



PROGRAM ANALYSIS FOR MANAGED RUNTIMES IN PRESENCE OF DYNAMIC FEATURES*



ADITYA ANAND[†] AND MANAS THAKUR[†]

OBJECTS ALLOCATION

- Objects in Java are allocated on the heap.
- Access time from heap is high. Garbage collection is an overhead.
- Optimization – Method-local stack allocation of objects.
- JIT Time Analysis – Highly imprecise.
- Static Analysis – Affected by dynamic features like DCL, HCR, Callbacks etc.

DYNAMIC CLASS LOADING & HOTCODE REPLACEMENT

```

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 */

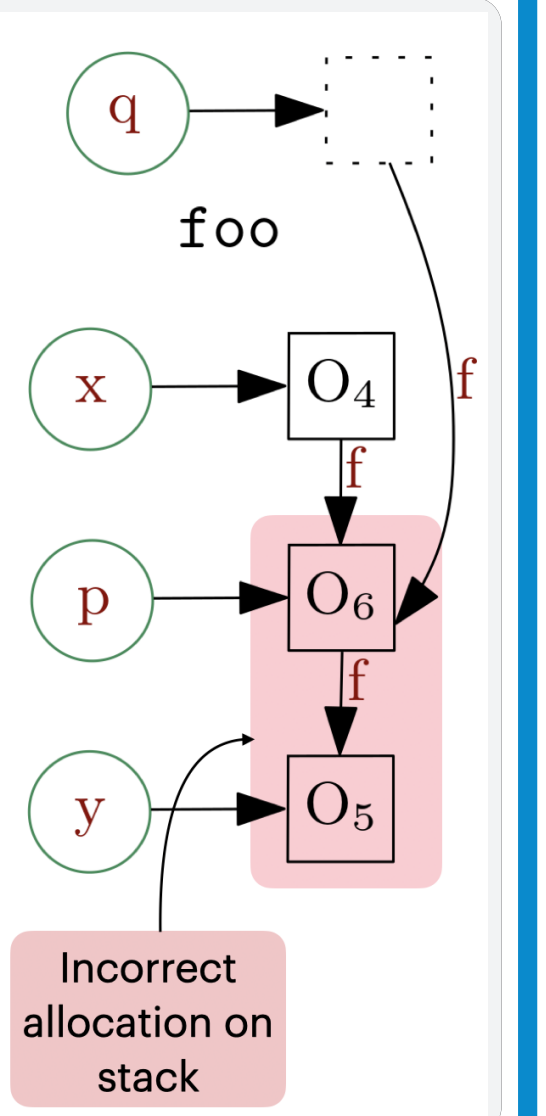
```

Dynamically loaded

```

1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.     A x = new A(); // O4
5.     A y = new A(); // O5
6.     x.f = new A(); // O6
7.     A p = x.f;
8.     bar(p, y);
9.     r.zar(p, q);
10.  } /* method foo */
11. void bar(A p1, A p2) {
12.   p1.f = p2;
13. } /* method bar */
14. void zar(A p, A q) {
15.   q.f = p;
16. } /* class A */
17. } /* class A */

