# Java Memory Management Lab Report

The Java Virtual Machine (JVM) is an abstract computing machine that enables Java programs to run on any device or operating system.

## 1. JVM Internals

The Java Virtual Machine (JVM) is composed of several key components:

1. Class Loader: Loads, links, and initializes Java classes.
2. Runtime Data Areas:
   - Heap: Where objects are allocated.
   - Method Area: Stores class structures and static variables.
   - Java Stacks: One per thread, stores local variables and partial results.
   - PC Registers: Stores the current execution point for each thread.
   - Native Method Stacks: Used for native method invocations.
3. Execution Engine: Executes bytecode.
4. Native Interface: Allows interaction with native code.

The Heap is the primary focus for garbage collection and memory management.

## 2. Garbage Collector Tuning

Java offers several garbage collectors, each with different characteristics:

1. Serial GC (-XX:+UseSerialGC): A simple, single-threaded collector suitable for small applications.
2. Parallel GC (-XX:+UseParallelGC): Uses multiple threads for faster collection.
3. G1GC (Garbage First) (-XX:+UseG1GC): Designed for large heaps with predictable pause times.
4. ZGC (-XX:+UseZGC): Designed for very large heaps with low latency.

Let's compare ParallelGC and G1GC:

```
// Serial GC
java -XX:+UseSerialGC MemoryProfileDemo.java

// ParallelGC
java -XX:+UseParallelGC MemoryProfileDemo.java

// G1GC
java -XX:+UseG1GC MemoryProfileDemo.java

// ZGC
java -XX:+UseZGC MemoryProfileDemo.java
```

To measure GC performance, use verbose GC logging:

```
java -Xlog:gc*:file=gc.log MemoryProfileDemo.java
```

## 3. Memory Profiling

To profile memory usage, we can use tools like VisualVM or JConsole. Here's a simple program to demonstrate memory allocation:

```java
import java.util.ArrayList;
import java.util.List;

public class MemoryProfileDemo {
    private static final int MB = 1024 * 1024;

    public static void main(String[] args) throws InterruptedException {
        List<byte[]> list = new ArrayList<>();
        System.out.println("Starting memory allocation...");

        for (int i = 0; i < 1000; i++) {
            byte[] b = new byte[MB]; // Allocate 1MB
            list.add(b);
            System.out.println("Allocated " + (i + 1) + "MB");

            if (i % 10 == 0) {
                // Clear half of the list every 10 iterations
                for (int j = 0; j < list.size() / 2; j++) {
                    list.remove(0);
                }
                System.out.println("Cleared half of the allocations");
            }

            Thread.sleep(50); // Sleep to slow down allocation
        }

        System.out.println("Memory allocation completed");
    }
}
```

## 4. Memory Optimization

Here are some techniques to optimize memory usage:

1. Object Pooling: Object Reuse

```java
public class ObjectPool<T> {
    private List<T> pool;
    private Supplier<T> creator;

    public ObjectPool(Supplier<T> creator, int initialSize) {
```

```java
        this.creator = creator;
        pool = new ArrayList<>(initialSize);
        for (int i = 0; i < initialSize; i++) {
            pool.add(creator.get());
        }
    }

    public T acquire() {
        if (pool.isEmpty()) {
            return creator.get();
        }
        return pool.remove(pool.size() - 1);
    }

    public void release(T obj) {
        pool.add(obj);
    }
}
```

2. Use Primitive Types:

```java
// Instead of
Integer[] array = new Integer[1000];

// Use
int[] array = new int[1000];
```

3. Avoid Unnecessary Object Creation:

```java
// Instead of
String result = new String("Hello");

// Use
String result = "Hello";
```

4. Use StringBuilder for String Concatenation:

```java
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append("Item ").append(i).append(", ");
}
String result = sb.toString();
```

## Garbage Collector Performance Comparison

## Summary of Findings

1. G1GC (Garbage-First Garbage Collector)

- Initial heap capacity: 384M
- Maximum heap capacity: 6116M
- Number of GC cycles: 100
- Average GC pause time: ~2.4ms
- Longest GC pause: 43.095ms (GC 29)
- Heap usage after final GC: 519M->159M

Key observations:

- G1GC shows very short pause times, mostly under 10ms.
- It efficiently reclaims memory, often reducing heap usage by 50% or more.
- G1GC performs concurrent marking and cleaning, minimizing stop-the-world pauses.

2. Parallel GC

- Initial heap capacity: 384M
- Maximum heap capacity: 6116M
- Number of GC cycles: 45
- Average young generation GC pause time: ~10-20ms
- Average full GC pause time: ~10-15ms
- Heap usage after final GC: 560M->131M

Key observations:

- Parallel GC has longer pause times compared to G1GC but still maintains relatively low latency.
- It shows efficient memory reclamation, especially during full GCs.
- The parallel nature of this collector allows it to utilize multiple CPU cores effectively.

3. Serial GC

- Initial heap capacity: 384M
- Maximum heap capacity: 6116M
- Number of GC cycles: 67
- Average young generation GC pause time: ~10-20ms
- Average full GC pause time: ~15-30ms
- Heap usage after final GC: 508M->122M

Key observations:

- Serial GC has longer pause times compared to both G1GC and Parallel GC.
- It shows good memory reclamation efficiency, similar to Parallel GC.
- Being single-threaded, it may not utilize multiple CPU cores as effectively as Parallel GC.

4. ZGC (Z Garbage Collector)

- Initial heap capacity: 384M
- Maximum heap capacity: 6116M
- Number of GC cycles: 19

- Average GC pause time: ~24.2ms
- Longest GC pause: 49.891ms
- Heap usage after final GC: 508M->122M

Key observations:

- ZGC shows relatively low pause times, though not as low as G1GC.
- It demonstrates excellent memory reclamation, often reducing heap usage by more than 75%.
- ZGC employs concurrent processing, which helps in reducing pause times.

## Comparison and Analysis

1. Pause Times: G1GC < ZGC < Parallel GC < Serial GC G1GC shows the lowest pause times, making it suitable for applications requiring low latency.

2. Memory Reclamation Efficiency: All collectors show good memory reclamation, with ZGC and G1GC often showing slightly better results.

3. Scalability: G1GC and ZGC are designed for large heaps and show good scalability. Parallel GC scales well with multiple CPU cores. Serial GC may not be ideal for multi-core systems or large heaps.

4. Concurrent Processing: G1GC and ZGC employ concurrent processing, which helps reduce pause times. Parallel and Serial GC rely more on stop-the-world pauses.

5. Complexity: G1GC and ZGC are more complex collectors, which can lead to less predictable behavior in some cases. Parallel and Serial GC are simpler and may be more predictable.

## Recommendations

1. For applications requiring low latency and running on multi-core systems with large heaps, G1GC or ZGC would be the best choices.

2. For batch processing or applications where throughput is more important than latency, Parallel GC could be a good option.

3. For small applications or those running on systems with limited resources, Serial GC might be sufficient.

4. G1GC seems to offer the best overall performance in terms of pause times and efficiency, making it a good default choice for many applications.

## Memory Management Best Practices

1. Properly close resources using try-with-resources.
2. Use weak references for caching to allow GC when memory is low.
3. Avoid finalizers and prefer try-with-resources for cleanup.
4. Size your heap appropriately for your application's needs.
5. Monitor and tune GC performance regularly.
6. Use tools like VisualVM, JConsole, or JProfiler for memory analysis.
7. Consider using value types (JDK 15+) for small, immutable objects.

By implementing these techniques and following best practices, you can significantly improve your application's memory usage and performance.