

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2025)

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

*IIT Bombay, †IBM Canada



17th October 2025

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

Principles of Staged Static+Dynamic Partial Analysis

SAS 22

Aditya Anand^{id} and Manas Thakur^{(✉)id}

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

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

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

PLDI 24

RESEARCH

Partial program analysis for staged compilation systems

Aditya Anand¹ · Manas Thakur¹

FMSD 24

Received: 30 April 2023 / Accepted: 16 May 2024 / Published online: 13 June 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

All these works use AOT-analysis results to perform optimizations in JIT Compilers.

Optimistic Stack Allocation

- Optimizes Object Allocation
- **Stack Allocation:** Allocate method local objects on the stack frame.
- Perform precise **escape analysis** **statically** and **optimistically** allocate objects on stack at run-time.
- Need to ensure **functional correctness**.

- First approach to **soundly** and **efficiently** use static (offline) analysis results in a **JIT compiler!**



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

PLDI 24

- Efficient run-time checks to determine if the static-analysis results correspond to the **runtime environment**; if not — **Heapify**.

Speculative Optimization in JIT Compilers

- Profile Information:

Standard Speculative Optimizations

- Basic invocation, loop invariant values.
- Type profiles at type-cast statements and polymorphic callsites.
- Branch information, instance of checks.

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

- Dynamic Class Hierarchy:

- Information about loaded subclasses of a given class during execution.

- Inlining Table:

- Information about the list of methods inlined at various callsites.

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 */
```

$\sqcap : \{B, C\} \rightarrow [O_5], [O_7] \text{ [Escaping]}$

1. Polymorphic Callsites

```
1. class A {  
    . . .  
4. void foo(A z) {  
5.     A x = new A(); // 05  
6.     A y = new A(); // 06  
7.     x.f = new A(); // 07  
    . . .  
14.    z.bar(x);  
15. } /* method foo */  
16. } /* class A */
```

polymorphic_cond

A.foo() [..] [z₁₄, {B}, {0₅, 0₇}]

```
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**”.

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 */
```

$\sqcap : \rightarrow O_6$ (Escaping)

2. Branching

```
1. class A {  
    . . .  
5. void foo(A p1) {  
6.     A y = new A(); // 06  
    . . .  
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, {0₆}]

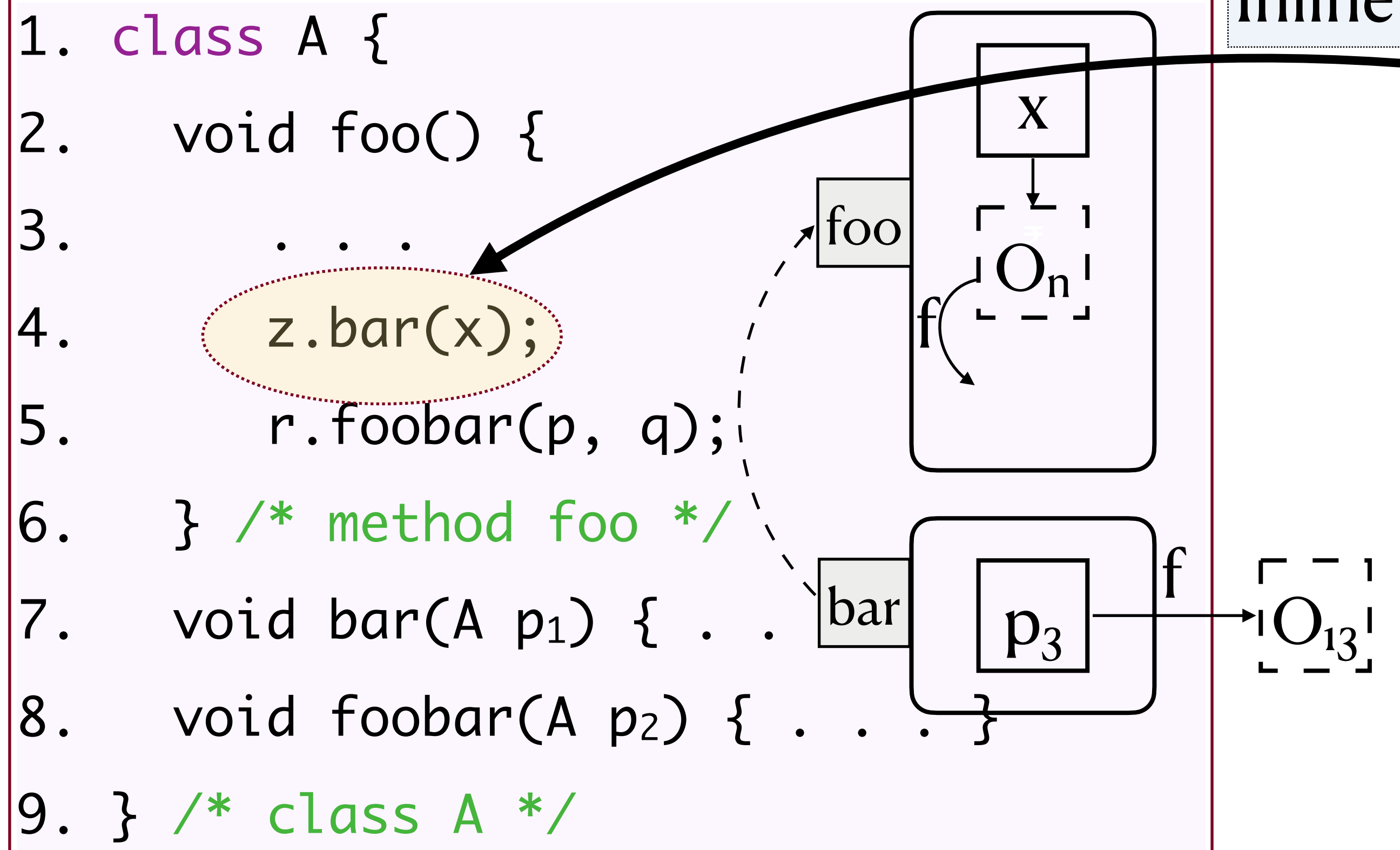
3. Method Inlining

```
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 { . . . }
```

O₁₃ marked as Escaping.

3. Method Inlining



```
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. } /* class B */  
17. }  
18. class C extends A { ... }
```

inlining_cond

A.foo() [..] [4, B.bar(p₃), {O₁₃}]

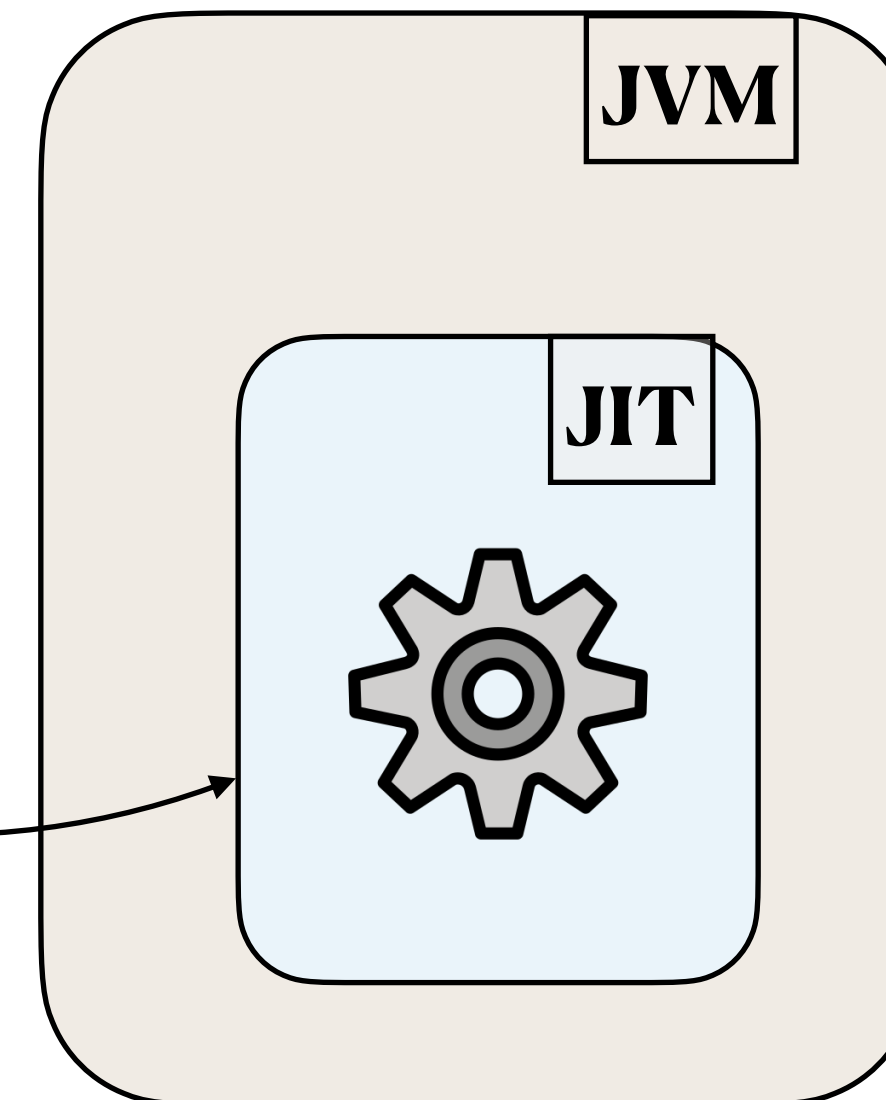
Summary

Direct

Conditional

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

Statically generated results



Algorithm in the
paper

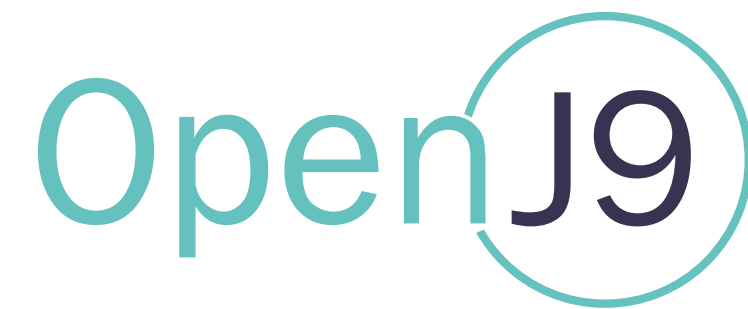
Evaluation (Setup)

- 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)