```



CALLBACKS

```

App1:
1. public void foo1(A p1) {
2.   . . .
3.   A x = new A();
4.   this.bar(x);
5. }

App2:
1. public void foo2(A p1) {
2.   . . .
3.   A x = new A();
4.   this.bar(x);
5. }

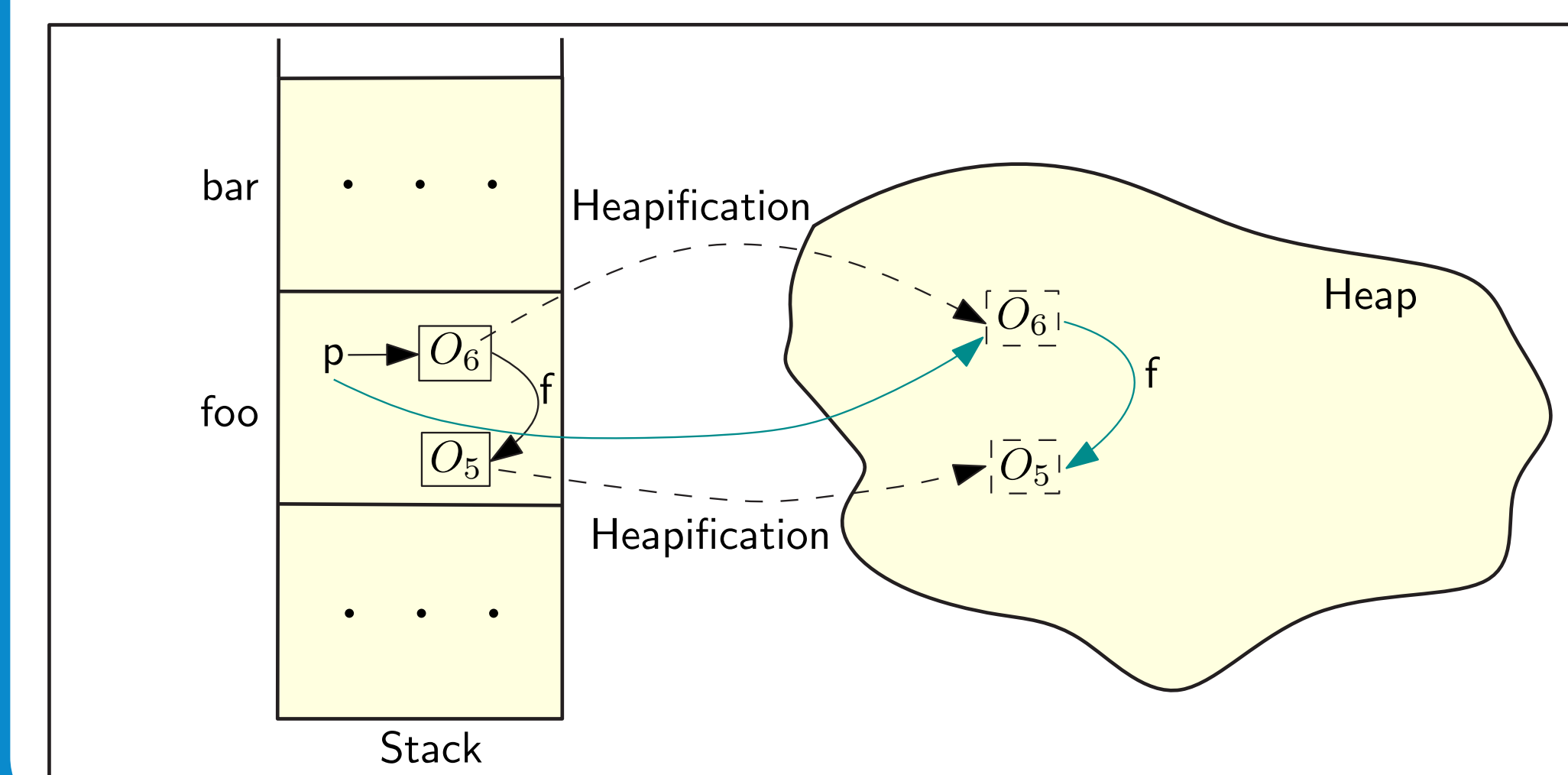
Lib:
1. public void lib() {
2.   A x = new A();
3.   global = x; // Escapes
4.   this.bar(x);
5. }

1. class A extends Library {
2.   @Override
3.   void bar(T p1) {
4.     A a = new A(); // O4
5.     p1.f = a;
6.   }
7. }

```

Incorrect stack allocation of O₄ in Lib.

