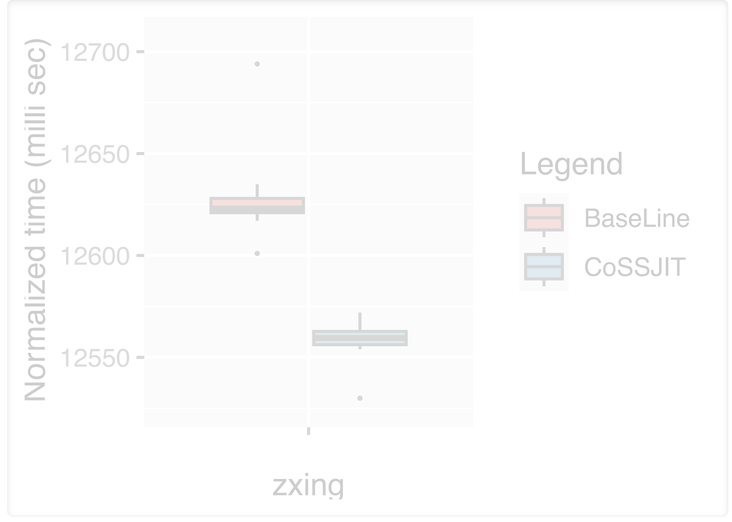




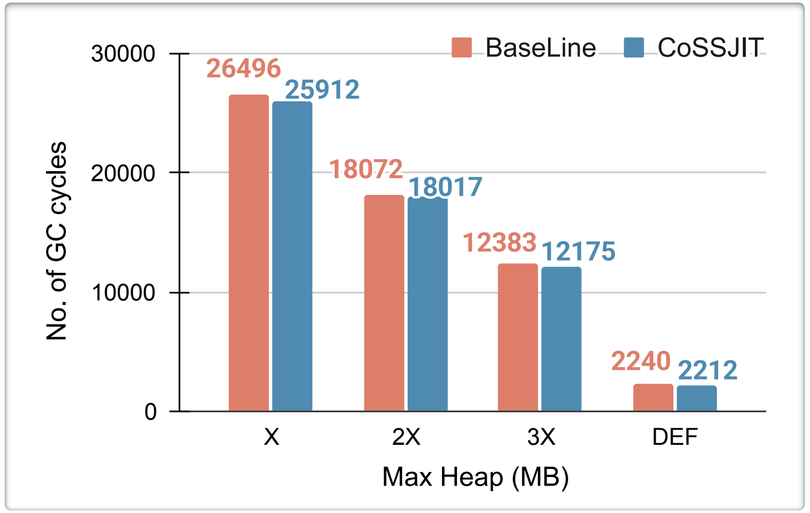


Performance Improvement: 6.7%↑

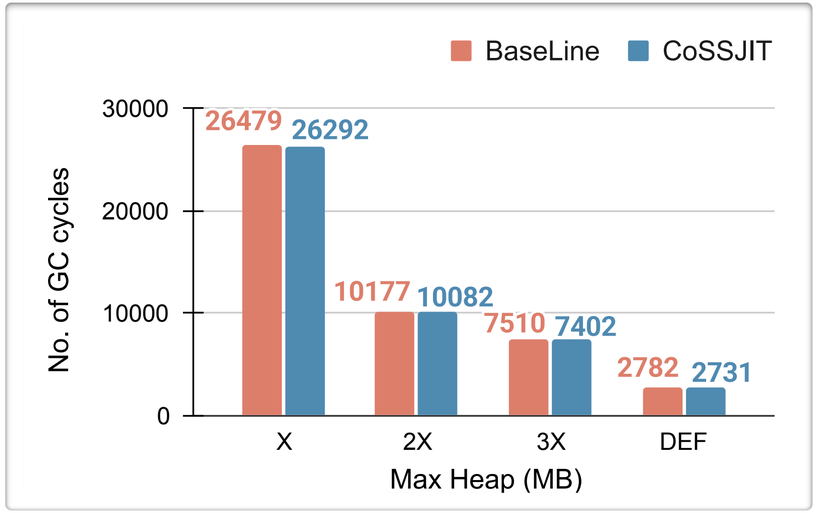




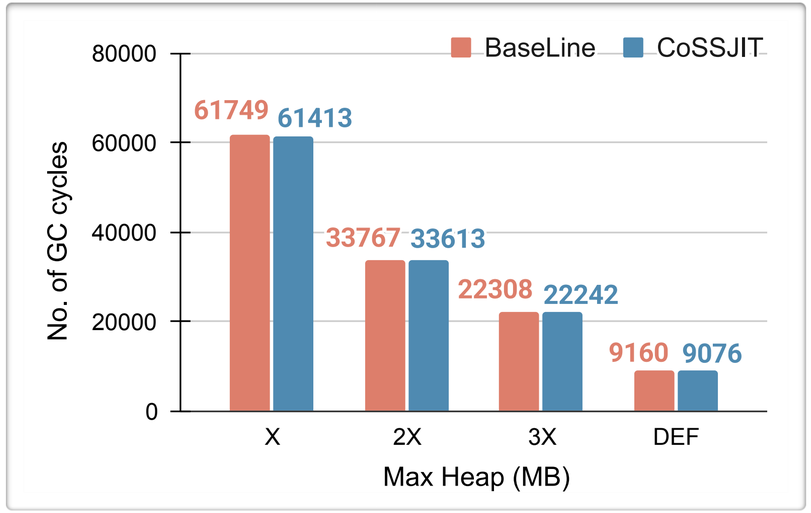
Garbage Collection



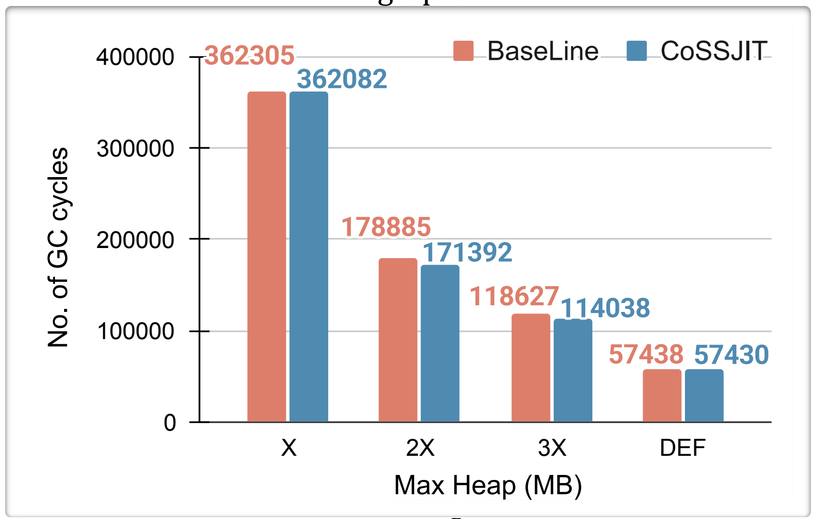
graphchi



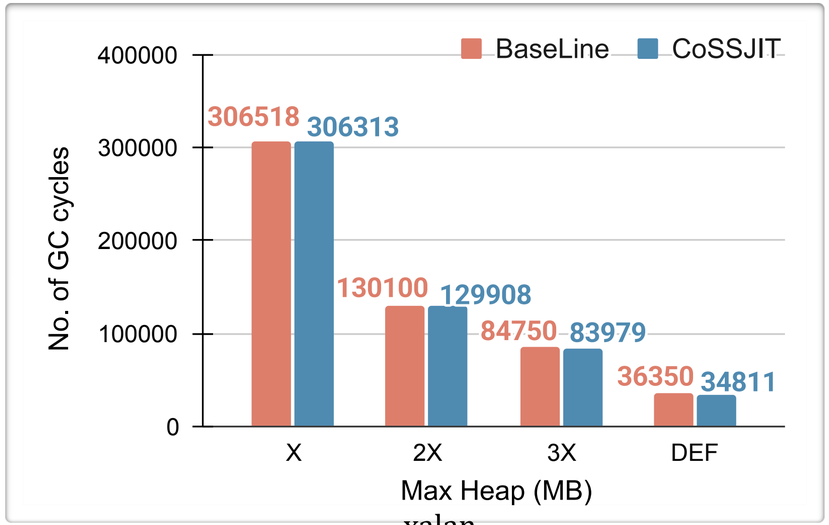
luindex



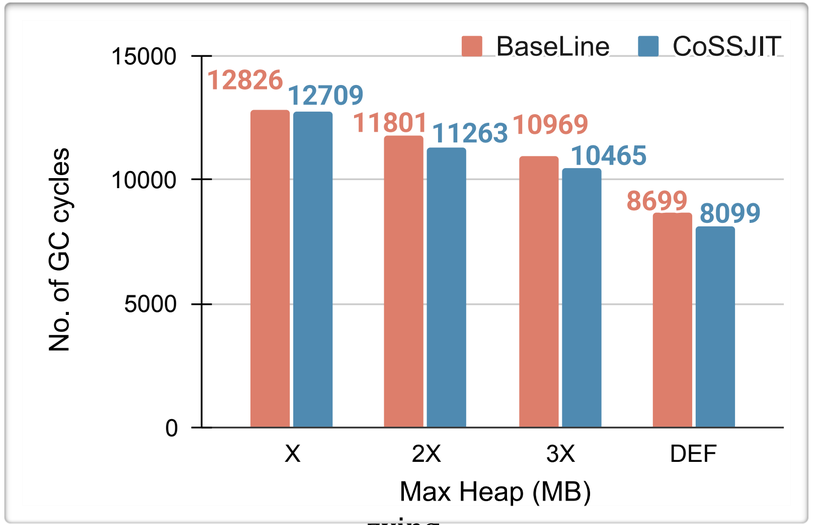