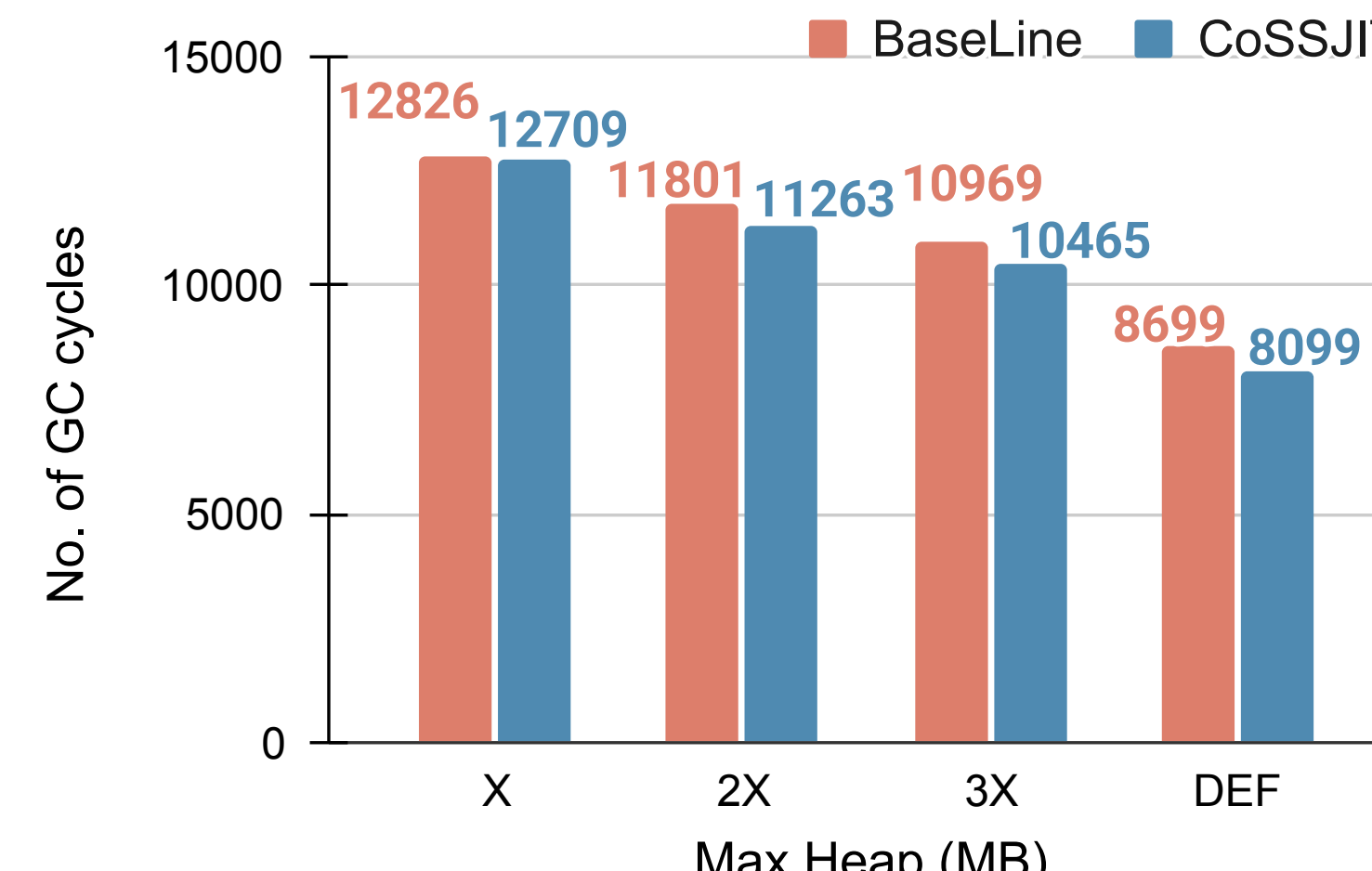
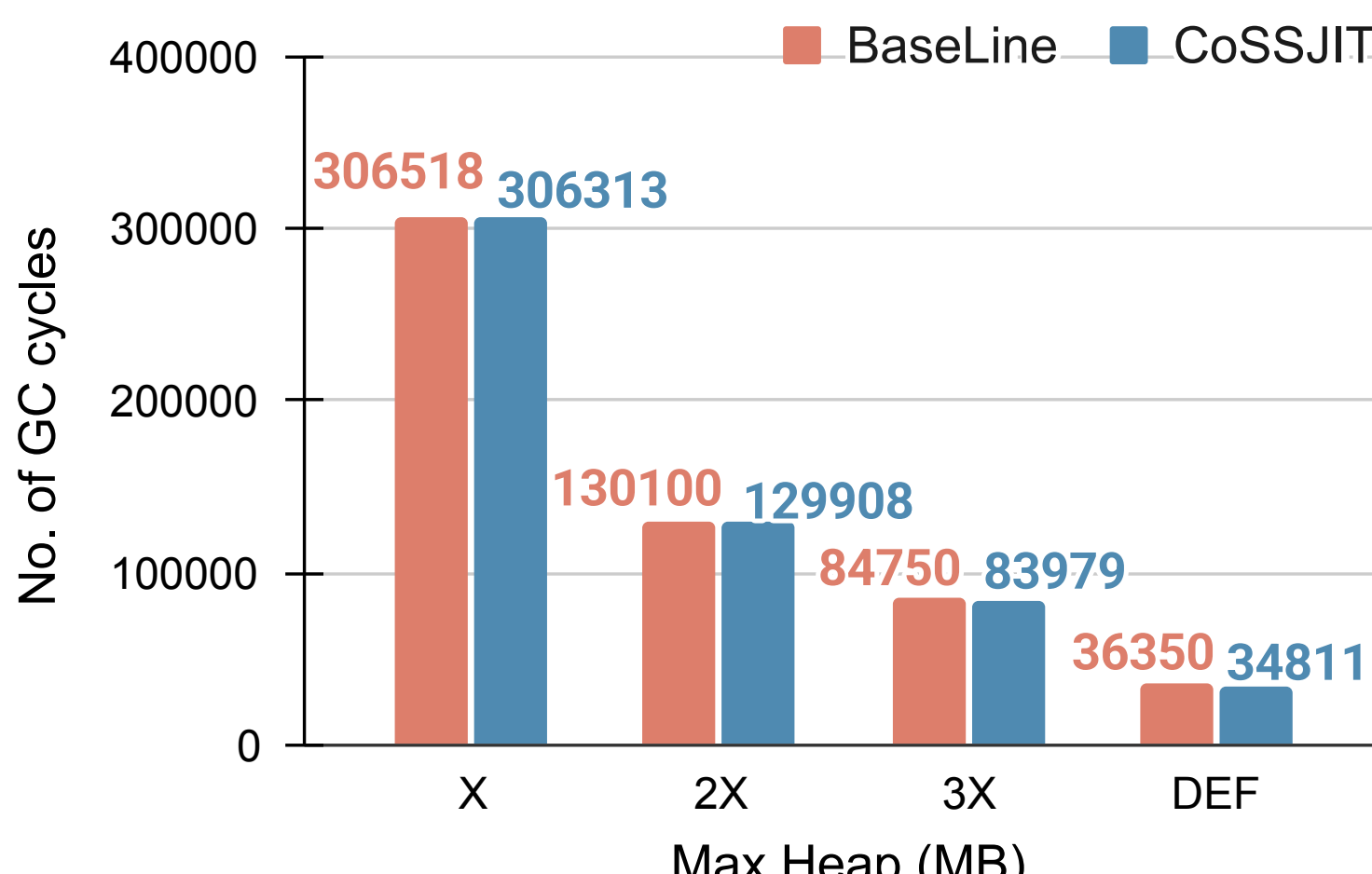
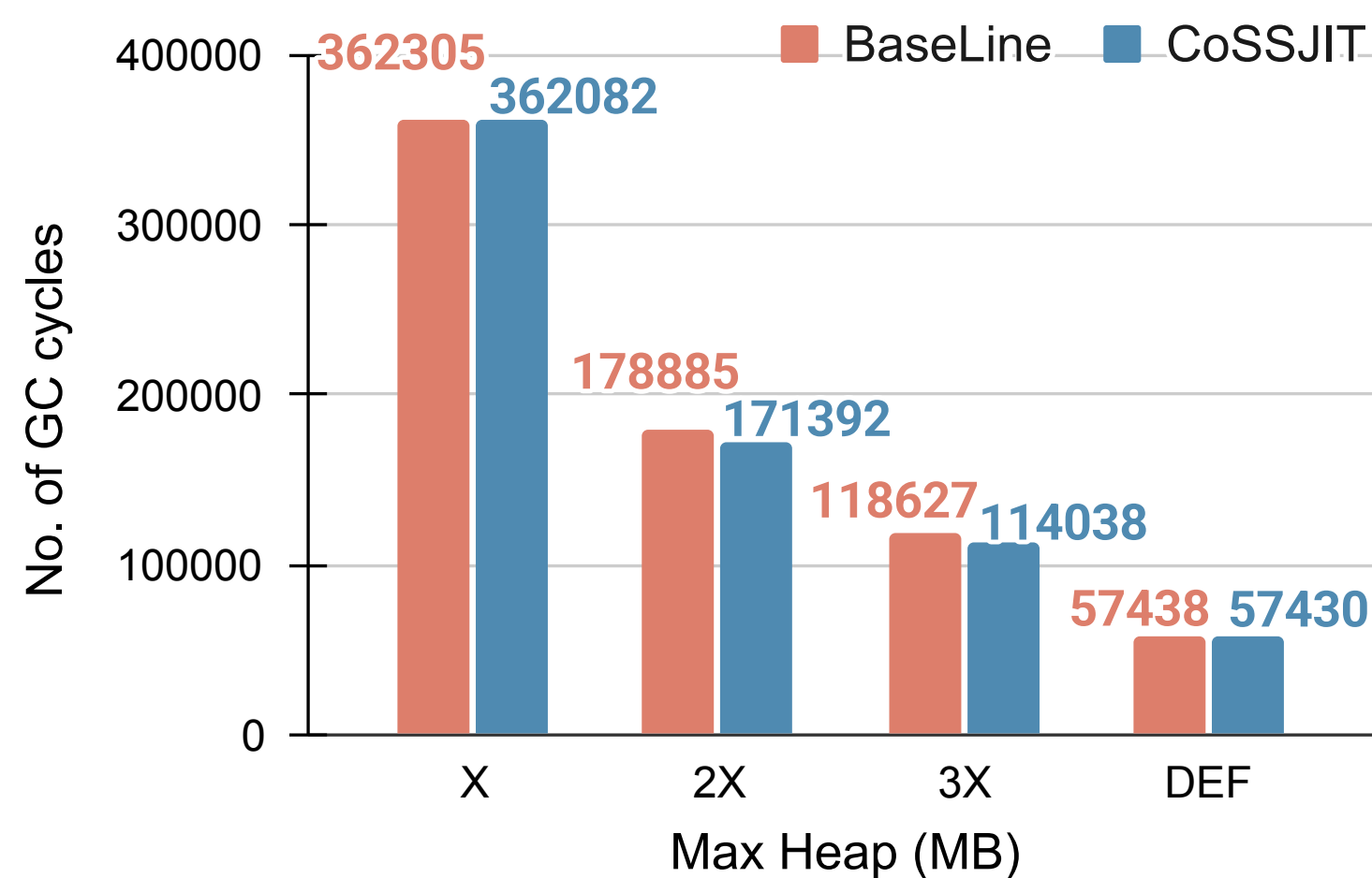
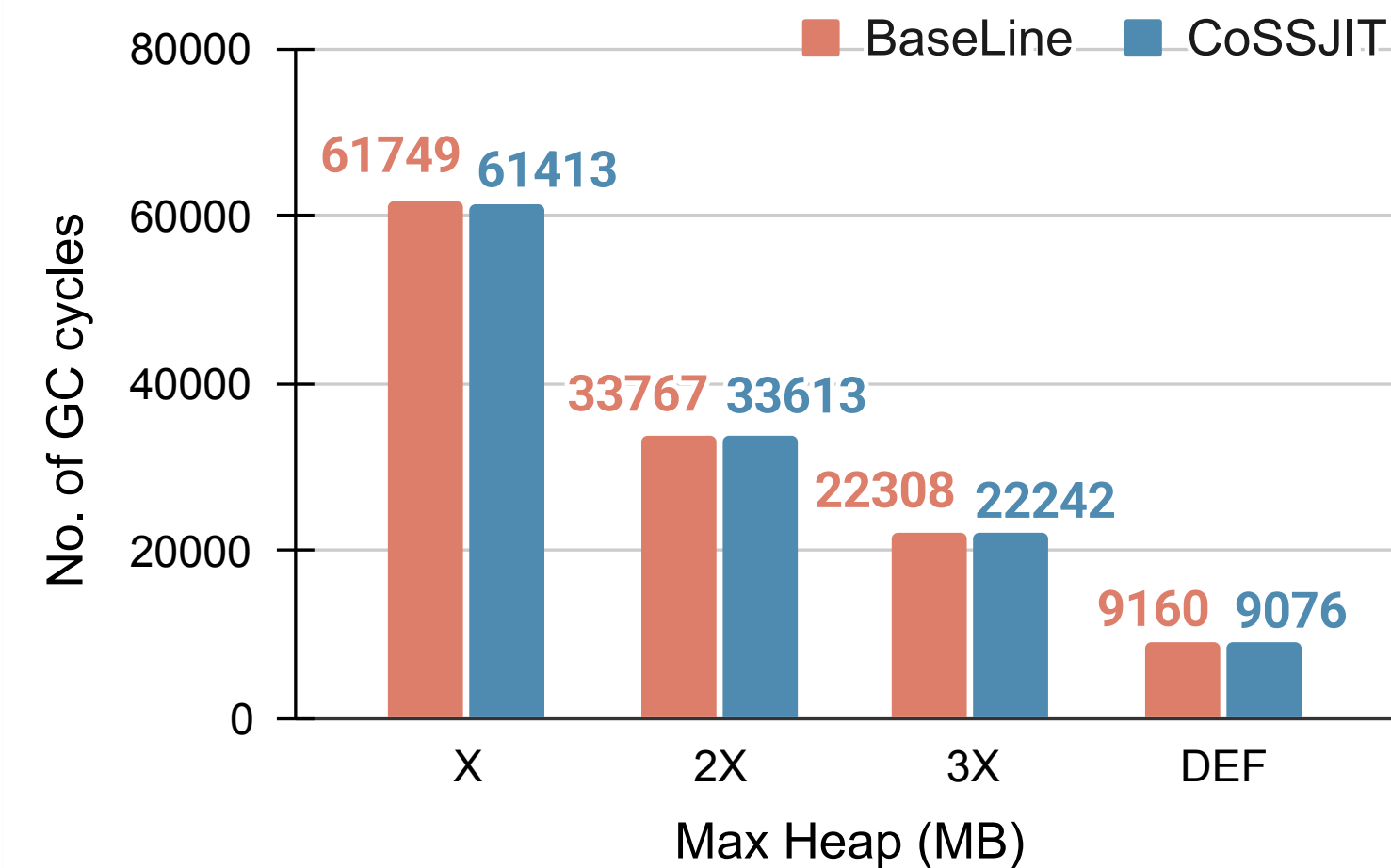
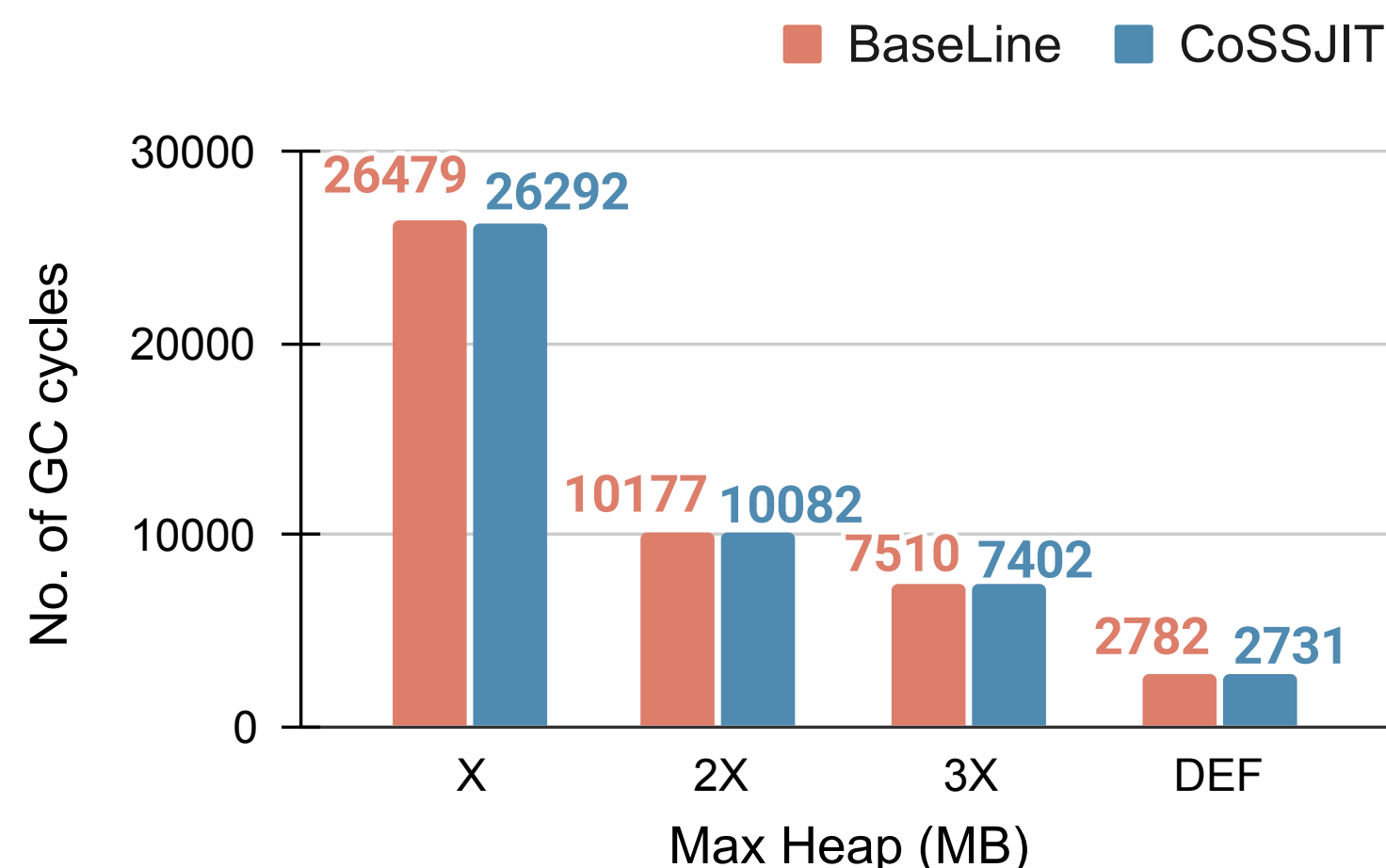
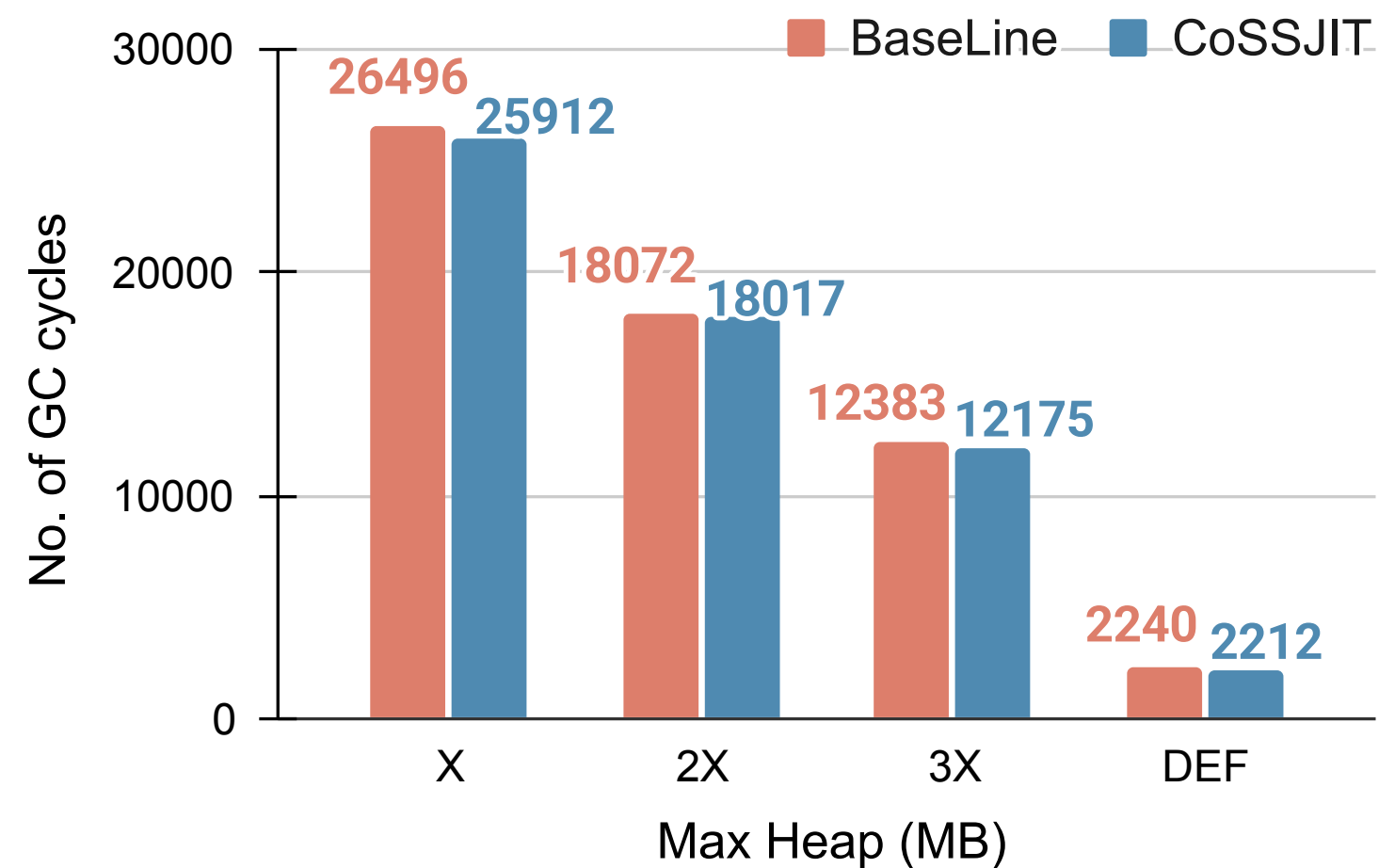
	Base Scheme (BaseLine)			Conditional Scheme (CoSSJIT)			
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes	Heapify Count
avroora	12	104.2M (38.0%)	3335MB	14	106.5M (39.3%)	3391.4MB	0.4M
compress	8	0.01M (3.29%)	1720MB	18	0.093M (15.5%)	2.75MB	0M
graphchi	12	349M (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)