HEAPIFICATION



HEAPIFICATION ALGO

```

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

```

STACK ORDERING

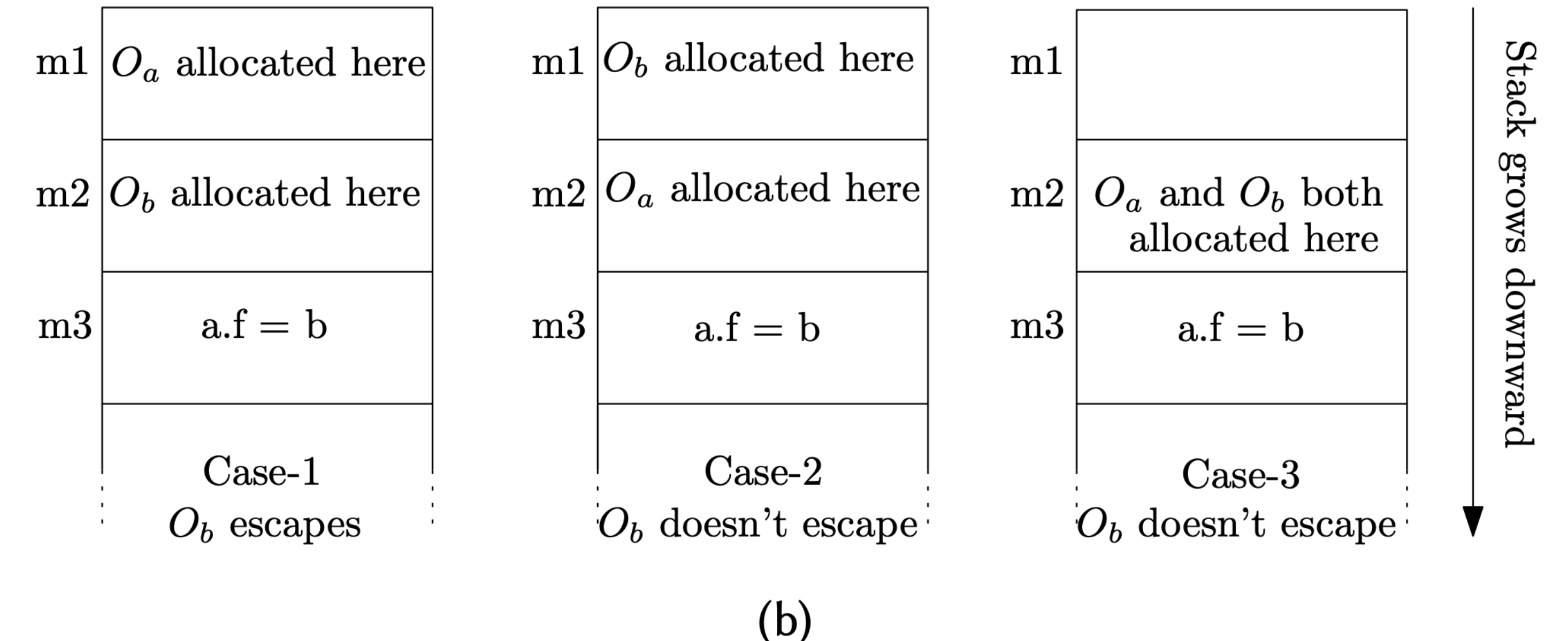
- Traversing stack-frames for parameters while checking for heapification is costly.
- Establish object ordering to enable address comparison for heapification checks, minimizing the need for frequent stack walks.
- Statically create a partial order of stack allocatable objects and use the stack-ordering in the VM to re-order the list of stack allocated objects.
- Reduces cost of heapification checks for the cases where objects doesn't escape.