lusearch



sunflow

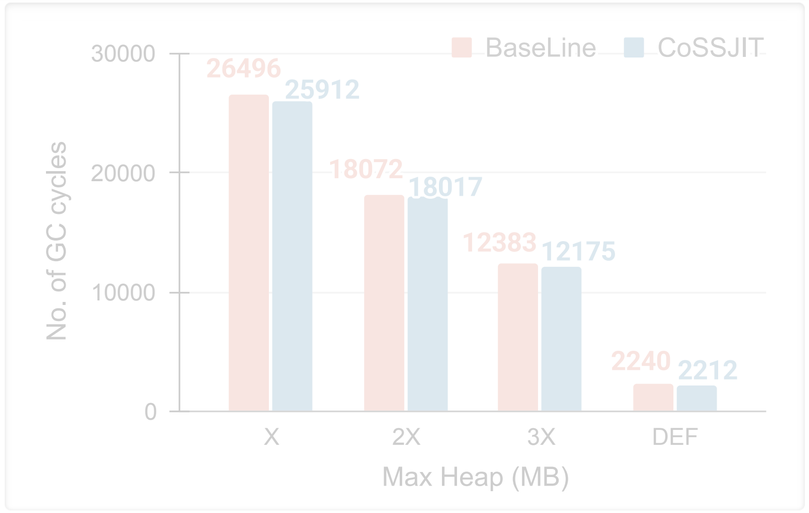


xalan

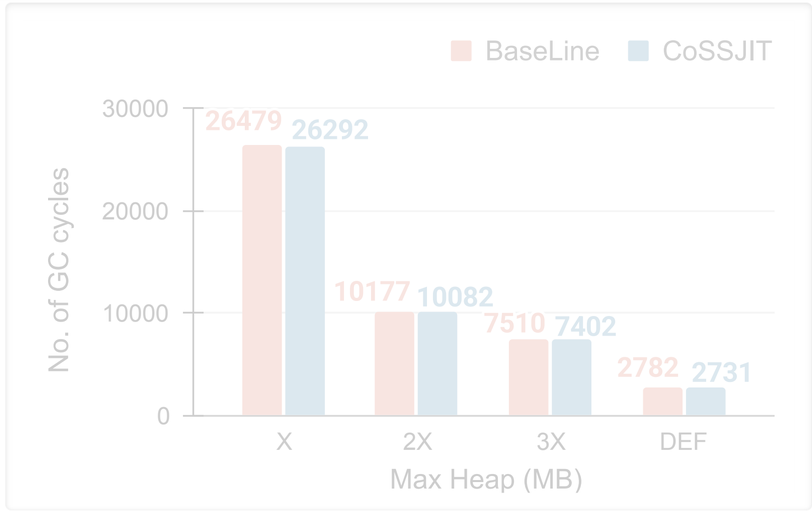


zxing

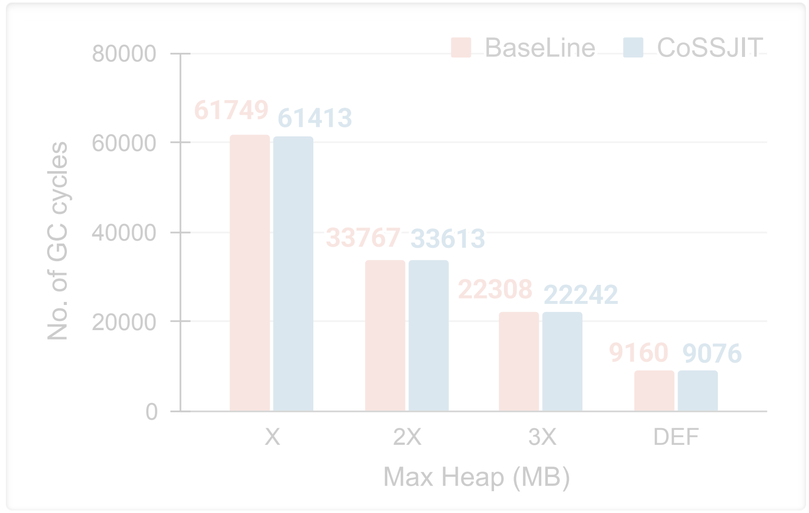
Garbage Collection



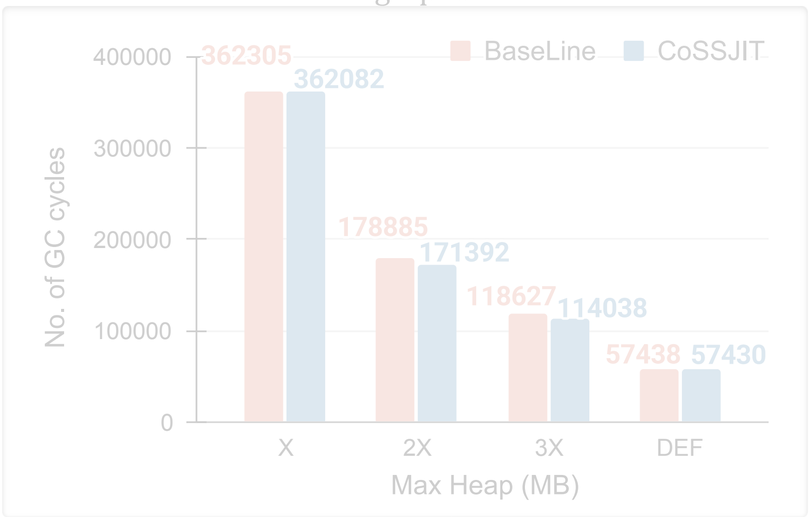
graphchi



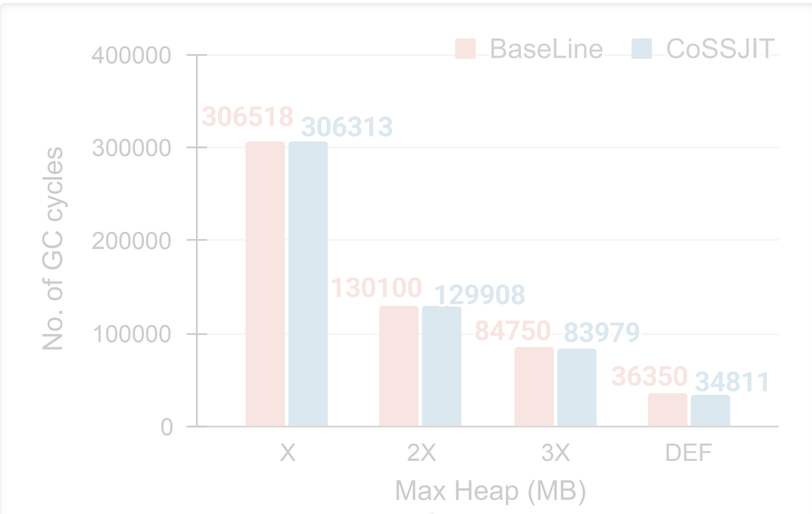
luindex



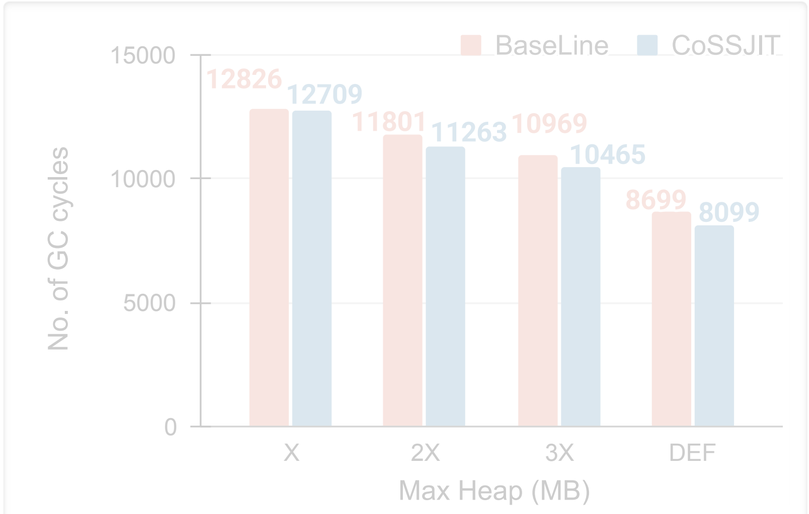
lusearch



sunflow

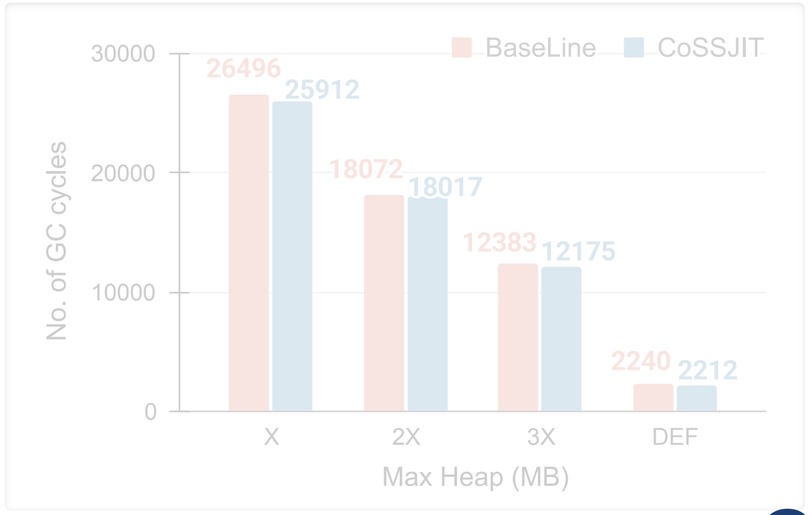