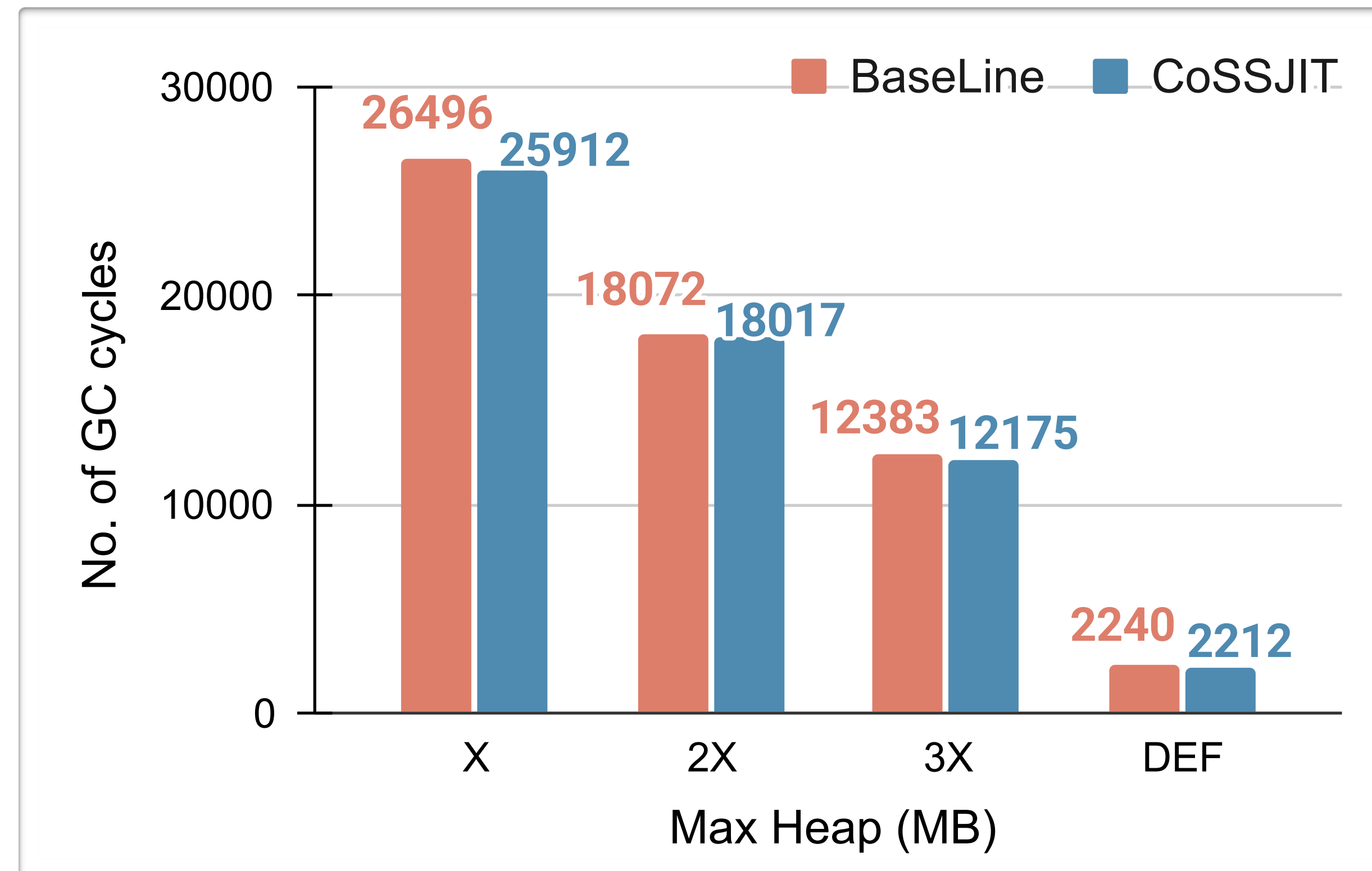
	Base Scheme (BaseLine)			Conditional Scheme (CoSSJIT)			
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes	Heapify Count
avroora	12	104.2M (38.0%)	3335MB	14	106.5M (39.3%)	3391.4MB	0.4M
compress	8	0.01M (3.29%)	1720MB	18	0.093M (15.5%)	2.75MB	0M
graphchi	12	149M (5.20%)	8327MB	104	104.1M (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

