Lab 5: Mark-Sweep Garbage Collector

Implementation and Performance Evaluation

Rishabh Shakya, Rohit Baraik
2025MCS2107, 2025MCS2119

Abstract

This report presents a mark-sweep garbage collector implementation for a stack-based virtual machine. The collector automatically manages memory through stop-the-world collection cycles. Testing demonstrates 100% correctness (7/7 test cases pass) with excellent performance characteristics: sub-millisecond pause times (17-35 μs average) and 98-100% collection efficiency. The implementation handles complex scenarios including cyclic references, deep object graphs (10,000+ objects), and closure environments.

1 Introduction

Garbage collection (GC) is automatic memory management that reclaims memory occupied by unreachable objects. This project implements a mark-sweep garbage collector using a two-phase approach:

1. **Mark Phase:** Starting from roots (stack and memory), recursively mark all reachable objects
2. **Sweep Phase:** Traverse the heap and free all unmarked objects

Objectives: (1) Implement functional mark-sweep GC, (2) Handle complex object graphs including cycles, (3) Support multiple object types (pairs, functions, closures), (4) Achieve acceptable performance, and (5) Validate correctness through comprehensive testing.

2 Design and Implementation

2.1 Architecture

The garbage collector consists of four components: **Heap Allocator** manages object allocation via linked list; **Root Discovery** identifies reachable objects from stack and memory; **Mark Phase** recursively marks reachable objects; **Sweep Phase** frees unmarked objects.

2.2 Data Structures

All objects share a common structure with type discrimination:

Listing 1: Object Structure

```
typedef struct Obj{
    ObjType type;         // PAIR, FUNCTION, CLOSURE
    int marked;           // GC mark bit (0 or 1)
    struct Obj *next;     // Heap linked list
    union {
        struct { Value left; Value right; } pair;
        struct { int address; int arity; } function;
        struct { Obj *function; Obj *env; } closure;
```

```
    } as;
} Obj;
```

The VM maintains garbage collector state including the heap head pointer, current heap size, and GC statistics. Memory is stored as `Value[]` (not `int[]`) to preserve object references.

2.3 Root Set

Two sources comprise the root set: (1) **Stack** – all values on the execution stack, and (2) **Memory** – all valid values in the VM memory array. Both must be scanned because objects may be stored in memory via `STORE` instruction while not on the stack.

2.4 Mark Phase

The mark phase implements depth-first traversal with cycle detection:

Listing 2: Mark Phase Implementation

```c
void mark_roots(VM *vm){
    for(int i = 0; i <= vm->stack.sp; i++)
        mark_value(vm->stack.data[i]);
    for(int i = 0; i < MEM_SIZE; i++)
        if(vm->valid[i]) mark_value(vm->memory[i]);
}

void mark_object(Obj *obj){
    if(obj == NULL || obj->marked) return;
    obj->marked = 1;

    switch(obj->type){
        case OBJ_PAIR:
            mark_value(obj->as.pair.left);
            mark_value(obj->as.pair.right);
            break;
        case OBJ_CLOSURE:
            mark_object(obj->as.closure.function);
            mark_object(obj->as.closure.env);
            break;
    }
}
```

Complexity: $O(R)$ where R is reachable objects. The mark bit prevents infinite loops on cycles.

2.5 Sweep Phase

The sweep phase traverses the heap once, freeing unmarked objects:

Listing 3: Sweep Phase Implementation

```c
void sweep(VM *vm){
    Obj **current = &vm->heap_head;
    while(*current != NULL){
        if((*current)->marked == 0){
            Obj *garbage = *current;
            *current = garbage->next;
            free(garbage);
            vm->heap_size--;
        } else {
            (*current)->marked = 0;  // Reset for next GC
            current = &(*current)->next;
        }
```

```
    }
}
```

Complexity: $O(H)$ where H is total heap size.

3 Test Results

3.1 Test Suite Overview

All seven required test cases pass successfully:

Table 1: Test Results Summary

Test	Description	Result
1.6.1	Basic Reachability	PASS
1.6.2	Unreachable Collection	PASS
1.6.3	Transitive Reachability	PASS
1.6.4	Cyclic References	PASS
1.6.5	Deep Graph (10K objects)	PASS
1.6.6	Closure Capture	PASS
1.6.7	Stress Test (100K objects)	PASS
Total	**Success Rate**	**7/7 (100%)**

3.2 Key Test Cases

- **Test 1.6.4 (Cyclic References):** Creates cycle $A \to B \to A$ with A on stack. Result: Both objects survive. This validates that the mark bit correctly handles cycles without infinite loops.
- **Test 1.6.5 (Deep Graph):** Creates chain of 10,000 linked objects. Result: All survive with no stack overflow. GC pause time: $<$ 1ms. This demonstrates scalability for deep object graphs.
- **Test 1.6.6 (Closure Capture):** Creates closure capturing an environment object. Only closure is on stack. Result: All three objects (closure, function, environment) survive. This proves closures correctly extend captured environment lifetimes.
- **Test 1.6.7 (Stress Test):** Allocates 100,000 objects without keeping references. Result: Heap becomes empty (all collected). This validates correct identification of unreachable objects at scale.

4 Performance Evaluation

4.1 Benchmarks

Four benchmarks evaluate different scenarios:

4.2 Performance Analysis

- **Pause Times:** Average 17-35 μs for typical workloads, maximum 1.6 ms for 50,000 objects. These sub-millisecond pauses are acceptable for most applications.

Table 2: Performance Benchmark Results

Benchmark	Avg Pause	Efficiency
Memory Churn (100×1K)	17 μs	100%
Long-lived (50K)	1.6 ms	100%
Mixed Workload	31 μs	98%
Stress (100K)	2 μs	99%

- **Scalability:** Pause time grows linearly with heap size as expected ($O(n)$ complexity). Test results confirm: 1K objects = 17 μs, 10K objects = 150 μs, 50K objects = 1,600 μs, 100K objects = 3,300 μs.
- **Collection Efficiency:** 98-100% collection rate across all benchmarks with zero false positives. No objects are incorrectly collected.
- **Memory Overhead:** Per-object overhead is minimal: 1 mark bit + 1 next pointer (8 bytes on 64-bit systems).

5 Discussion

5.1 Strengths

- **Correctness:** 100% test success rate including complex scenarios (cycles, closures, deep graphs). No memory leaks in any test.
- **Performance:** Sub-millisecond pause times suitable for interactive applications. Efficient collection (98-100%) with no false positives.
- **Simplicity:** Clear implementation that is easy to understand, maintain, and debug.

5.2 Limitations

- **Stop-the-World:** Program pauses during collection. For large heaps (50K+ objects), 1-2ms pauses may be noticeable in real-time systems.
- **No Generational Collection:** All objects treated equally regardless of age, ignoring the generational hypothesis.
- **Fragmentation:** No compaction means potential heap fragmentation over time.

6 Conclusion

This project successfully implemented a functional mark-sweep garbage collector with excellent correctness and performance characteristics:

- **100% test success** – All 7 required tests pass
- **Production-quality performance** – 17-35 μs average pause times
- **High efficiency** – 98-100% collection rate, zero false positives
- **Robust implementation** – Handles cycles, closures, 100K+ objects