Datta Ksheeraj - 21CS30037

Assignment-1 part-2

SUB-PART-1

Using the callgrind outputs or call graphs to find the percentage of instructions consumed by GC for the different GC configurations.

Full GC

Total Instructions executed: 1,166,851,516

Function calls: (Involving Garbage Collector)

a. Propagatemark - 54,623,247 (4.68%)
b. Reallymarkobject - 16,254,143 (1.39%)
c. Singlestep - 15,176,089 (1.30%)
d. luaC_newobj - 12,626,649 (1.08%)
e. Sweepstep - 11,463,439 (0.98%)
f. luaC_step - 4,250,149 (0.36%)
g. Freeobj - 9,020,480 (0.77%)

Instructions executed under Lua gc: 123,414,196
Percentage of Lua gc instructions: 10.5766%

Explanation: Full GC typically runs less frequently but does a more comprehensive sweep of all objects when it does run. This can lead to a higher percentage of instructions being consumed during the GC cycles, as it processes all objects in the memory space. The fact that Full GC consumes 10.58% of the total instructions aligns with the expectation that it performs a deep clean of memory, albeit less often

Incremental GC

Total Instructions executed: 1,247,745,738

Function calls: (Involving Garbage Collector)

a. Propagatemark -78,397,967 (6.28%)
b. Reallymarkobject - 23,317,190 (1.87%)

c. Singlestep - 21,829,392 (1.75%)
d. luaC_newobj - 12,626,649 (1.01%)
e. Sweepstep - 16,421,800 (0.98%)
f. luaC_step - 6,111,107 (0.49%)
g. Freeobj - 9,020,480 (0.72%)

Instructions executed under Lua gc: 167,724,585
Percentage of Lua gc instructions: 13.442%

Explanation: Incremental GC spreads out its work over time, performing small portions of the garbage collection task more frequently. This results in a higher percentage of instructions being consumed by the GC since it's constantly working to manage memory. The 13.44% indicates that while each individual GC cycle is less costly, the cumulative effect of continuous collection results in a higher overall instruction count

Generational GC

Total Instructions executed: 1,185,439,348

Function calls: (Involving Garbage Collector)

a. Propagatemark -57,701,542 (4.87%)
b. Reallymarkobject - 17,165,691 (1.45%)
c. luaC_newobj - 12,003,052 (1.01%)
d. Sweepstep - 13,587,086 (1.16%)
e. Freeobj - 9,020,480 (0.76%)

Instructions executed under Lua gc: 119,772,363
Percentage of Lua gc instructions: 9.23%

Explanation: Generational GC focuses on collecting younger objects more frequently while leaving older objects to be collected less often. This strategy can lead to more efficient memory management with fewer instructions consumed by GC (9.23%). The lower percentage suggests that the generational approach effectively reduces the overhead of garbage collection by focusing efforts on objects most likely to be garbage

Summary:

- **Full GC** performs a thorough collection with moderate efficiency, consuming 10.58% of instructions.
- **Incremental GC** is more instruction-intensive due to its constant memory management, consuming the highest percentage at 13.44%.

- **Generational GC** is the most efficient in this test, consuming the least instructions at 9.23%.

SUB-PART-2

m = 100 n = 100

FULL GC

Total: 107,946,707
Percentage: 13.19

INCREMENTAL GC

total : 114,449,555
Percentage: 11.87

GENERATIONAL GC

Total: 111,612,832
Percentage: 9.29%

m = 500 n = 100

FULL GC

Total: 585,420,014
Percentage: 13.22

INCREMENTAL GC

total : 590,307,083
Percentage: 12.46

GENERATIONAL GC

Total: 583,297,604
Percentage: 9.85%

m = 5000 n = 100

FULL GC

Total: 6,050,531,900
Percentage: 10.93

INCREMENTAL GC

total : 6,076,478,438
Percentage: 10.29

GENERATIONAL GC

Total: 6,006,044,541
Percentage: 9.57%

Trend Analysis Across Different Matrix Sizes (Including m = 1000, n = 100):

1. Small Matrix (m = 100, n = 100):

- **Full GC (13.19%)** consumes the most instructions as a percentage, followed by **Incremental GC (11.87%)**, and **Generational GC (9.29%)**.
- **Explanation**: For smaller matrices, the overhead of initiating a full garbage collection cycle is relatively high. Incremental GC spreads its workload, reducing overhead, and Generational GC is the most efficient, focusing on short-lived objects.

2. Medium Matrix (m = 500, n = 100):

- The percentage of instructions consumed by GC remains fairly consistent: **Full GC (13.22%)**, **Incremental GC (12.46%)**, and **Generational GC (9.85%)**.
- **Explanation**: As the matrix size increases, the total memory allocation grows, leading to more frequent or extensive garbage collection cycles. The percentages remain stable, with Generational GC still showing the best efficiency.

3. Large Matrix (m = 1000, n = 100):

- **Full GC (10.58%)**, **Incremental GC (13.44%)**, and **Generational GC (9.23%)**.
- **Explanation**: With an even larger matrix, the workload grows, and the percentage of instructions consumed by Full GC slightly decreases. Incremental GC, however, sees a higher percentage, possibly due to the increased frequency of its smaller GC cycles. Generational GC remains the most efficient, continuing to minimize overhead.

