Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes *

Aditya Anand† and Manas Thakur†

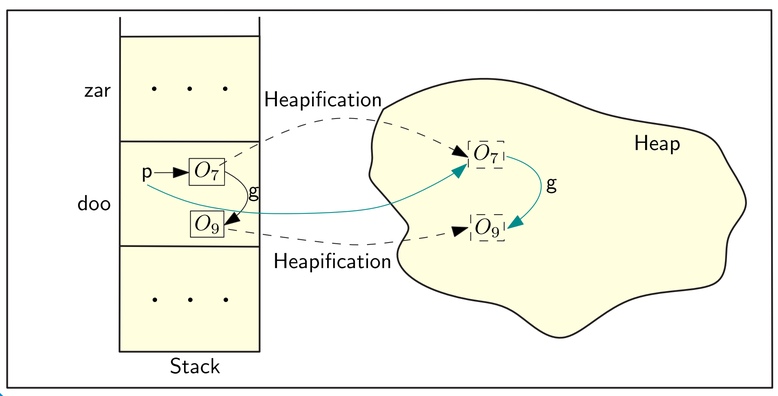
Objects Allocation

- Objects in Java are allocated on the heap.
- Automatic GC eases programmer burden and reduces memory bugs.
- Access time from heap is high. Garbage collection is an overhead.
- Optimization like method-local stack allocation is performed to improve the overall performance of runtime.

Dynamic Features

```
1  class A { B g; }
2  class D {
3     A f;
4     void doo(D q) {
5        D a = new D(); // O5
6        A b = new A(); // O6
7        a.f = new A(); // O7
8        A p = a.f;
9        a.g = new B(); // O9
10       zar(p, q);
11       bar(a, b);
12    } /* method doo */
13    void zar(A p, D q) {
14       q.f = new A(); // O14   [HCR: q.f = p;]
15       ...
16    } /* method zar */
17    void bar(D p1, A p2) {
18       p1.f = p2;
19    } /* method bar */
20 } /* class D */
```

Heapification



Heapification Alogrithm

Procedure HeapificationCheckAtStore(`srgReg`, `destReg`)
- **if** `srcReg` < `stackBaseReg` *OR* `srcReg` > `stackEndReg` **then**
  - No heapification required. /* Source object is outside stack bounds */
- **else**
  - /* Source object is present on the stack */
  - **if** `destReg` < `stackBaseReg` *OR* `destReg` > `stackEndReg` **then**
    - /* Destination object is outside stack bounds, hence source object escapes */
    - Heapify starting from source object.
  - **else**
    - /* Both source and destination objects are on the stack */
    - **if** `srcReg` >= `destReg` **then**
      - /* Source has been allocated before destination and hence does not escape */
      - No heapification required.
    - **else**
      - /* Destination object has been allocated in either the same frame or a deeper frame as compared to source object */
      - Perform stack-walk and heapify if needed.

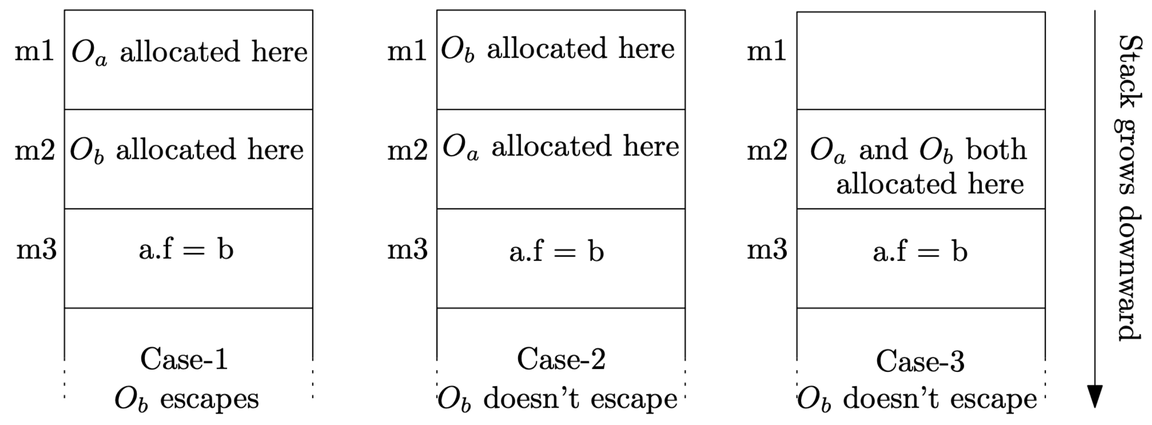
Stack Ordering

- 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.