IMPROVING EFFICIENCY BY STACK ORDERING

```

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

```



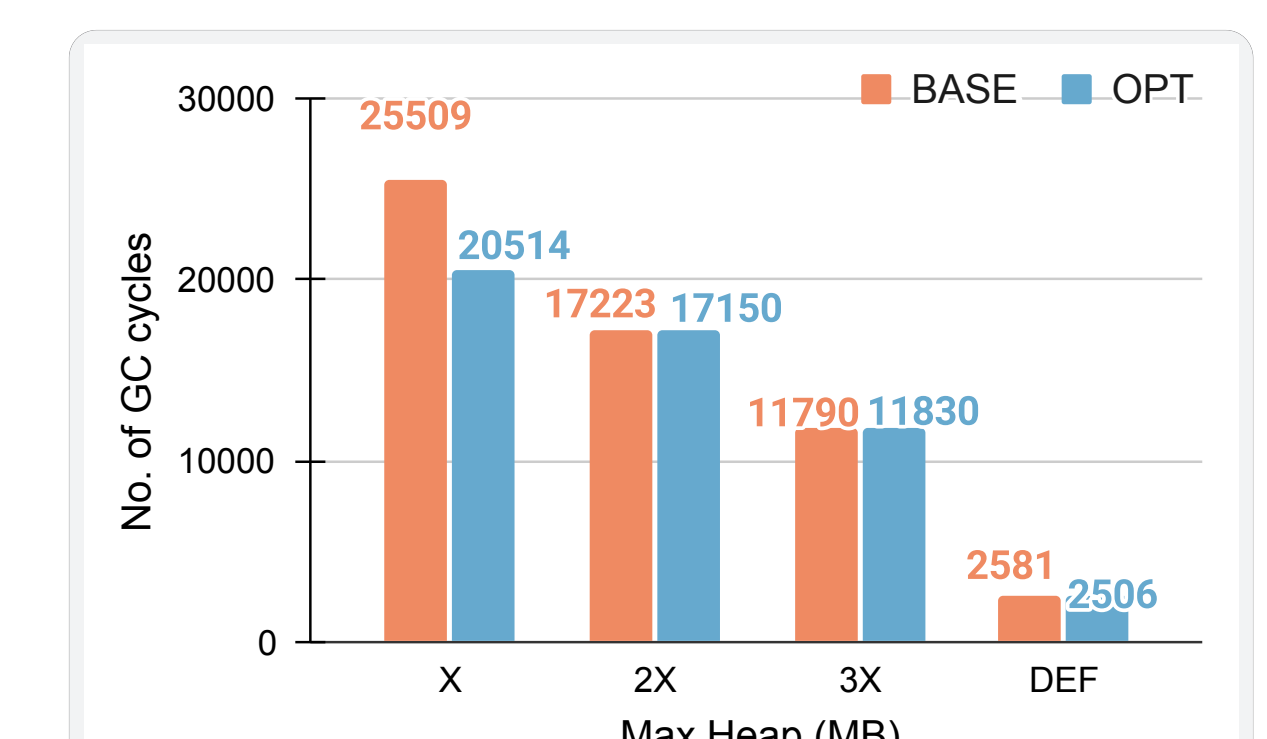
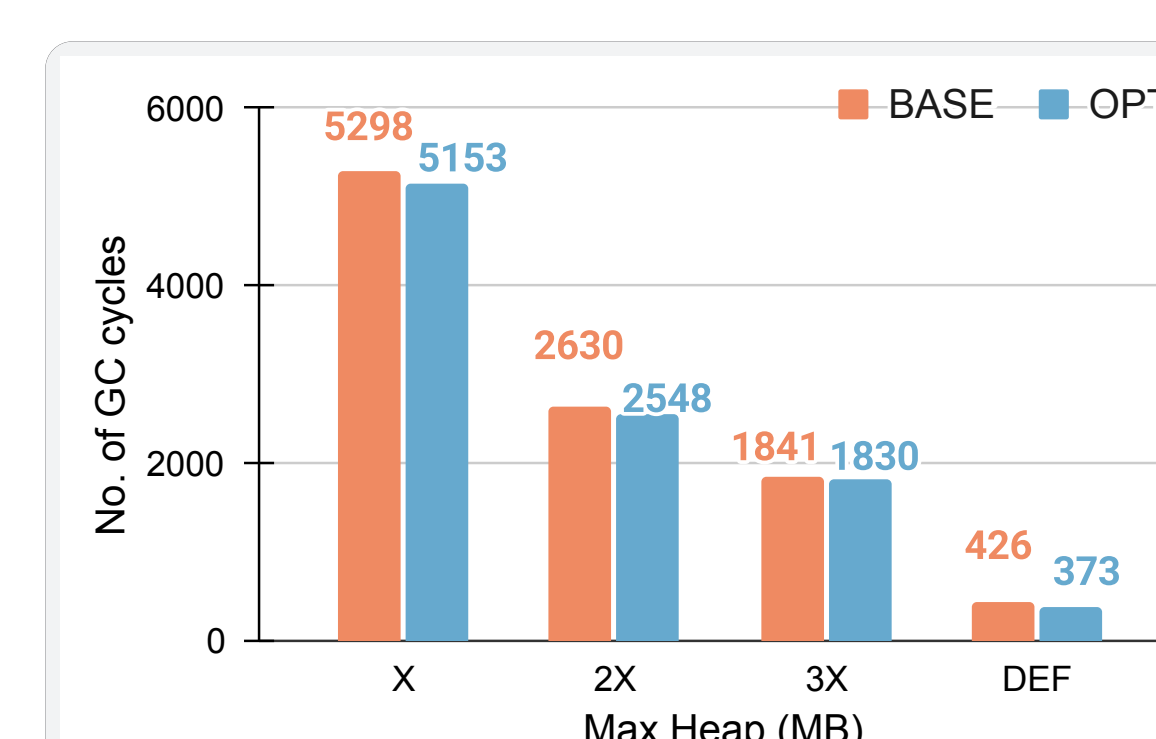