xalan

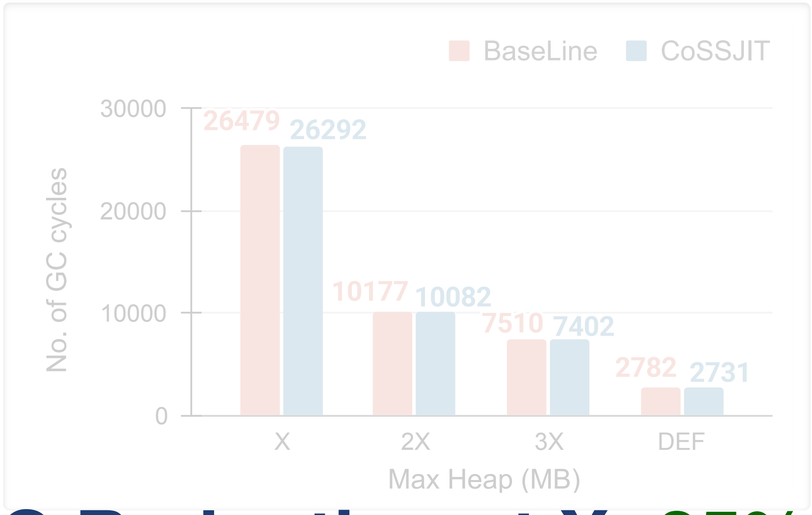


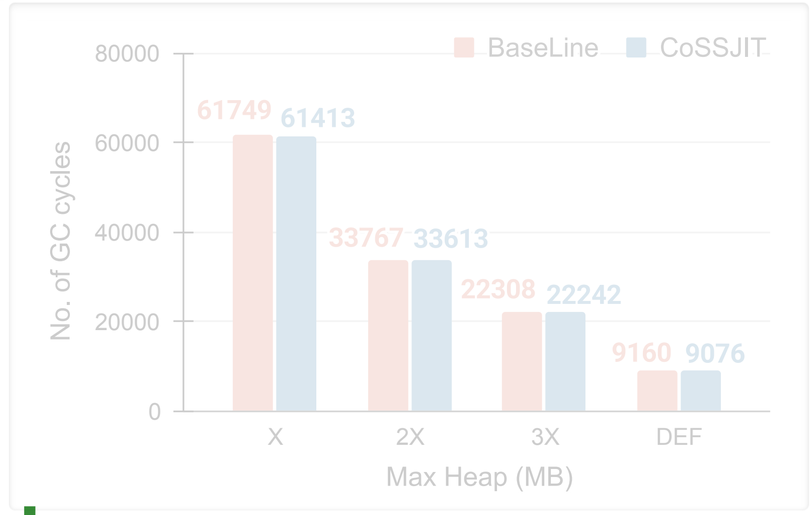
zxing

Garbage Collection



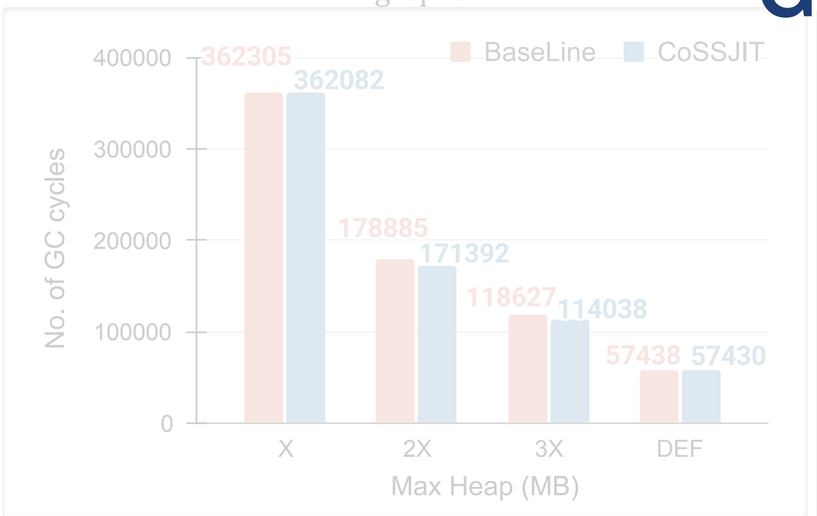
graphchi



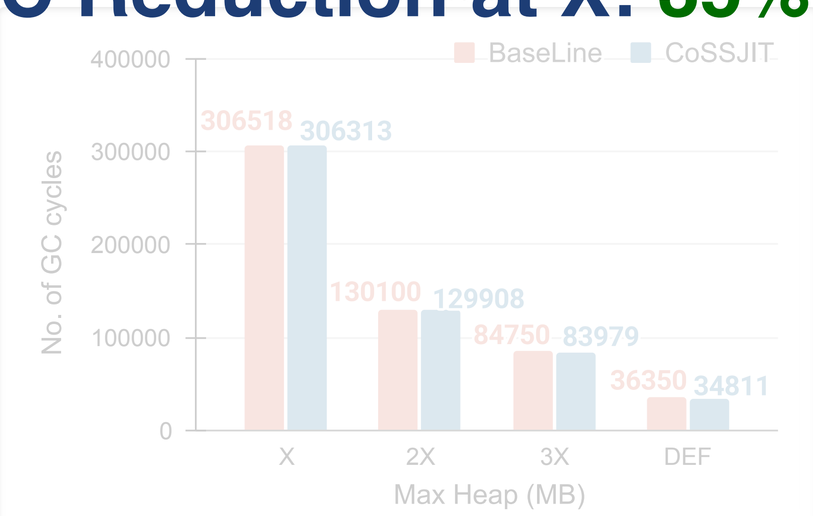


lusearch

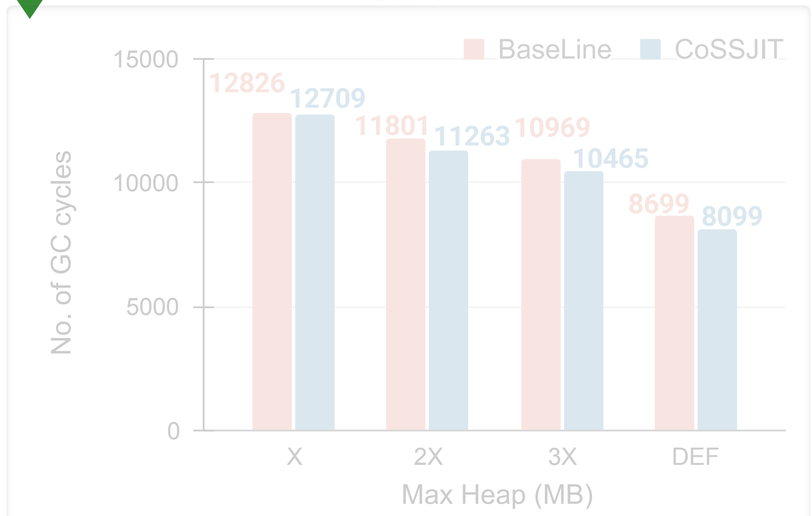
GC Reduction at X: 35% ↓



sunflow



xalan



zxing

Overheads