Improving efficiency by Stack Ordering

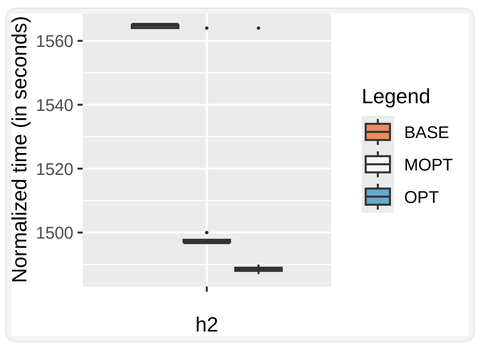
```
class T {
  T f;
  void m1() {m2(...);}
  void m2() {m3(...);}
  void m3(T a, T b) {
      a.f = b;
  }
}
```

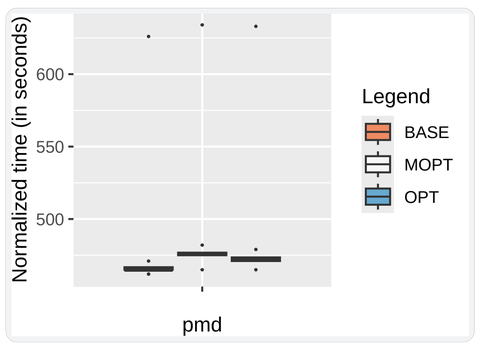
(a)

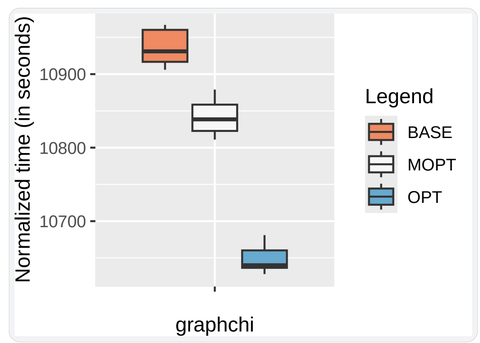


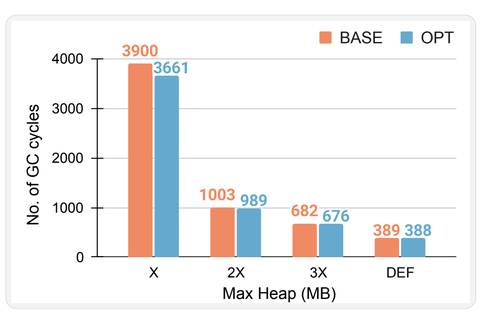
(b)

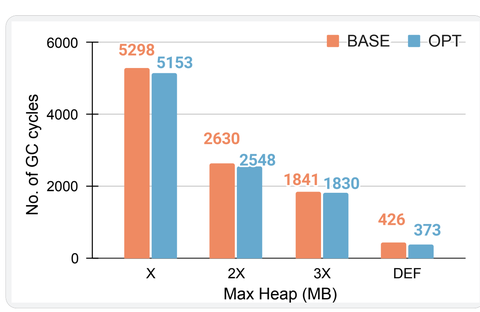
Performance Improvement

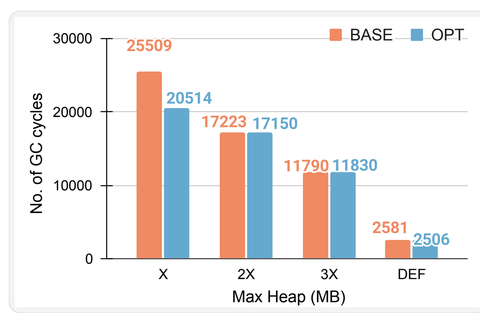












Stack Allocation

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

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

Academic Research and Careers for Students (ARCS 2025), Coimbatore, India.

† Author adresses: {adityaanand, manas}@cse.iitb.ac.in. PLATO Lab, 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