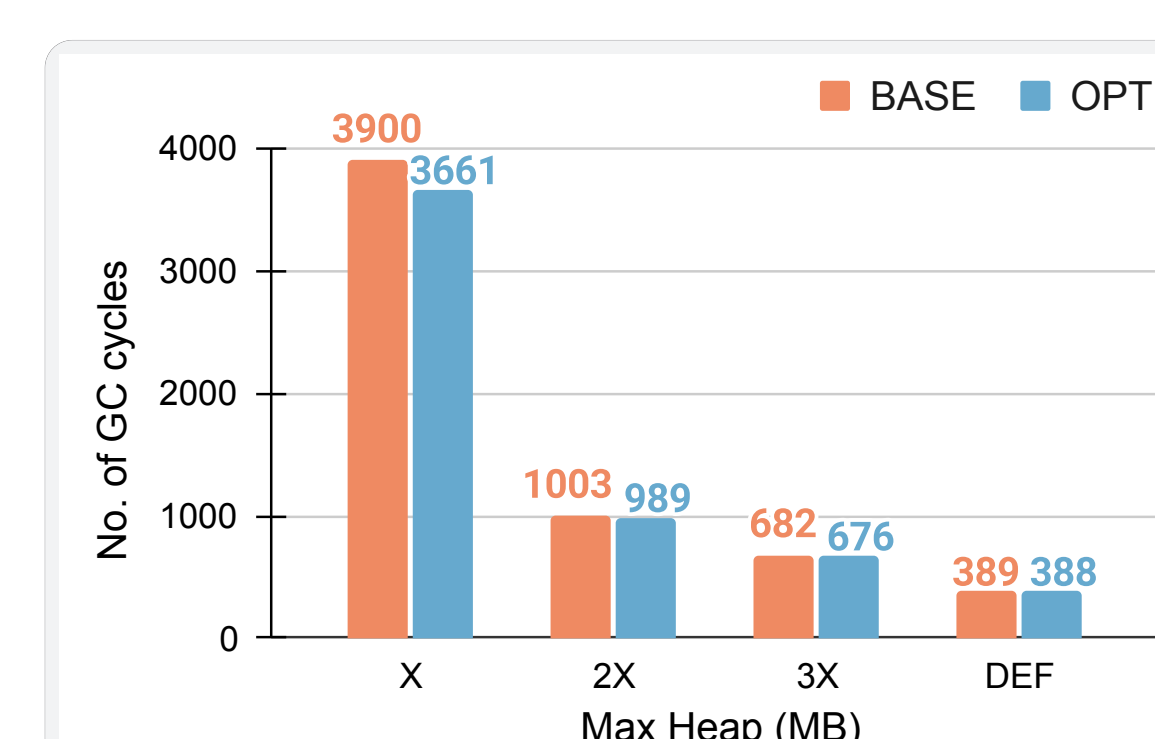
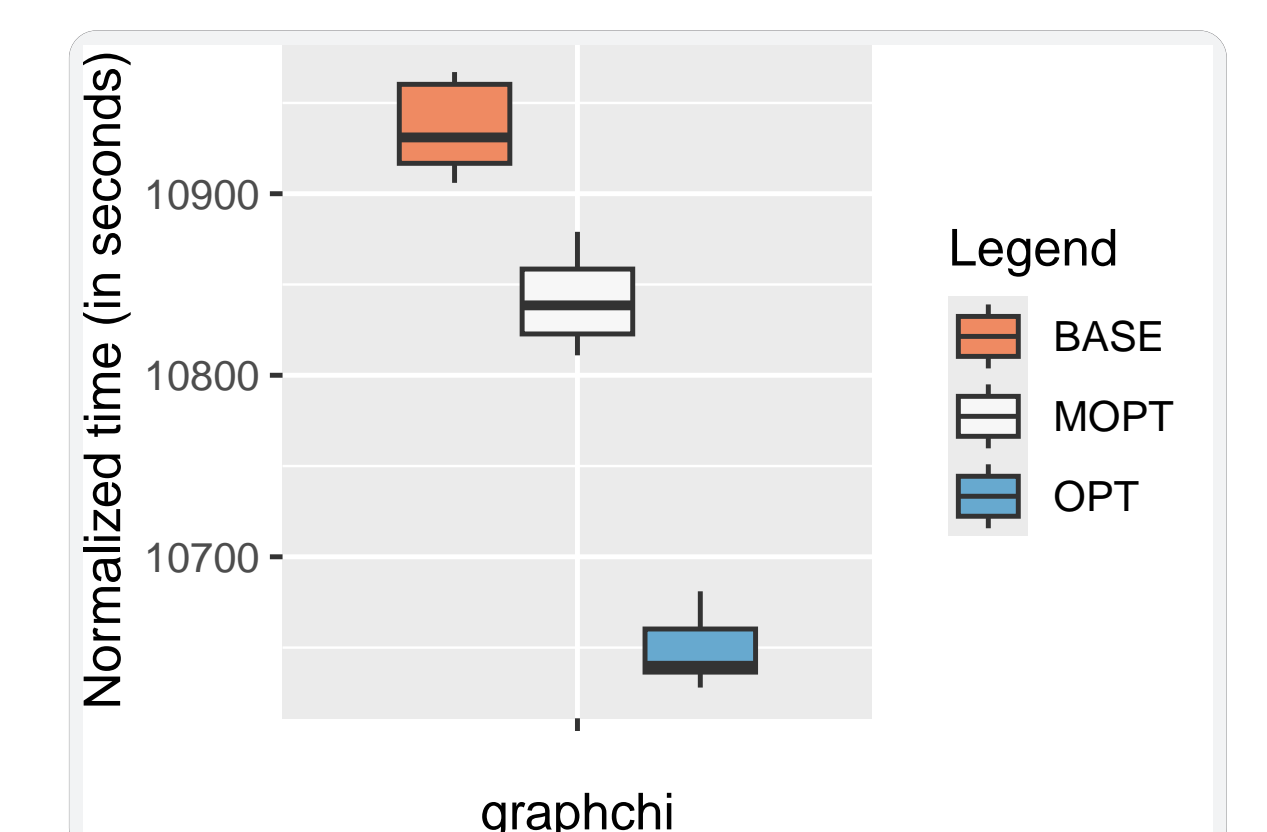