Benchmark	Static Time(s)	.class size (MB)	.res size (MB)	Total ET (s)	Improvement (s)	JIT Overhead(s)
avrora	92	8.7	0.29	294	1.6	0.2
fop	735	27	1.40	264	1.0	0.9
graphchi	105	9.8	0.34	2524	325.3	0.2
h2	160	2.2	0.35	173	4.0	0.06
luindex	90	1.3	0.28	165	6.0	0.7
pmd	277	27	0.67	570	41.0	0.8
sunflow	131	11	0.37	448	4.1	0.08
xalan	113	11	0.38	415	6.0	0.7
zxing	157	11	0.45	1298	20.2	0.5

Overheads

Benchmark	Static Time(s)	.class size (MB)	.res size (MB)	Total ET (s)	Improvement (s)	JIT Overhead(s)
avrora	92	8.7	0.29	294	1.6	0.2
fop	735	27	1.40	264	1.0	0.9
graphchi	105	9.8	0.34	2524	325.3	0.2
h2	160	2.2	0.35	173	4.0	0.06
luindex	90	1.3	0.28	165	6.0	0.7
pmd	277	27	0.67	570	41.0	0.8
sunflow	131	11	0.37	448	4.1	0.08
xalan	113	11	0.38	415	6.0	0.7
zxing	157	11	0.45	1298	20.2	0.5

Overheads

Benchmark	Static Time(s)	.class size (MB)	.res size (MB)	Total ET (s)	Improvement (s)	JIT Overhead(s)
avrora	92	8.7	0.29	294	1.6	0.2
fop	735	27	1.40	264	1.0	0.9
graphchi	105	9.8	0.34	2524	325.3	0.2
h2	160	2.2	0.35	173	4.0	0.06
luindex	90	1.3	0.28	165	6.0	0.7
pmd	277	27	0.67	570	41.0	0.8
sunflow	131	11	0.37	448	4.1	0.08
xalan	113	11	0.38	415	6.0	0.7
zxing	157	11	0.45	1298	20.2	0.5

