CS60203: Design Optimization of Computing Systems *August 28, 2024*

Assignment 1 : Part 2

Benchmarking Lua's GC

Instructor: Mainack Mondal *Written-By: Sanskar Mittal*

Part 1

Full GC	Incremental GC	Generational GC
22.71	14.93	18.87

Table 1: Percentage(%) of instructions consumed by different GC configurations
Here, n = 100 and m = 1000

1. **Full GC** : The relatively high percentage (**22.71%**) indicates that this process is more instruction-intensive. Full GC typically pauses the entire application, making it more expensive in terms of computational resources. This higher cost is expected because the GC must examine every object, which is time-consuming and costly in terms of instructions.

2. **Incremental GC** : Incremental GC divides the GC process into smaller chunks, spreading the work across multiple cycles rather than completing it all at once. The lower percentage (**14.93%**) compared to Full GC suggests that while it still incurs some overhead, it is more efficient in its use of instructions, likely because it avoids the large, disruptive pauses associated with Full GC.

3. **Generational GC** : Generational GC optimizes by focusing on collecting young objects (which are more likely to be short-lived) more frequently while performing full collections less often. It balances the need for frequent small collections with occasional full sweeps, making it a middle-ground in terms of efficiency.

Part 2

n	m	Full GC	Incremental GC	Generational GC
100	100	22.59	12.47	17.07
100	500	22.69	15.78	14.49
100	5000	22.71	17.83	15.05

Table 2: Percentage(%) of instructions consumed by different GC configurations for different m values

Trend Analysis for Garbage Collectors -

1. **Full GC** performs a complete sweep of all objects in the system, regardless of how much garbage is present. This consistency suggests that Full GC's workload **scales directly** with the **size** of the memory space it manages, rather than the amount of garbage which is quite evident with its stop-the-world functionality.

2. **Incremental GC's** workload grows with the amount of garbage because it spreads the collection process over multiple cycles. As m increases, there is more garbage to collect, leading to **more incremental steps** and thus more instructions consumed.

3. With more garbage, **Generational GC** proves to consume the least percentage of instructions among all three GCs. This is because it divides its garbage collection process among young and old generations. By focusing on collecting young objects, which are more likely to be short-lived, Generational GC minimizes the need to **frequently scan older objects**, which are less likely to be garbage. This division allows it to handle large amounts of garbage more efficiently, as it can quickly reclaim memory from young objects while deferring the more costly collection of older objects.

Part 3

GC configuration	Branch Misses	Page Faults	Cache Misses	Instructions Per Cycle
None	2,21,619	**16,254**	16,75,494	**2.90**
Full	2,66,008	**16,254**	16,91,255	2.76
Incremental	2,96,571	11,034	**36,67,683**	2.03
Generational	**3,69,099**	9,926	26,40,932	2.27

Table 3: Metric analysis using Perf for different GC configurations
Here, n = 100 and m = 1000

Perf Analysis for different metric -

1. **Branch Miss**: Branch misses occur when the CPU's branch predictor fails to correctly predict the outcome of a conditional statement, leading to pipeline stalls and performance degradation. The likelihood of branch misses is influenced by the complexity and unpredictability of the control flow within the GC algorithm. Hence, it is a **relevant** metric for GC configuration analysis.

   The number of branch misses is highest for the **Generational GC** configuration. This indicates that Generational GC introduces the most complex and unpredictable control flow, leading to a higher rate of branch prediction failures by the CPU.

   On the other hand, **No GC** configuration has the least number of branch misses, indicating a simpler and more predictable control flow.

2. **Page Faults**: Page faults occur when a process attempts to access a memory page that is not currently in physical memory. This results in a context switch to the operating system, which

then loads the required page from disk into memory. Page faults are a significant source of performance overhead, as disk access is orders of magnitude slower than memory access. A good GC with low page fault rate showcases a fast and efficient memory management system. Hence, it is a **relevant** metric for GC configuration analysis.

The number of page faults is lowest for the **Generational GC** configuration. This showcases that it is more efficient in terms of memory access, likely due to its focus on young objects that are frequently accessed, reducing the need to access memory pages that aren't already in RAM.

3. **Cache Misses**: Cache misses occur when the CPU attempts to access data that is not present in the cache, necessitating a slower access from the main memory. This results in performance overhead because retrieving data from the main memory is significantly slower than accessing data from the CPU cache. A GC configuration that leads to fewer cache misses demonstrates that the GC operations and memory accesses are more predictable and localized, allowing the CPU cache to be more effectively utilized. Hence, it is a **relevant** metric for GC configuration analysis.

   The number of cache misses is highest for the **Incremental GC** configuration. This is likely due to its interleaving with program execution and smaller, more frequent memory operations that disrupt the CPU cache.

4. **Instructions per Cycle**: Instructions per Cycle (IPC) measures the average number of instructions a CPU executes per clock cycle. It provides insight into how efficiently the CPU is utilizing its processing resources. A higher IPC indicates that the CPU is effectively executing more instructions in each cycle, leading to better overall performance. GC configuration with a higher IPC values indicate that it is optimized to minimize pipeline stalls and disruptions, leading to better overall CPU performance and efficiency. Hence, it is a **relevant** metric for GC configuration analysis.

   The number of instructions per cycle is highest for **No GC** configuration. This is expected as it doesn't introduce any extra interference with the CPU's pipeline.