Stack Allocation: 1.4x↑ Stack Bytes: 5.7x↑
(Less Heap Allocation)

Garbage Collection

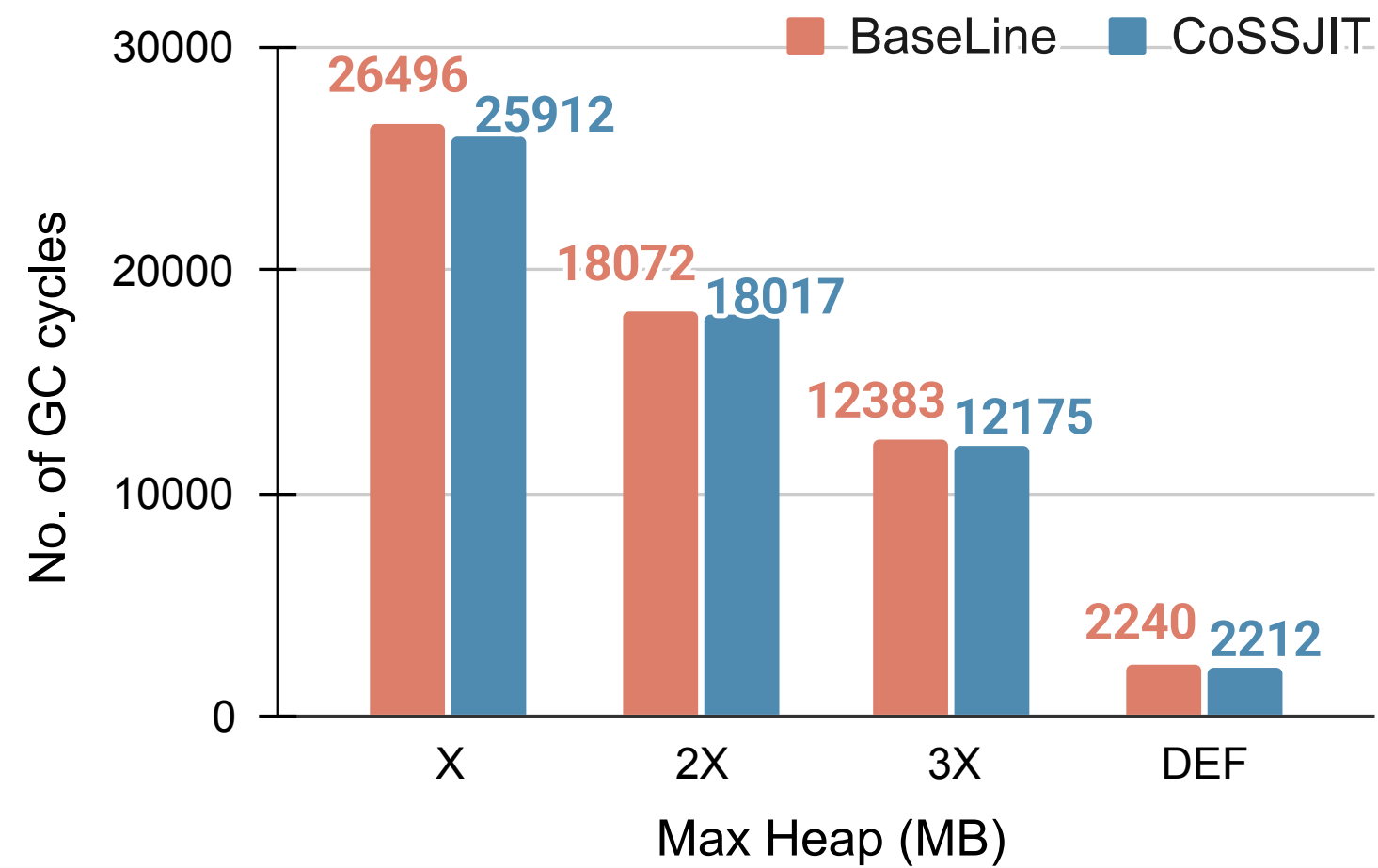


Garbage Collection

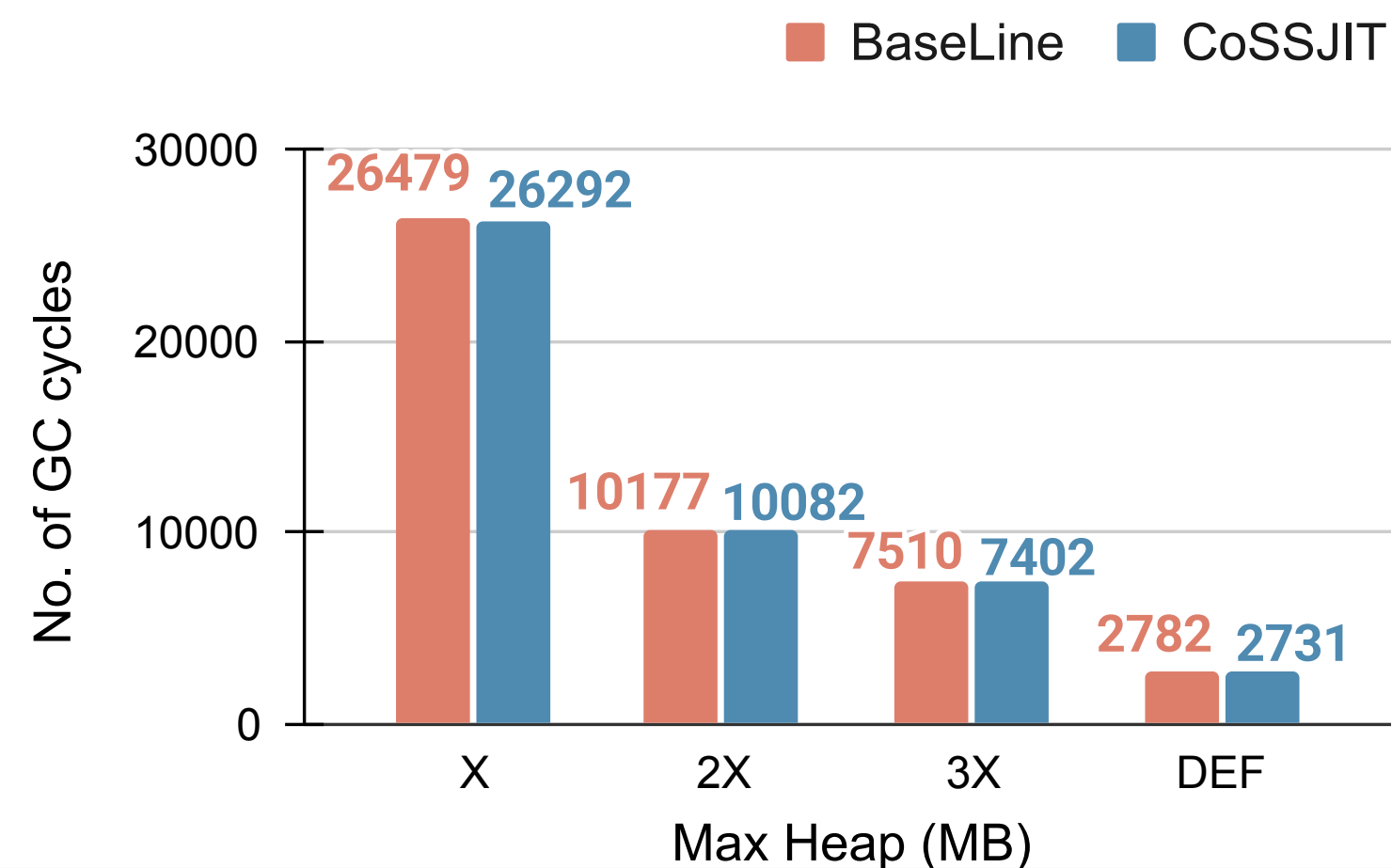


graphchi

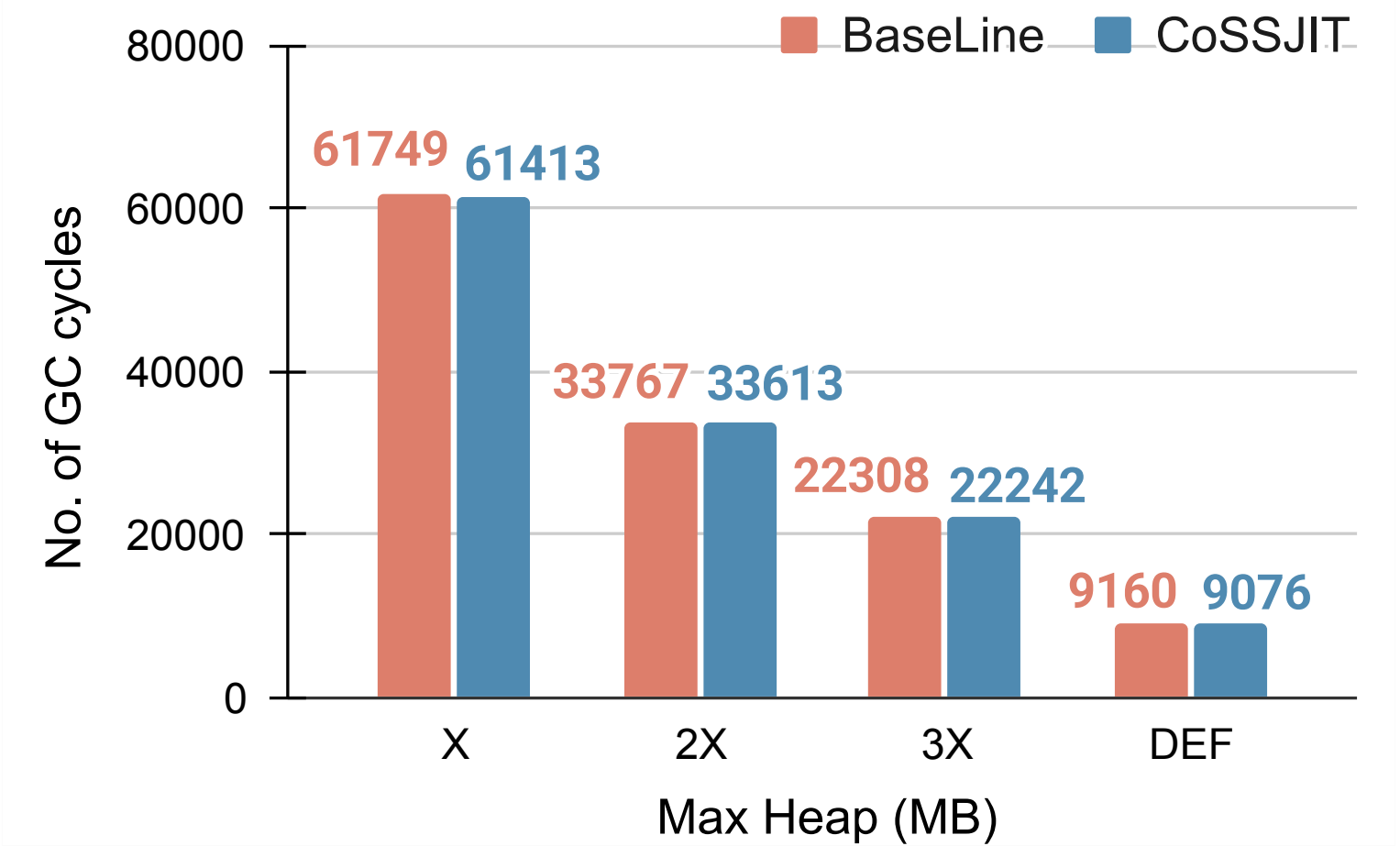
Garbage Collection