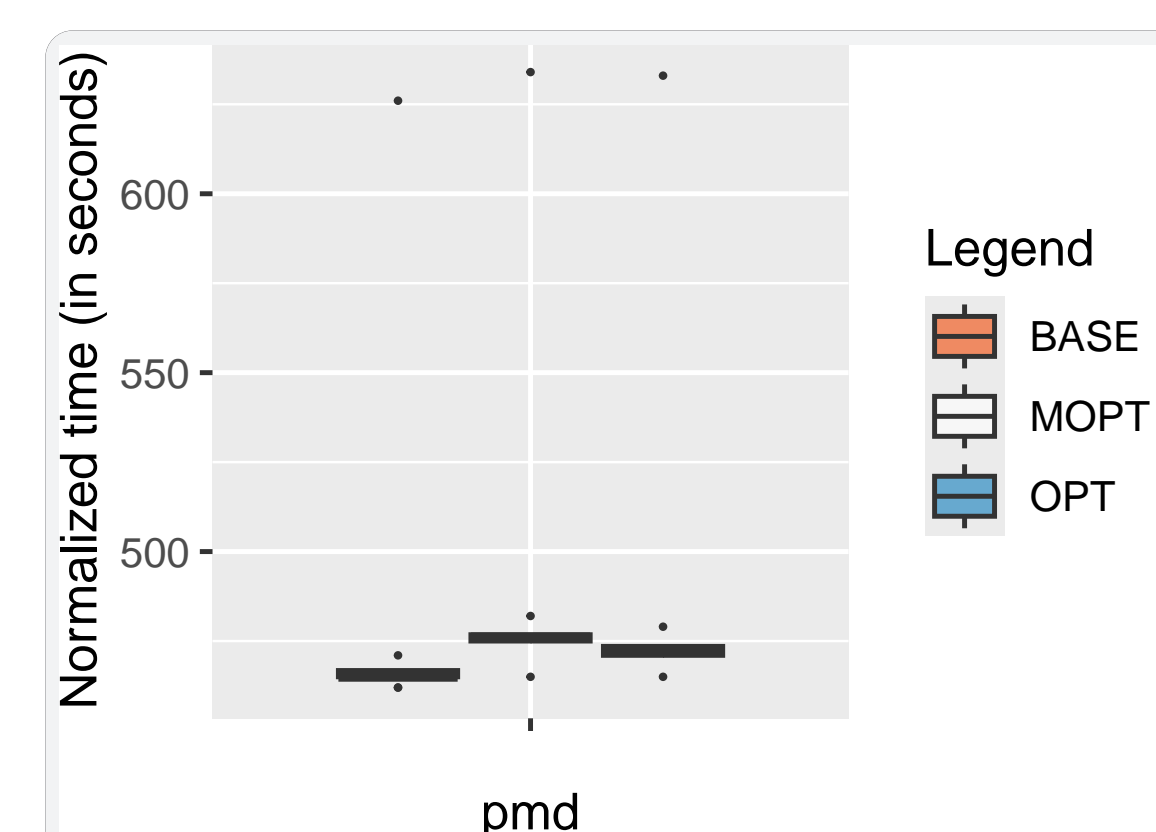
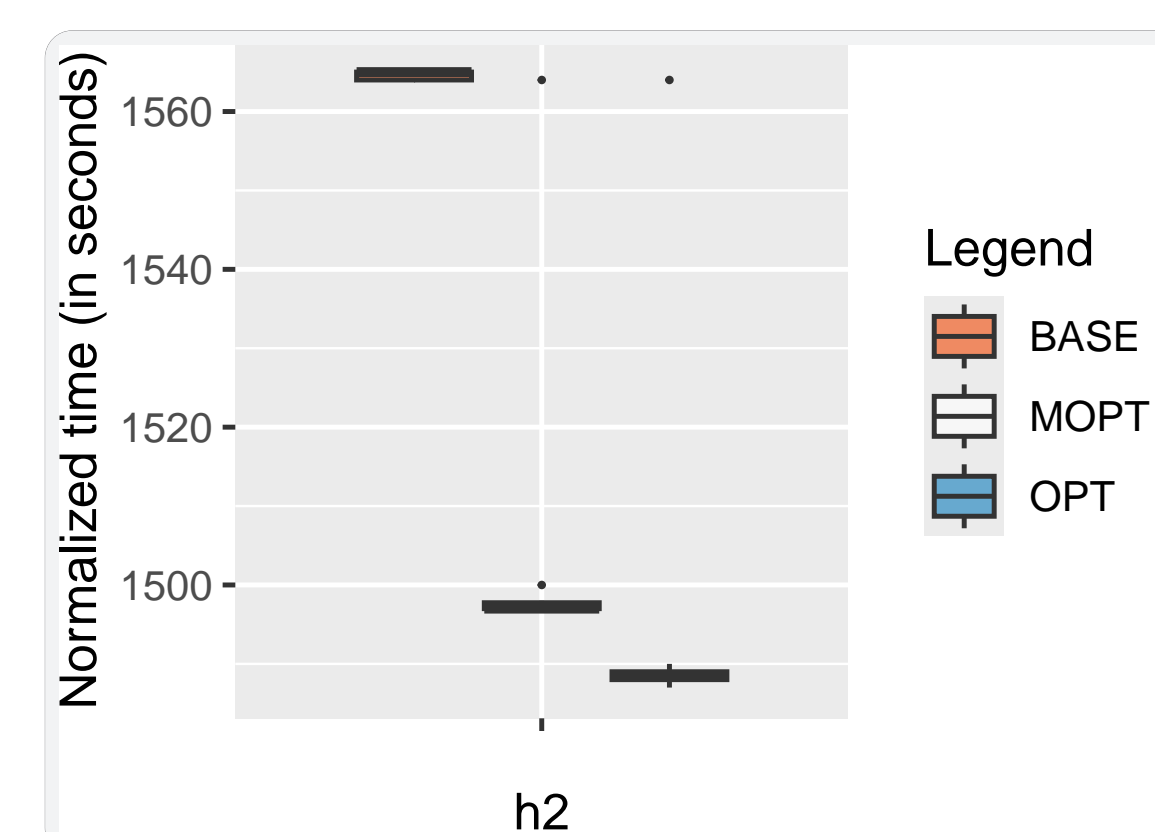
STACK ALLOCATION