4. Very Large Matrix (m = 5000, n = 100):

- **Full GC (10.93%)**, **Incremental GC (10.29%)**, and **Generational GC (9.57%)**.
- **Explanation**: With the largest matrix, the total number of instructions executed increases significantly, reducing the relative impact of GC instructions. The trend of decreasing GC instruction percentage continues, with Generational GC maintaining its efficiency.

General Trends:

1. **Full GC**: High overhead in small to medium matrices, but its percentage of consumed instructions decreases as the matrix size increases, reflecting its less frequent but more comprehensive sweeps.
2. **Incremental GC**: Starts with a moderate overhead in small matrices, increases for medium to large matrices, and slightly decreases again for very large matrices. The frequent small collections of Incremental GC become more impactful as memory demands grow but taper off slightly as the workload becomes extensive.
3. **Generational GC**: Consistently shows the lowest percentage of GC instructions across all matrix sizes. It efficiently handles varying memory demands, particularly excelling with larger workloads due to its focus on short-lived objects

SUB-PART-3

Metric	Full GC	Incremental GC	Generational GC	No GC
Branch Misses	355,800	400,295	416,634	341,400
Page Faults	10,863	10,988	9,865	16,258
Cache Misses	5,462,468	7,958,781	6,803,041	1,597,491
Instructions	1,233,271,924	1,304,586,999	1,258,602,067	1,100,322,973
Instructions/Cycle	2.12	1.84	1.71	2.87
Cycles	581,551,967	710,415,779	735,535,149	382,791,710
Elapsed Time (s)	0.145	0.179	0.184	0.090
User Time (s)	0.130	0.162	0.172	0.068
System Time (s)	0.015	0.017	0.012	0.022

Metric Analysis

1. **Branch Misses**

- Full GC: 355,800
- Incremental GC: 400,295
- Generational GC: 416,634
- No GC: 341,400

Relevance: Branch misses occur when the CPU incorrectly predicts the outcome of a branch (e.g., an `if` statement). In scenarios involving GC, branch misses may increase due to the complexity and unpredictability of memory management tasks.

Variation Explanation:

- The No GC configuration has the lowest branch misses, as it avoids the complexity associated with garbage collection.
- Generational GC has the highest branch misses, due to its more frequent checks and collections, particularly in managing different generations of objects.
- Incremental GC and Full GC have more branch misses than No GC but fewer than Generational GC, reflecting their intermediate levels of complexity.

2. Page Faults

- Full GC: 10,863
- Incremental GC: 10,988
- Generational GC: 9,865
- No GC: 16,258

Relevance: Page faults occur when the CPU accesses a part of memory that isn't currently mapped to physical RAM, requiring the OS to handle it. This is relevant in scenarios with memory allocation and deallocation, such as GC.

Variation Explanation:

- No GC has the highest page faults, due to the continuous memory allocation without any cleanup.
- Generational GC has the lowest page faults, due to its efficient management of short-lived objects and its focus on frequently accessed memory regions.
- Full GC and Incremental GC have similar page faults, slightly higher than Generational GC but much lower than No GC.

3. Cache Misses

- Full GC: 5,462,468
- Incremental GC: 7,958,781
- Generational GC: 6,803,041
- No GC: 1,597,491

Relevance: Cache misses occur when data required by the CPU is not found in the cache, resulting in slower memory access. GC can cause cache misses by moving objects in memory, invalidating cache lines.

Variation Explanation:

- No GC has the lowest cache misses, as it avoids the cache-disruptive activities of garbage collection.
- Incremental GC has the highest cache misses, due to its frequent small collections that disturb the cache.
- Full GC and Generational GC have intermediate cache misses, with Generational GC performing slightly worse due to its frequent young generation collections.

4. **Instructions Per Cycle (IPC)**

- Full GC: 2.12
- Incremental GC: 1.84
- Generational GC: 1.71
- No GC: 2.87

Relevance: IPC measures how many instructions are executed per CPU cycle, reflecting CPU efficiency. A higher IPC indicates better performance.

Variation Explanation:

- No GC has the highest IPC, showing the CPU's high efficiency without the overhead of garbage collection.
- Full GC has the highest IPC among the GC configurations, suggesting that its infrequent but comprehensive collection cycles are less disruptive to CPU performance.
- Incremental GC and Generational GC have lower IPCs, due to the frequent interruptions caused by their respective collection strategies.

Summary of Trends Across GC Configurations

1. Branch Misses: Increase with the complexity and frequency of GC operations, with Generational GC having the most due to its focus on multiple generations.
2. Page Faults: Highest in the No GC scenario, where memory allocation is continuous without reclamation. Generational GC is the most efficient, with the fewest page faults.

3. Cache Misses: Lowest in the No GC configuration, with Incremental GC being the most disruptive to the cache due to frequent collections.
4. Instructions Per Cycle (IPC): Highest in the No GC scenario, reflecting the efficiency without GC overhead. Full GC maintains a relatively higher IPC compared to other GC configurations.

Conclusion

1. No GC is the most efficient configuration in terms of branch misses, page faults, cache misses, and IPC, but it doesn't manage memory, which can lead to increased page faults and potential memory exhaustion.
2. Generational GC balances efficiency with memory management, showing the lowest page faults and moderate performance in other metrics.
3. Full GC and Incremental GC introduce more overhead, with Incremental GC being the least efficient due to its frequent interruptions, reflected in higher cache misses and lower IPC.