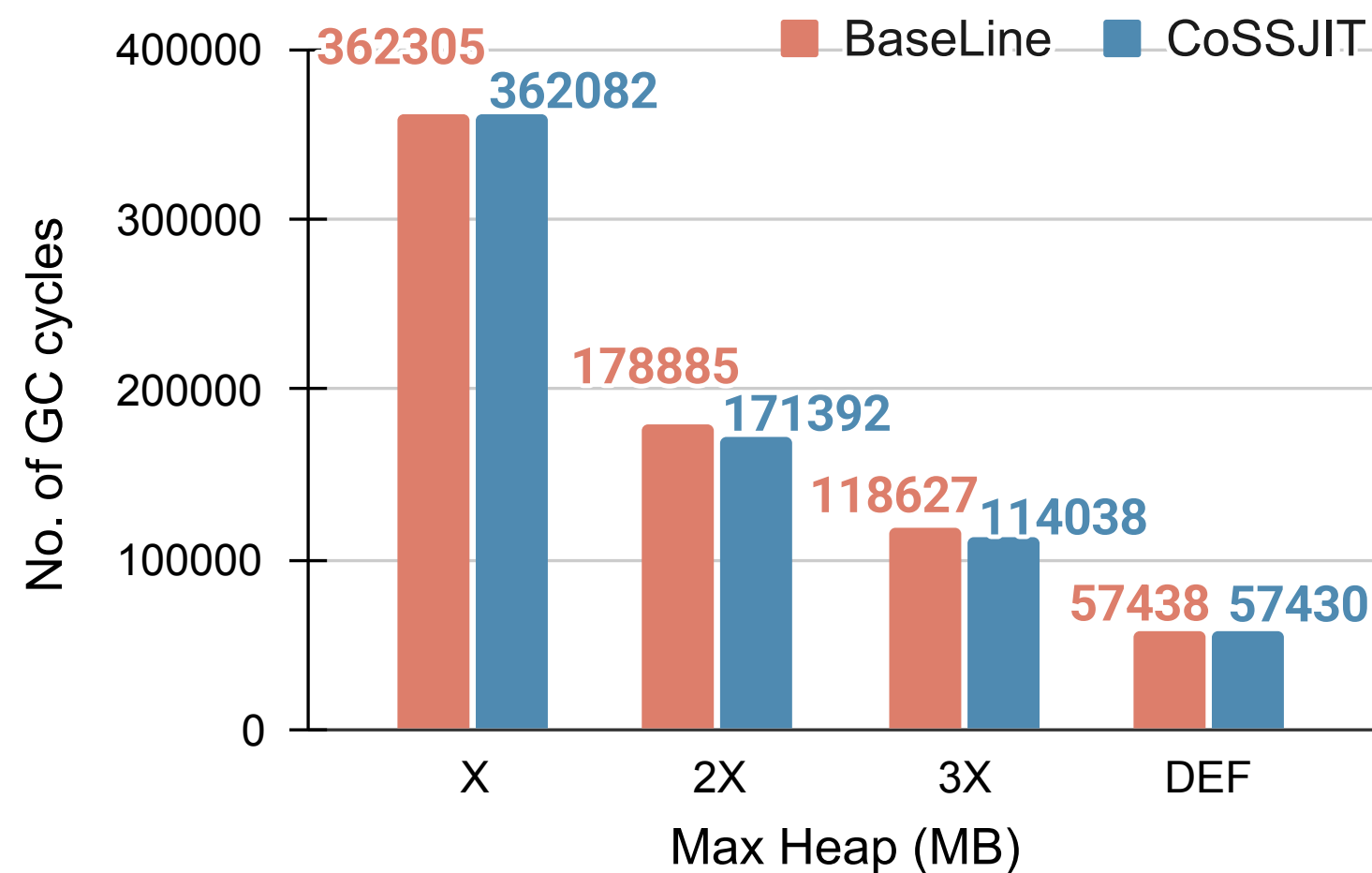
graphchi



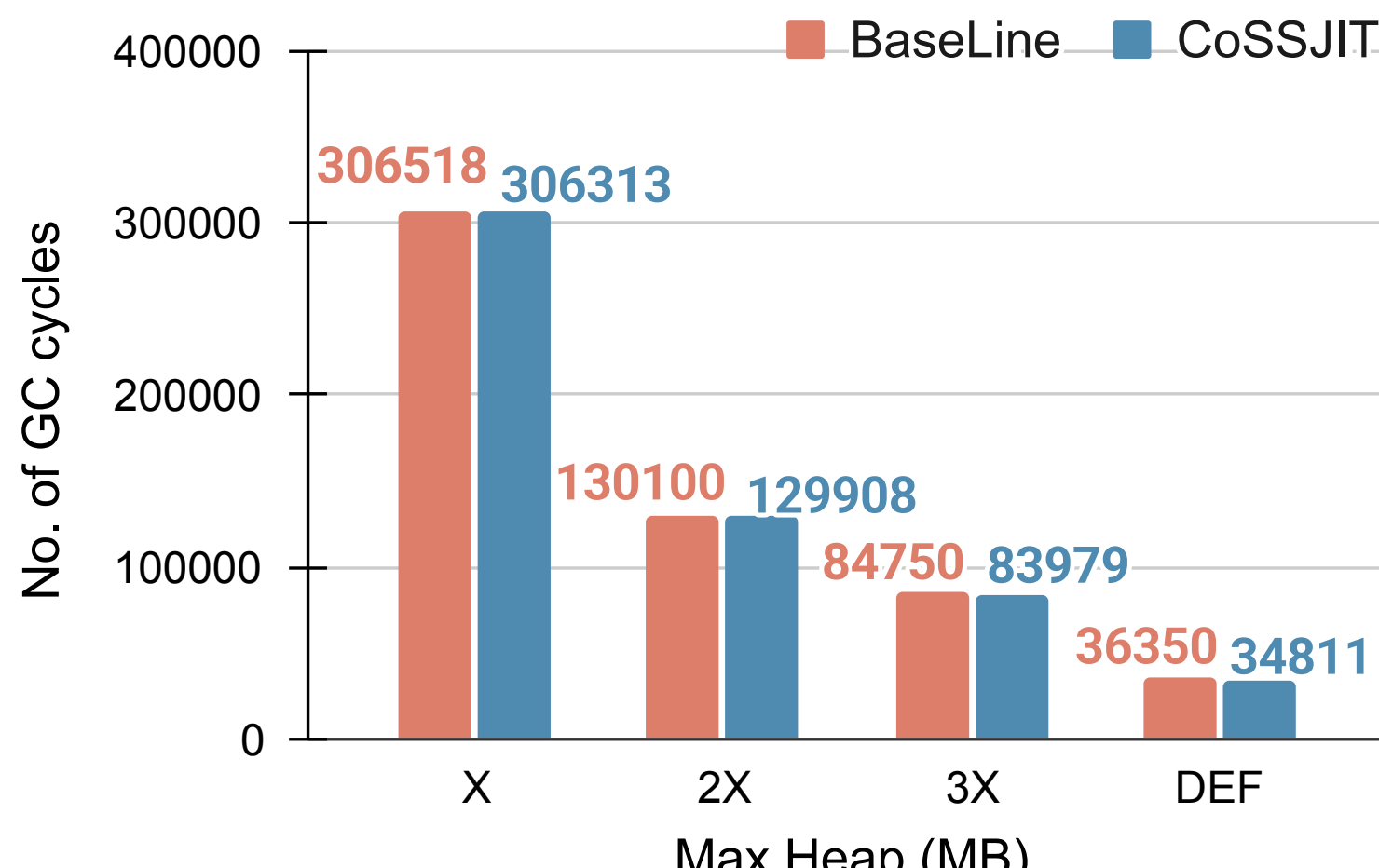
index



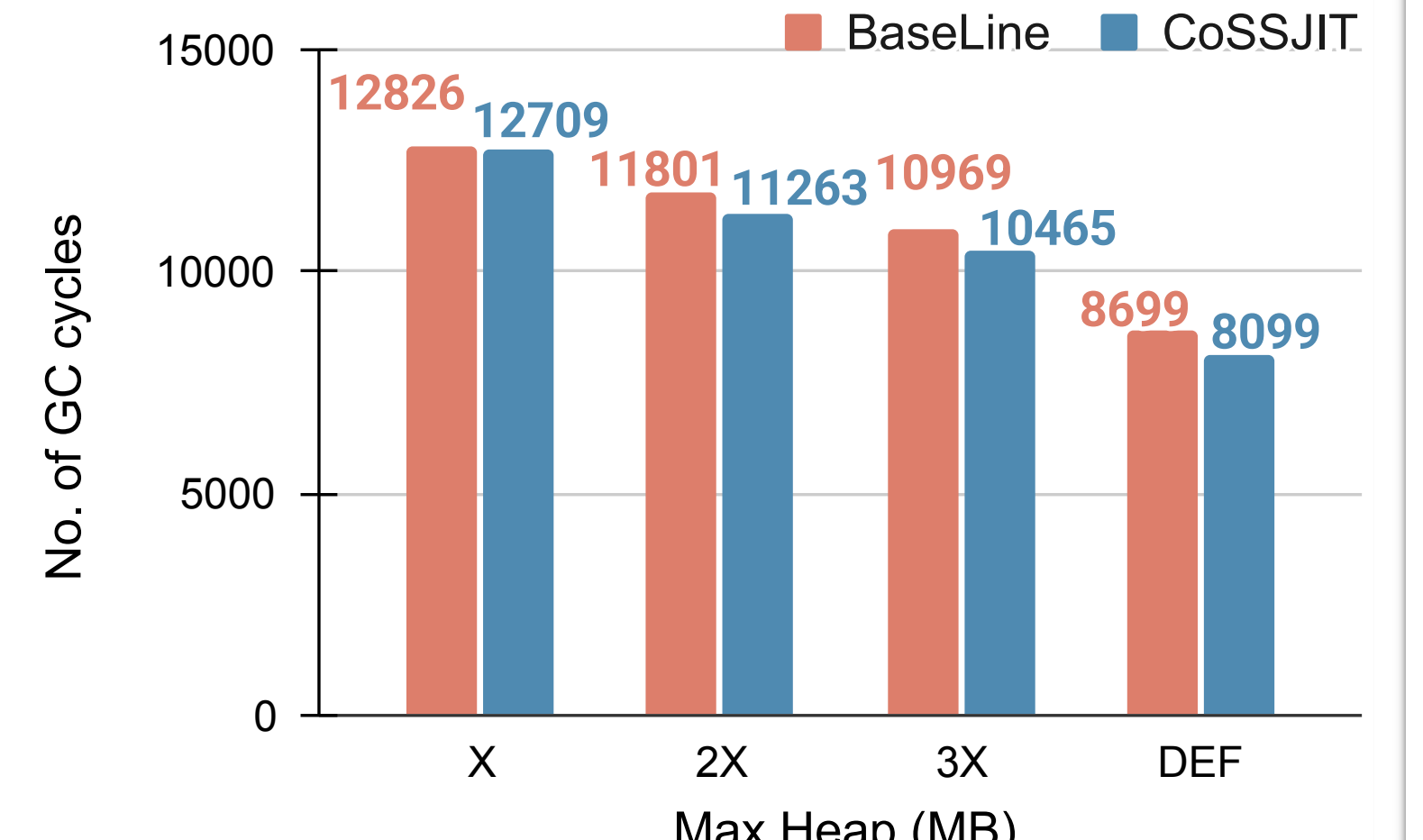
lusearch



sunflow



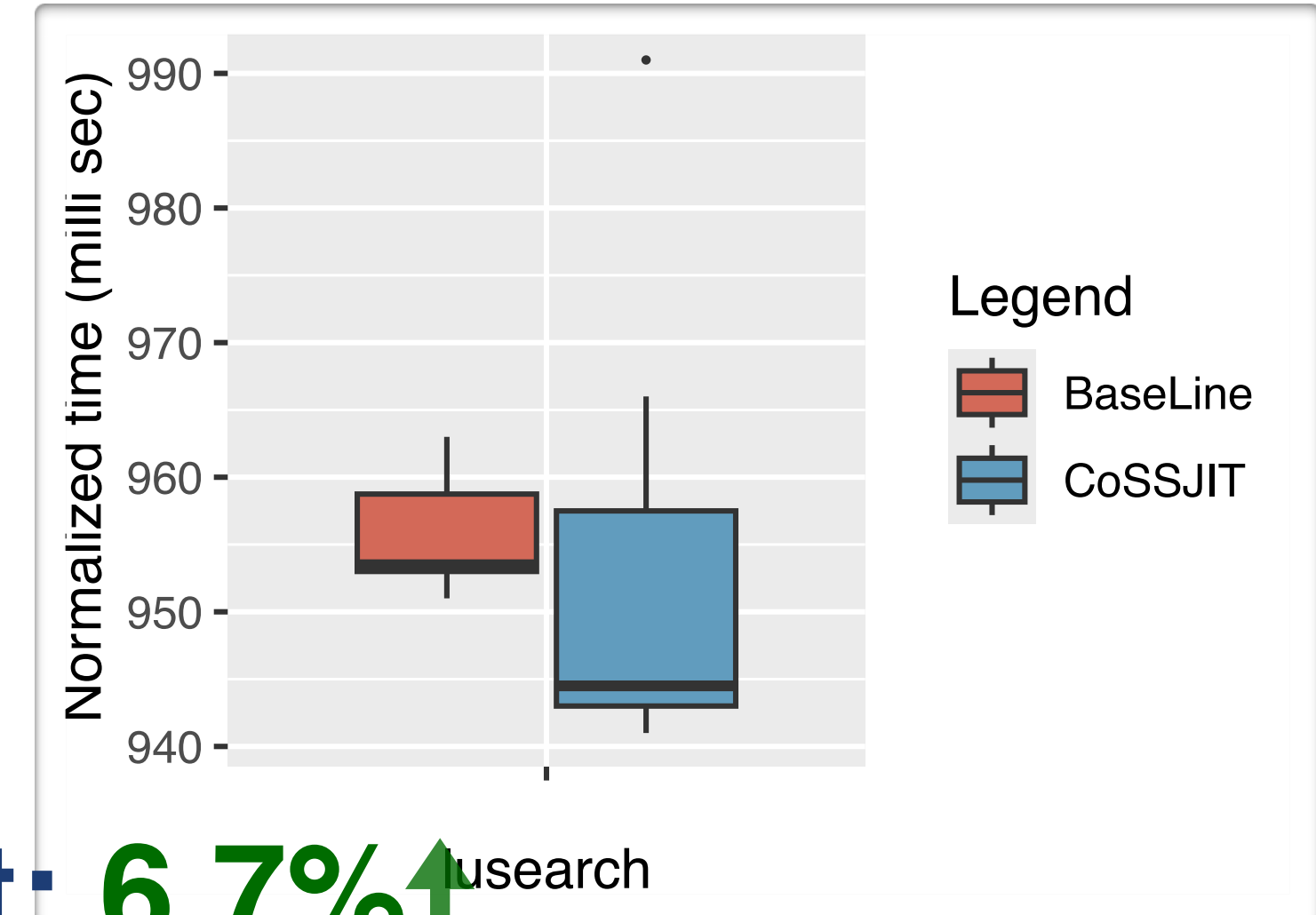
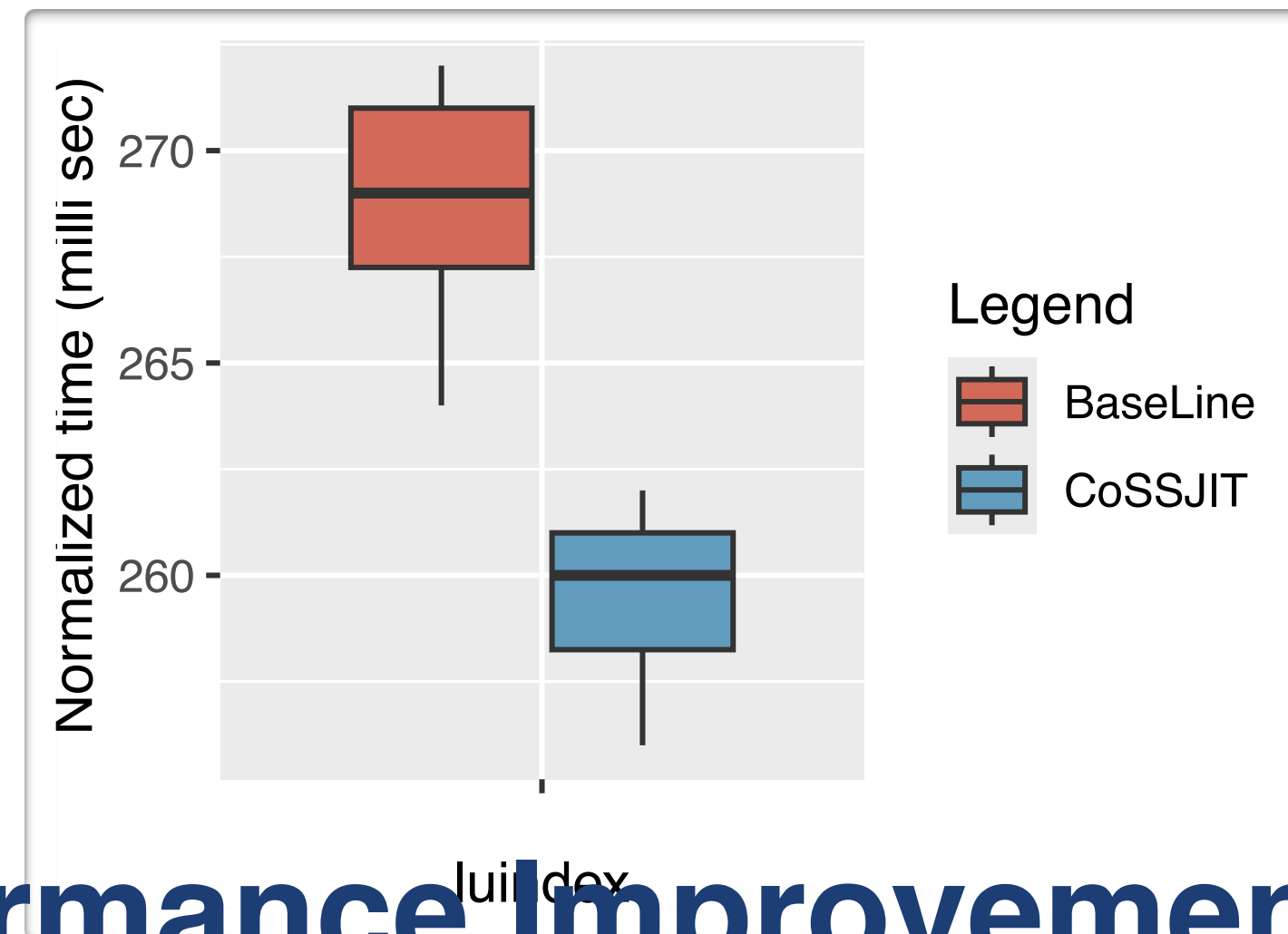
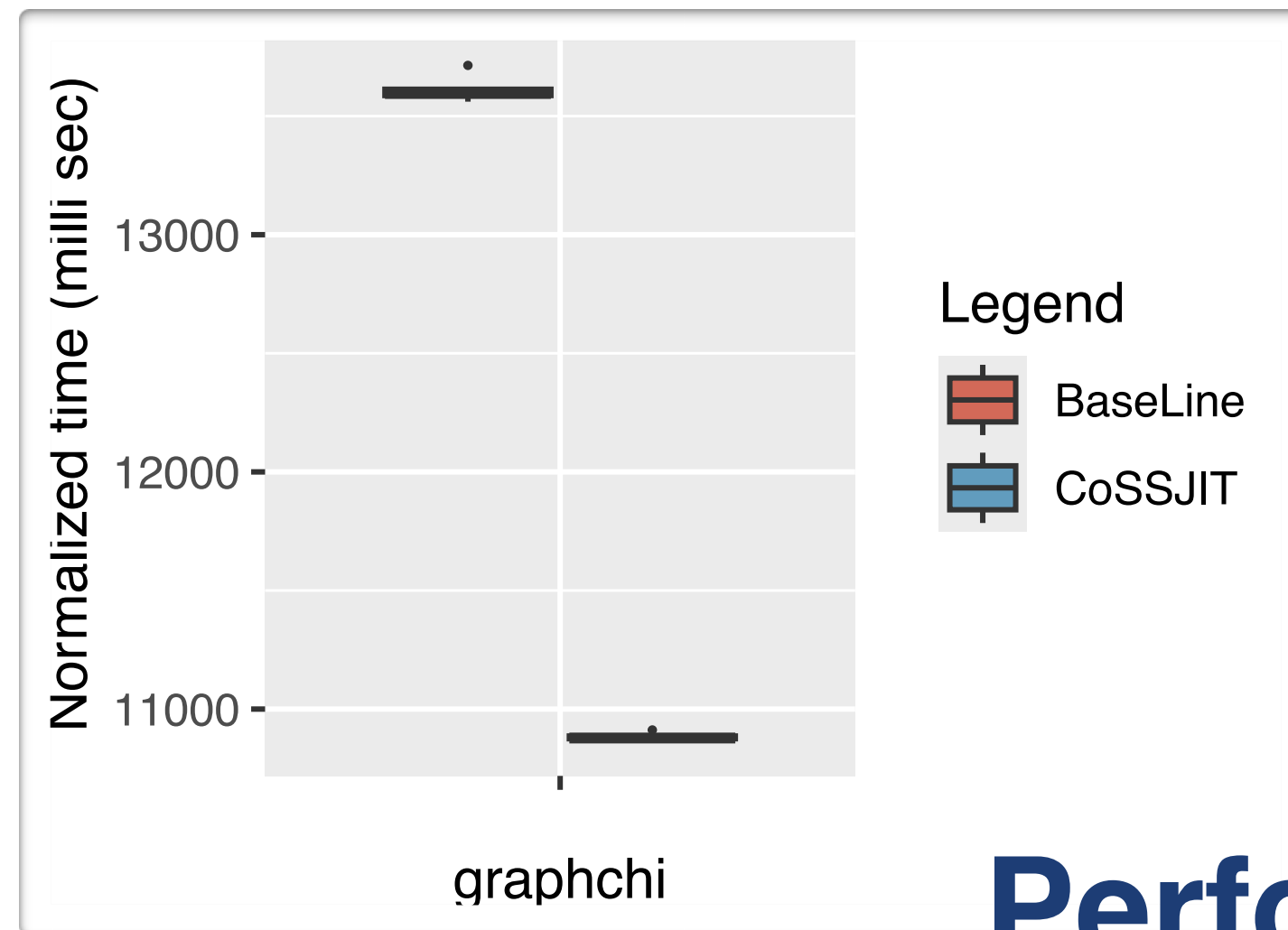
xalan