h2			
BASE		OPT	
Stack-Objects	Stack-Bytes	Stack-Objects	Stack-Bytes
29M	0.5 GB	452M	10.8 GB

pmd			
BASE		OPT	
Stack-Objects	Stack-Bytes	Stack-Objects	Stack-Bytes
52M	1.3 GB	105M	2.4 GB

graphchi			
BASE		OPT	
Stack-Objects	Stack-Bytes	Stack-Objects	Stack-Bytes
0.0M	0 GB	506M	9.1 GB

PERFORMANCE IMPROVEMENT



CONCLUSION

- Proposed an idea to have dynamic checks for potential incorrect stack allocations, along with repairing memory layout by heapifying escaping objects and correcting their references.
- An efficient approach for performing heapification checks by ordering objects on the stack.

FUTURE WORK

- **Future Work:** Perform more aggressive stack-allocation & enable further optimizations in the JIT compilers.

RESEARCH AND INNOVATION SYMPOSIUM IN COMPUTING (RISC 2025), IIT BOMBAY

[†] Author addresses: {adityaanand, manas}@cse.iitb.ac.in. Department of CSE, IIT Bombay, Mumbai.
 * Anand et al. "Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes." In Proceedings of the ACM on Programming Languages (PLDI), Copenhagen, Denmark, June 24-28, 2024. URL: <https://doi.org/10.1145/3656389>