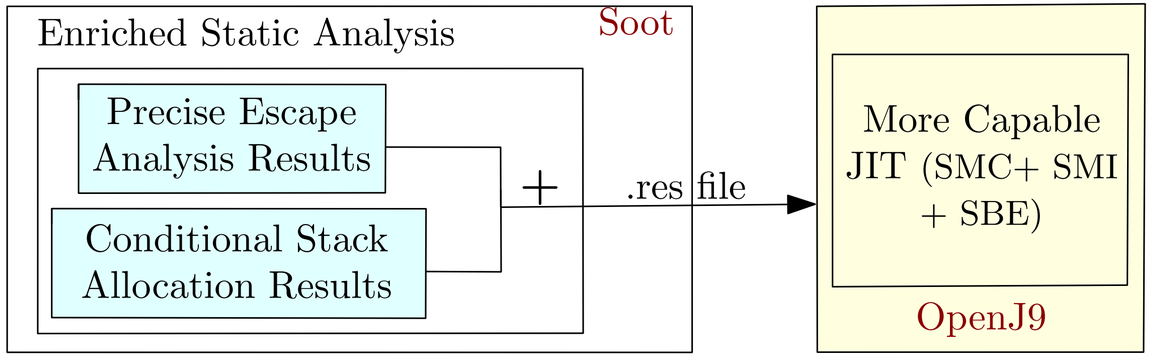
Take Aways

Take Aways

To summarize: Proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis.

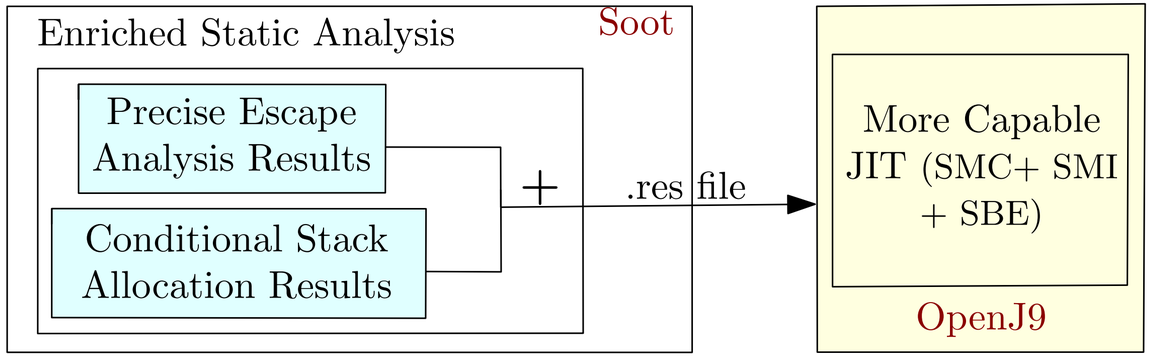
Take Aways

To summarize: Proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis.



Take Aways

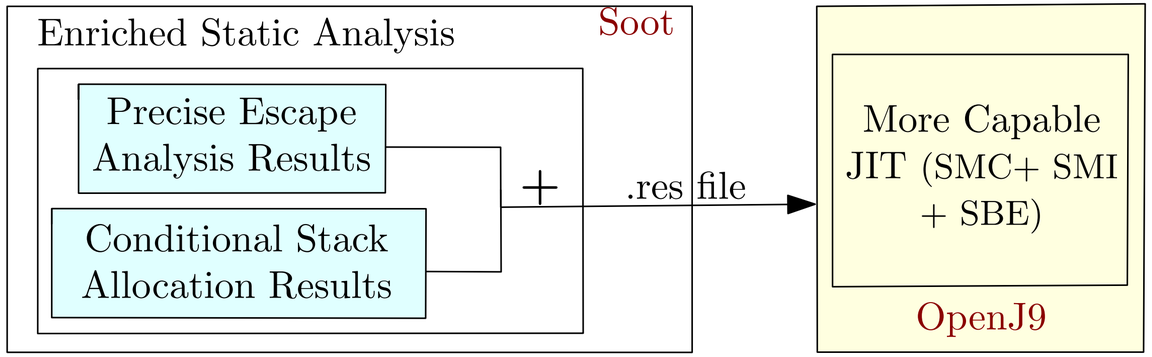
To summarize: Proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis.



- Enriched the static analysis with the possible run-time based speculative results.

Take Aways

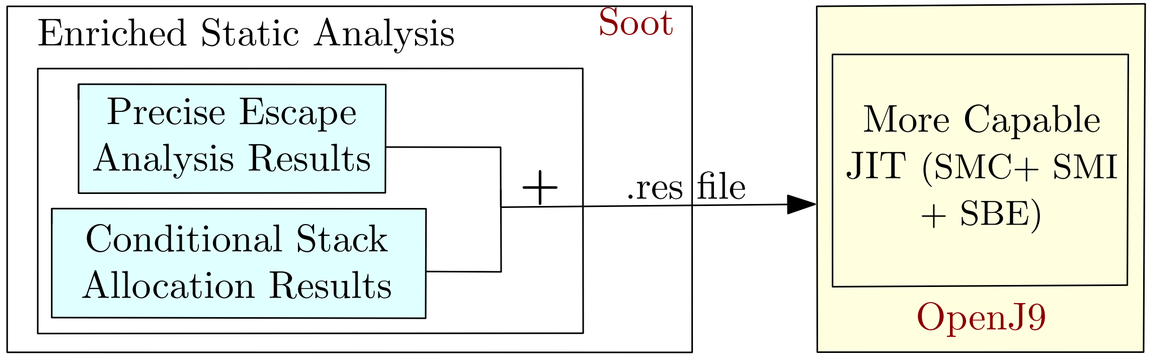
To summarize: Proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis.



- Enriched the static analysis with the possible run-time based speculative results.
- Mechanism in the JIT compiler to incorporate the conditional static analysis results.

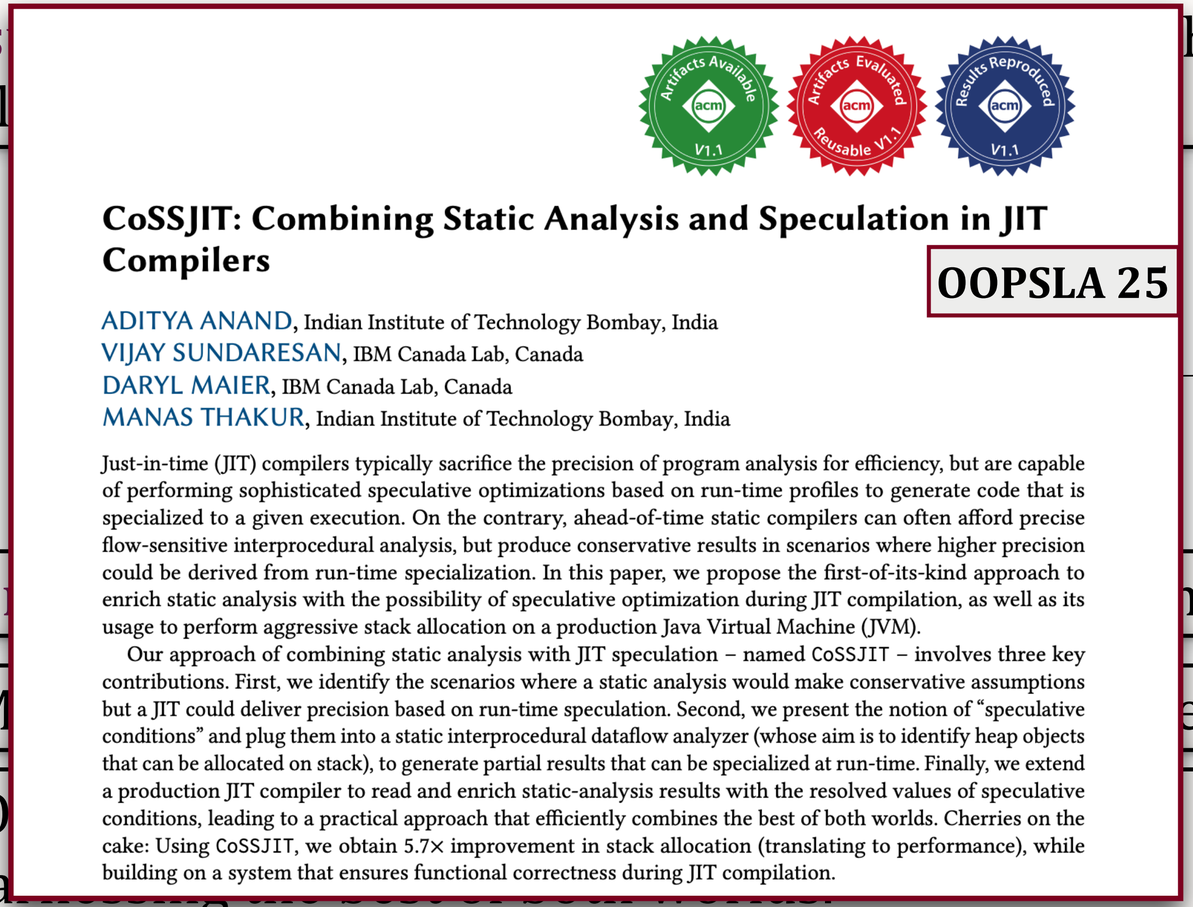
Take Aways

To summarize: Proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis.



- Enriched the static analysis with the possible run-time based speculative results.
- Mechanism in the JIT compiler to incorporate the conditional static analysis results.
- Overall, one of the first approach that strikes a balance between static analysis and JIT, harnessing the best of both worlds.

Take Aways


Artifacts Available
Artifacts Evaluated
Reusable V1.1
Results Reproduced
V1.1
CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers
OOPSLA 25
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.

Take Aways

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

OOPSLA 25

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.

Thank You !!