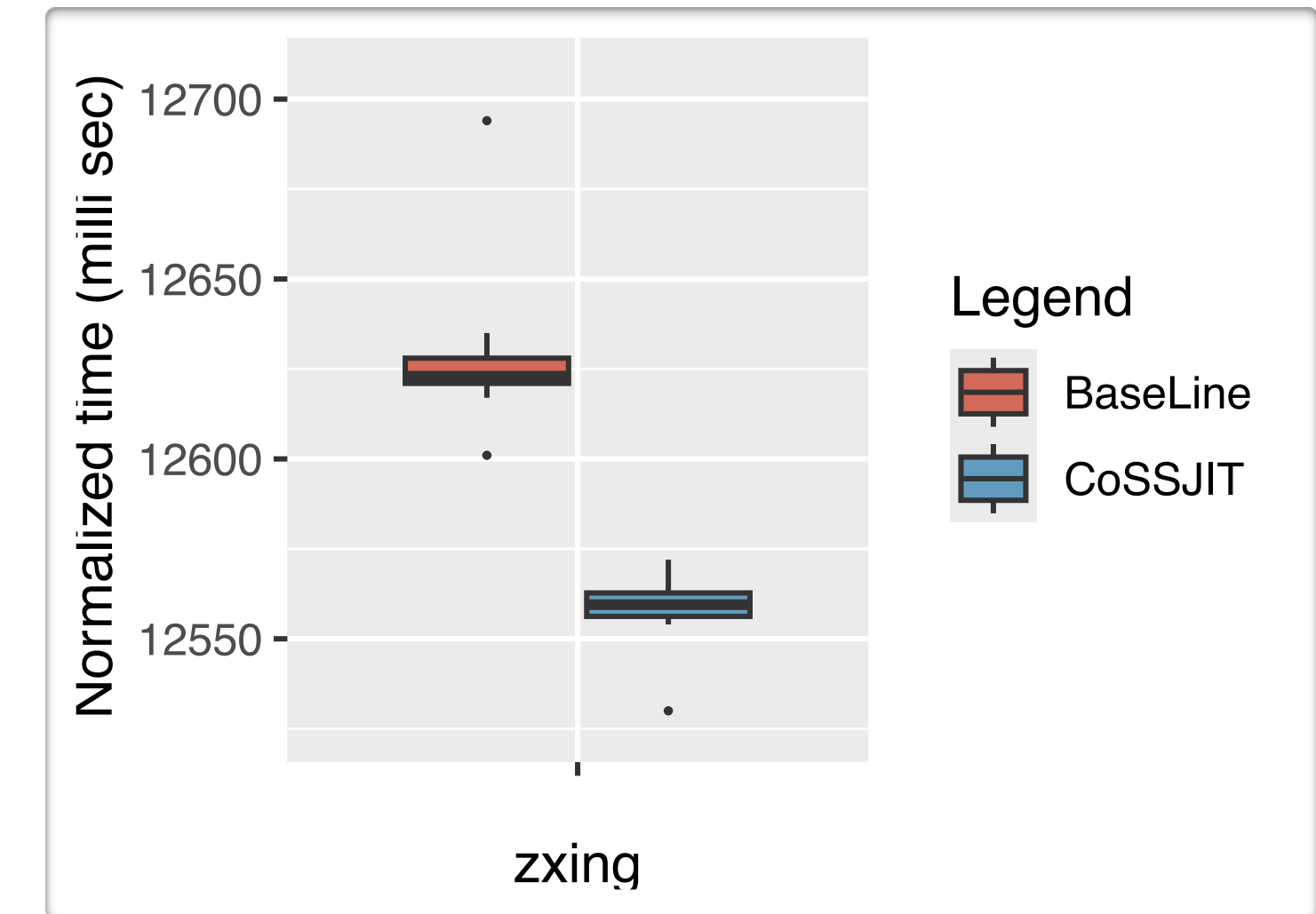
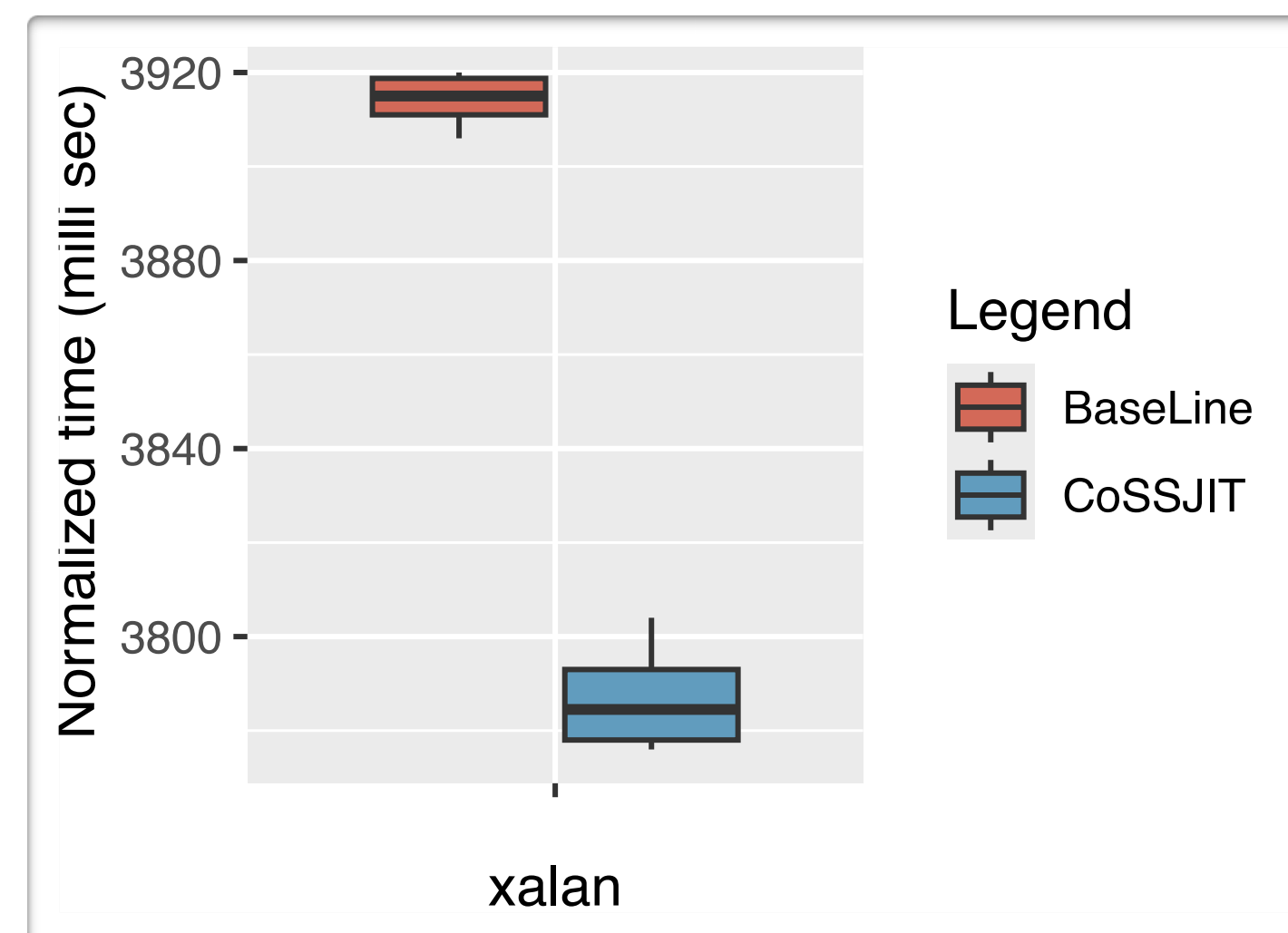
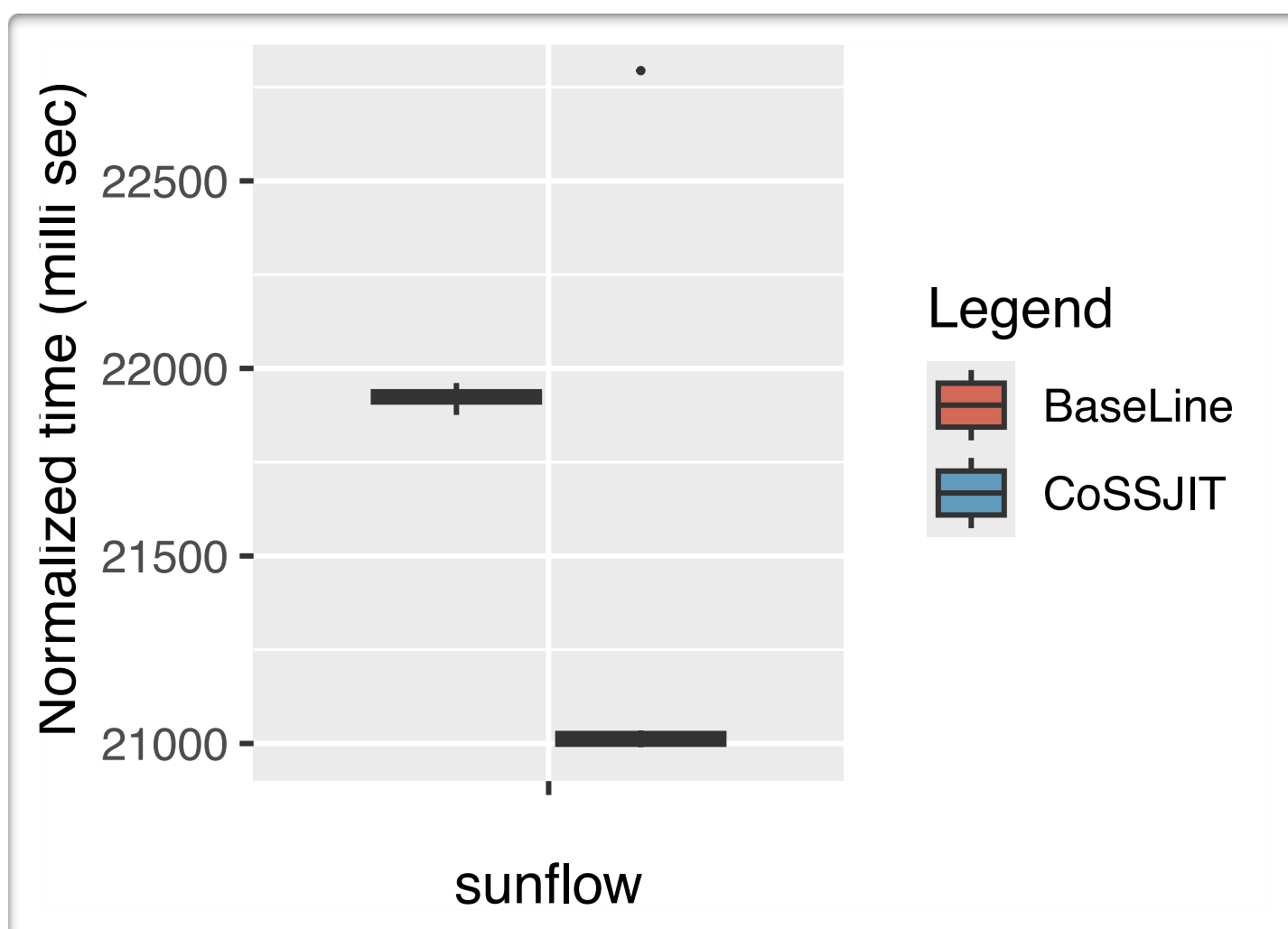
zxing

GC Reduction at X: 35% ↓

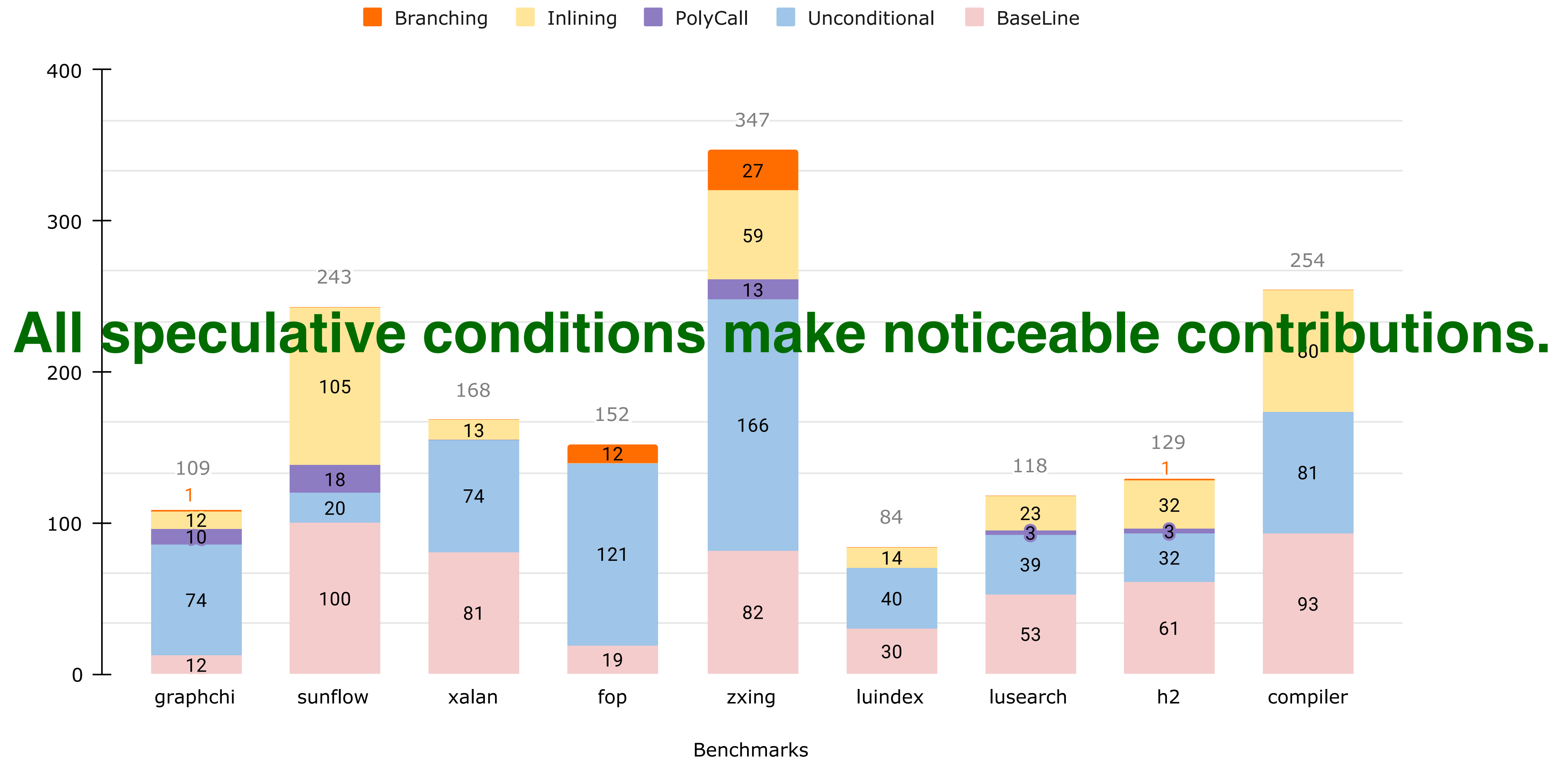
Performance Improvement



Performance Improvement: 6.7%↑

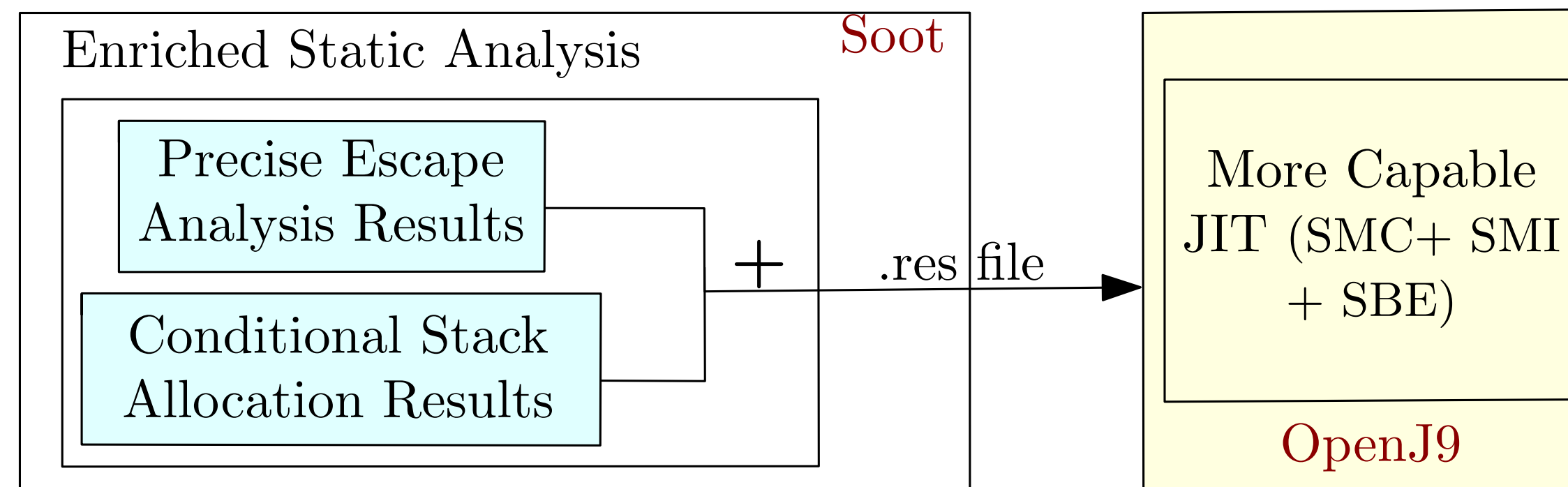


Contributions by different Speculative Conditions



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 strikes a balance between static analysis and JIT speculation, harnessing the best of both the worlds.

Thank You !!

CoSSJIT: Take Aways

- En
- Me



CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

ADITYA ANAND, Indian Institute of Technology Bombay, India

VIJAY SUNDARESAN, IBM Canada Lab, Canada

DARYL MAIER, IBM Canada Lab, Canada

MANAS THAKUR, Indian Institute of Technology Bombay, India

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

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

ecuc

CO

J

bal

ds.

T

- Ov
- sp

Thank You !!

CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers