

CSE 605: Advanced Concepts in Programming Languages

Garbage First (G1) Garbage Collector

Team: Trash Talkers

Name	UB ID
Niharika Deshmukh	50471109
Harshith Pokala Muni	50478778
Viswanath Vankadara	50468995
Faizuddin Mohammed	50471170
Vinuthna Reddy Gutha	50471097

Report Outline

1. [Problem Statement](#)
2. [Related Work \(Checkpoint 3 included\)](#)
 - a. [Different Literature on Garbage Collectors.](#)
 - b. [Overview of Garbage collectors and techniques that currently exist \(Checkpoint 1, 2 and 3\)](#)
3. [Software Stack Description](#)
4. [Proposed Solution](#)
 - a. [Proposed Solution Checkpoint 1 & 2](#)
 - b. [Proposed Solution Checkpoint 3](#)
 - i. [Region - as small block of memory \(using marksweep GC\)](#)
 - ii. [Region - as space \(using CopyGC\)](#)
 - c. [What has changed in the proposed solution ?](#)
5. [Implementation](#)
 - a. [Implementing MarkSweep Garbage Collector](#)
 - b. [Implementing Region based Garbage Collector by extending MarkSweep GC](#)
 - c. [Implementing Region based Garbage Collector by extending CopyGC](#)
6. [Evaluation strategies](#)
 - a. [Measurements for benchmarks and hypothesis for each evaluation metric:](#)
7. [Results so far](#)
 - a. [Results of extending MarkSweep GC to build Region Based GC](#)
 - b. [Result of extending CopyGC to build Region based GC](#)
 - c. [Benchmark Evaluation on MarkSweep GC, CMC and Generational GC](#)
8. [Checkpoint 1 vs Checkpoint 2: Key Differences and Reasons for Change](#)
9. [Limitations](#)
 - a. [Potential Limitations\(Checkpoint 1 & 2\)](#)
 - b. [Limitations\(Checkpoint 3\)](#)
10. [Group Work Statement](#)
11. [References](#)

Problem Statement:

Let us discuss a case study to understand the importance of the project we are working on:

There was a team of engineers working on a seemingly benign problem. Their elastic search cluster was responding slowly. They tried the ad hoc techniques first, which is writing batch queries to reduce write operations, caching common responses, etc. It was not enough. They then move to complex solutions such as indexing the data on multiple fields, optimizing queries, etc. No improvement. And then? They finally moved to expanding the cluster and increasing the heap memory size. The problem got much worse. The problem was taken to the staff engineer and he asked, "Have you tried finding which type of request usually fails?", the engineers replied that there is no pattern.

After some discussion, the staff engineer solved the problem. The problem being: that under tremendous load, Elasticsearch was frantically searching for more heap memory to create index data structures.

As a last resort, it would ask the GC to collect dead objects to free up precious memory.

The Garbage Collector had no choice but to move into "Stop the World" Garbage Collections. Hence, user latency skyrocketed.

Modern high-performance applications often run into bottlenecks because of garbage collection (GC). GC is the process of automatically reclaiming memory that is no longer needed by the program. While existing GC algorithms, such as Generational Garbage Collection (Gen GC), have been improved over time, there is still a need for efficient GC mechanisms that can be tailored to specific use cases and environments. G1 Garbage collector is an advancement in GC's.

G1 GC aims for predictable pause times and handles large heaps efficiently by dividing the heap into many small regions, managing them individually to reduce fragmentation. It performs incremental compaction and some GC work concurrently with application threads to minimize pause times. Unlike traditional Generational GC, G1 GC has a sophisticated collection strategy focusing on collecting regions with the most garbage first, thus being more efficient. It's often better suited for applications with larger heap sizes and strict pause time requirements

Related Work (Checkpoint 3 included):

1. The '**Deconstructing the Garbage-First Collector**' paper by **Wenyu Zhao and Stephen M. Blackburn** from the **Australian National University** is vital for our project. It offers detailed insights into the G1 Garbage Collector's components, such as concurrent marking, generational collection, remembered sets, and barrier mechanisms. These aspects are integral for replicating G1's efficiency in reducing pause times and optimizing memory management. The paper's performance impact analysis, benchmarked with the DaCapo suite, provides a comprehensive framework for enhancing our garbage collector's design in the JikesRVM environment. Key results include a 64% and 93% reduction in GC pause times from concurrent marking and generational collection, respectively. The low space overhead of remembered sets (0.66%) and the moderate time overhead of barriers (12.4%) offer important benchmarks.
2. The insights from **Paul R. Wilson's "Uniprocessor Garbage Collection Techniques"** significantly helps our project. The paper's detailed examination of incremental garbage collection methods, designed to minimize pause times, aligns with our objectives in optimizing memory management. Wilson's discussion on techniques like interleaving garbage collection tasks with program execution provides a practical approach to enhance system efficiency. Furthermore, the emphasis on generational garbage collection, which focuses on frequent collection of younger generations, mirrors our strategy for effective heap memory management. These elements combined offer a robust theoretical and practical foundation, crucial for developing and refining our project's garbage collection strategies.
3. In the paper, **Garbage-First Garbage Collection** **David Detlefs, Christine Flood, Steve Heller, Tony Printezis** it can be summarized that the Garbage-First garbage collector is tailored for high-performance memory management in large multi-processor systems. It prioritizes efficiency, meeting soft real-time constraints, and maintaining high throughput. It achieves this by concurrently performing heap operations with mutation, using compact evacuation, and parallel processing. Key contributions include eliminating fine-grain free lists, efficient region prioritization for collection, and flexible scheduling of pauses. Future improvements aim to enhance efficiency through various strategies, including static analyses and heuristic tuning.

4. Memory Efficient Hard Real-Time Garbage Collection By Tobias Ritzau

The document discusses Memory Efficient Hard Real-Time Garbage Collection in the context of evolving software demands and hardware advancements. It addresses the challenge of unpredictability in runtime behavior due to complex software features, including garbage collection, traditionally considered unpredictable. Garbage collection, while easing developers' tasks by managing memory, often requires substantial additional memory.

The thesis introduces a predictable garbage collection method, real-time reference counting, enhancing memory efficiency by about 50% compared to previously presented efficient garbage collectors. It emphasizes the importance of predictability in real-time systems and proposes an optimization technique, object ownership, to improve performance by eliminating redundant reference count updates. This optimization is not limited to reference counters and can benefit other incremental garbage collectors.

Additionally, the document presents a static garbage collector capable of allocating objects statically or on the runtime stack. It inserts explicit instructions to reclaim heap-allocated memory, potentially eliminating the need for runtime garbage collection in a significant portion of Java applications. This static garbage collection method also offers the possibility to remove costly synchronization instructions, expanding its application beyond memory management.

The significance lies in offering a predictable garbage collection method, enhancing memory efficiency, and introducing static garbage collection techniques that could potentially alleviate the need for runtime garbage collection in various Java applications, thus impacting both memory management and runtime performance.

5. Fast multiprocessor memory allocation and garbage collection

Author: Hans- J Boehm

The research paper focuses on enhancing the performance of a garbage-collecting memory allocator for multithreaded applications on multiprocessor systems. The design builds upon previous approaches, emphasizing issues relevant to small-scale multiprocessors and addressing specific concerns not covered elsewhere.

Key Points to be noted:

Garbage Collector Scalability: Minor modifications to the collector code can achieve reasonable scalability for garbage collection in multiprocessor environments without significant performance degradation on a uniprocessor.

Comparison with Malloc-Free Implementations: The allocator, serving as a plug-in replacement for malloc/free, is compared to scalable malloc-free implementations, particularly Hoard. Surprisingly, the collector outperforms Hoard in certain tests, showcasing its efficiency. A similar performance advantage is observed in another garbage-collecting allocator.

Synchronization in Allocators: Garbage collectors seem to require less synchronization compared to explicit allocators. This observation suggests the potential for deriving faster explicit allocators from the strategies used in garbage collectors.

Thread-Local Storage Access: The paper explores speedy access to thread-local storage, crucial in allocator design adhering to standard calling conventions. Empirical evidence suggests that, especially in the presence of a garbage collector, faster access to thread-local storage can be achieved in a thread-independent manner, casting doubt on the utility of standard thread library facilities for this purpose.

The study highlights the efficacy of the enhanced garbage-collecting memory allocator for multi-threaded applications on multiprocessor systems. It not only addresses specific issues related to small-scale multiprocessors but also demonstrates superior performance compared to certain existing scalable malloc-free implementations. Additionally, it questions the conventional wisdom regarding synchronization in allocators and thread-local storage access methods in the context of garbage collection.

6. Profile-guided Proactive Garbage Collection for Locality Optimization:

Author: Wen-ke Chen, Sanjay Bhansali, Xiaofeng Gao, Weihaw Chuang

We draw upon Chen et al.'s "Profile-guided Proactive Garbage Collection for Locality Optimization" to enhance garbage collection in JikesRVM using MMTk. Their paper presents a system that leverages the garbage collector to continuously improve program data locality at run time with low overhead. It uses sampled profiles of data accesses to guide both page and cache locality optimizations, and triggers locality optimization independently of normal garbage collection for space. It also combines page and cache locality optimization in the same system, achieving significant execution time improvements for several applications. Their innovative approach to data locality in garbage-collected languages informs our strategies for managing large heaps and optimizing memory management in JVM environments. This paper's focus on proactive collection and optimization is a crucial reference point for our work, offering insights into balancing efficiency and performance in complex systems.

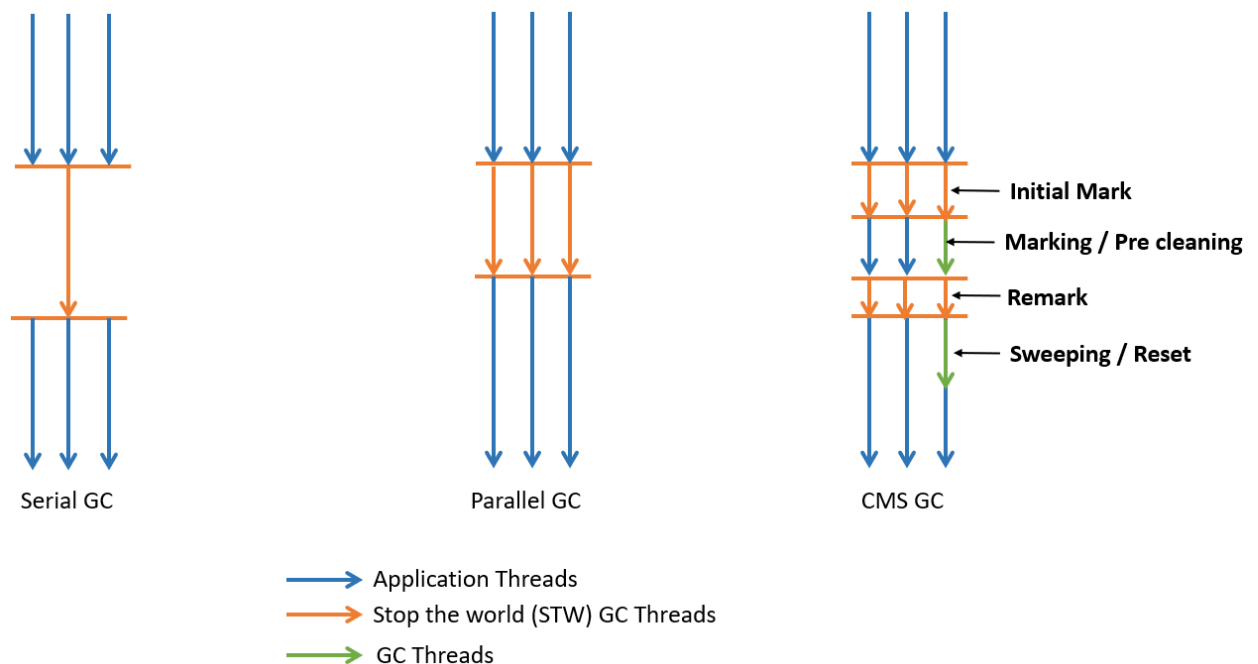
Overview of Garbage collectors and techniques that currently exist (Checkpoint 1, 2 and 3):

Serial Garbage Collector: The Serial Garbage Collector is a basic garbage collection algorithm in Java used to manage memory. This collector operates on a single thread for garbage collection tasks, making it simple in design and implementation. It employs a "stop-the-world" approach,

pausing all application threads during garbage collection to ensure safety in identifying and removing unreferenced objects.

Parallel GC: This collector uses multiple threads for garbage collection tasks, distributing work across cores to enhance performance. Similar to the Serial Collector, it employs a "stop-the-world" approach, pausing all application threads during garbage collection for a shorter duration compared to Serial GC. Suited for applications that can tolerate brief pauses to achieve enhanced throughput. It's suitable for mid-sized applications with moderate resource availability.

Concurrent Mark Sweep GC: The CMS Garbage Collector aims to minimize pauses by allowing application threads to run concurrently with certain garbage collection phases. Suited for applications that prioritize reduced pause times and aim to maintain responsiveness, such as web servers or applications with interactive user interfaces. Allows application threads to continue running during specific collection phases, minimizing interruptions and enhancing responsiveness.



Generational Garbage Collector:

The Garbage-First (G1) garbage collector in the Java Virtual Machine (JVM) is primarily based on the principles of generational and incremental garbage collection. Hence, we will be dwelling deeper into these Garbage collectors and how they work.

The Generational GC divides objects into different memory areas based on their ages. The heap is divided into two main areas: the Young Generation (for short-lived objects) and the Old Generation (for long-lived objects).

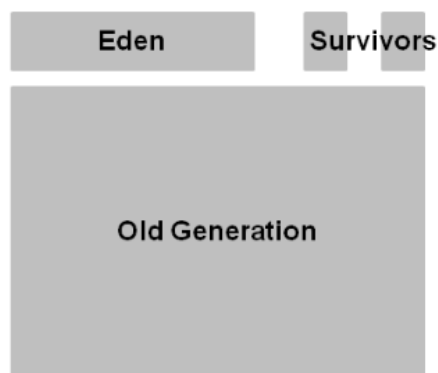
Let us now understand the Young Generation:

Young Generation: The Young Generation is further divided into three areas:

Eden Space: This is where new objects are initially allocated. It occupies a major portion of the Young Generation.

Survivor Spaces (S0 and S1): Objects that survive at least one garbage collection cycles in the Eden space are moved to the Survivor spaces.

Old Generation: Objects that have survived multiple garbage collection cycles in the Young Generation are eventually promoted to the Old Generation.



Strengths:

- **Improved Efficiency:** It efficiently handles short-lived objects, reducing the overhead during frequent collections.
- **Predictable Performance:** Minor collections in the Young Generation result in shorter, predictable pause times.
- **Optimized Memory Management:** Objects surviving multiple cycles get promoted, optimizing memory usage.
- **Adaptation to Application Behavior:** Generational collectors dynamically adjust generation sizes based on application behavior.

Drawbacks:

- **Complexity:** Managing separate heap areas, object promotion, and tuning require sophisticated algorithms.
- **Tuning Required:** They often need careful tuning for optimal performance.
- **Fragmentation:** Fragmentation issues in the Old Generation can impact the memory allocation efficiency.

Mark and Sweep

Process:

- The algorithm starts with root objects, which are references held in registers, stack, or global variables.
- It traverses the object graph recursively or iteratively, marking each object it reaches.
- Various traversal approaches, like depth-first or breadth-first search, are used to explore the object graph.

Marking Mechanism:

Objects have a mark bit. When an object is reached, this bit is set to indicate that the object is alive.

Sweep Phase:

- The algorithm iterates over the heap, checking the mark bit of each object.
- If the mark bit is not set (an unreachable object), the memory occupied by the object is reclaimed.
- The reclaimed memory is added to a free list for future allocations.

The main drawback of Mark Sweep GC is **Fragmentation**: Although mark-sweep reclaims memory, it leads to fragmentation over time as objects are deallocated non-contiguously.

There is another algorithm called **Mark-Compact Garbage Collection** which fixes the problem of fragmentation

- By compacting live objects, the algorithm effectively minimizes memory fragmentation.
- **Longer Pause Times:** Due to the marking and compacting phases, the algorithm can induce longer pause times.

Garbage-First (G1) garbage collector

Garbage collection mechanism in G1GC is like that of Mark and Sweep Collector. G1 performs concurrent global marking of live objects present in the entire heap. Once the live objects are identified, G1 knows what kind of regions have empty spaces (less lively). G1 starts the evacuation process with these regions. G1 uses a pause prediction model to meet the

requirements of user-defined pause time targets and collects the regions according to the specified time targets. G1 copies and moves the objects from one region to another during the evacuation process. In this process, both compaction and freeing up of memory take place simultaneously. This helps in reducing the fragmentation issue. This is different from Mark and Sweep collector as it does not do compactions.

Two Factors to consider for running Java applications:

- a. **Responsiveness:** It refers to how quickly an application responds for a given request. Some of the examples include UI requests, querying in databases, opening pages from websites, etc. Long pause times are not feasible for applications with a focus on responsiveness. Short pause times are preferred for those kinds of applications.
- b. **Throughput:** It focuses on maximizing the amount of work done in each time frame. Some of the examples include the number of batch executions in an hour, the number of database queries processed in an hour, etc. Applications with a focus on Throughput can allow long pause times as their focus is on benchmarks.

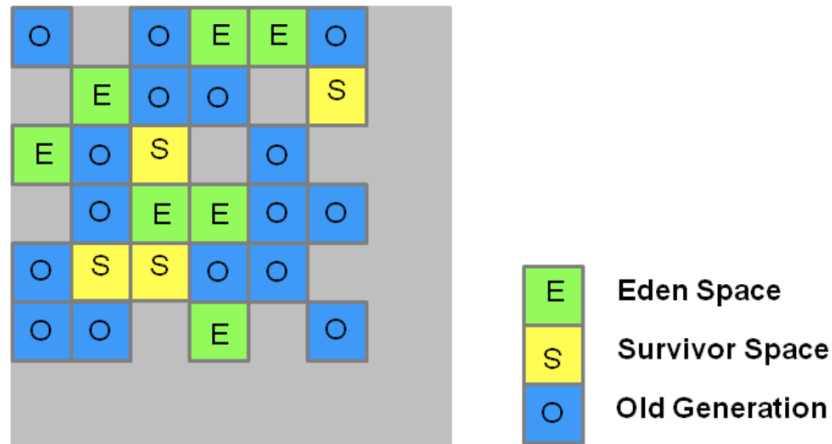
Accounting data structures in G1 that are responsible for minimizing or maximizing the size of each region in the heap. They are of two types:

- a. **Remembered Sets:** Tracks object references into a given region. For every region there is one RSet per region.
- b. **Collection Sets:** It contains the sets of regions that are collected in garbage collection. All the live data present in the sets are evacuated (copied or moved) during garbage collection.

How G1 GC works:

- a. The heap structure of G1 is split into many fixed size regions. Random size is chosen by JVM at the startup and the JVM generally targets around 2000 regions varying from 1 to 32 Mb. The memory is logically mapped into 3 spaces: Eden, Survivor and old generation spaces.

G1 Heap Allocation



- b. Memory regions are not required to be allocated in the contiguous manner like that CMS collector.
- c. In young garbage collection, the objects that are marked as live are moved from evacuated to survivor regions. If a certain threshold is met, they are even evacuated to old generation spaces. This happens in a stop-the-world pause.
- d. Old Generation Garbage Collection steps:
 - i. Initial Mark: It is a stop-the-world event. It identifies and marks the root regions and survivor regions that have a connection to the old generation objects.
 - ii. Root Region scanning: It happens simultaneously while the application is running. It scans the survivor regions and checks if there are any references to old generation objects. This process should be completed before any young GC.
 - iii. Concurrent Marking: It happens simultaneously while the application is running. It marks the living objects that are present in the entire heap. It can be interrupted by the young garbage collection process.
 - iv. Remark: It is a stop-the-world event. It completes the marking of live objects that were missing in the previous phase. It uses an algorithm called snapshot-at-the-beginning (SATB) which is faster than the algorithm that CMS collector uses.
 - v. Cleanup: It is a stop-the-world event. It performs accounting on live objects and completely free regions. The empty regions are reset and then added to the free list.
 - vi. Copying: It is a stop-the-world event. It copies or evacuates the live objects into new unused regions. This can be done with both young and old generation regions.

To summarize, the heap memory in G1 is split into regions. Young generation memory comprises a set of non-contiguous memory regions, making it easy to resize. The garbage collection events stop the world events i.e., all the application threads are stopped for performing this action. In old-generation GC, liveness information is calculated concurrently while the application runs. Snapshot-at-the-beginning is used for remarking which is faster than the algorithm used in the CMS collector. In the cleanup phase, the young generation and old generation are reclaimed at the same time.

Software Stack Description:

We will be using JikesRVM, a Java Virtual Machine (JVM). JVMs are crucial components that allow Java bytecode to be executed as native machine code on various platforms, ensuring the "write once, run anywhere" promise of Java.

JikesRVM's codebase is modular, with distinct components handling the various aspects of the JVM. The primary thing that we will be dealing with is MMTk.

MMTK:

The Memory Management Toolkit (MMTk) is a set of precise garbage collectors that have been used within JikesRVM

The parts of MMTk that we will be dealing with the most and that will be most useful for us are under the Plan (`org.mmtk.plan`) and Policy (`org.mmtk.policy`) packages.

To know the why behind this, let us know the purpose of these packages:

Plan Package:

Purpose:

At a high level, a plan orchestrates the garbage collection process. The plan package is concerned with the overall garbage collection strategy. It defines the high-level behavior of the garbage collector, including the phases of collection, the types of spaces used, and how they interact.

Contents: This package contains various garbage collection strategies such as generational, mark-sweep, etc. Each of these strategies might have its own sub-strategies or variations. For instance, under the generational strategy, there might be different implementations like generational copying or generational mark-sweep.

Relevance: Since we are building a G1 GC, using these strategies, and modifying them to fit our use case, plan package will most definitely be used.

We will leverage the collectors defined under this package and use them in such a way that helps us in building G1 GC. Say a Generational Collector (since G1 GC is a generational GC) like Copying Collector:

MMTk Plan gives five distinct classes, with consistent naming conventions indicating their function and inheritance. These classes include:

GenCopy: Main class representing the plan itself.

GenCopyCollector: Manages garbage collector threads.

GenCopyConstraints: Provides configuration information.

GenCopyMatureTraceLocal: Offers thread-local data structures for heap traversal.

GenCopyMutator: Handles operations specific to a mutator thread.

Policy Package:

Purpose: The policy package deals with the low-level details of memory management. It defines how individual regions of memory (or spaces) behave, including allocation, deallocation, etc.

Contents: This package contains definitions for various types of spaces, like CopySpace, MarkSweepSpace, etc. Each space has its own characteristics and behavior. For example, CopySpace is used in copying collectors where objects are copied from one space to another during collection.

Relevance: When dealing with the specifics of how memory regions behave or are managed, we would make use of the policy package.

To summarize:

The plan package defines the *what* and *when* of garbage collection i.e., what strategy to use and when to trigger various phases of the collection.

The policy package defines the *how* i.e., how each memory region behaves, how allocation and deallocation occur, etc.

Other parts of the software that might be useful for us would be:

org.mmtk.utility.options: This module contains the class Options, which provides a way to set and get options for the garbage collector.

org.mmtk.utility.Log: This module contains the class Log, which provides a way to log messages from the garbage collector.

org.vmmagic.pragma: This module contains the annotation @Uninterruptible, which indicates that a method must not be interrupted.

Proposed Solution:

Proposed Solution (Checkpoint 1 & 2):

The following are the main components of the proposed solution:

Region management: The proposed solution will implement a region manager to track the regions of the heap. The region manager will also be responsible for creating and deleting regions.

Garbage collection algorithm: The proposed solution will implement the G1 garbage collector algorithm. The G1 garbage collector algorithm will be responsible for selecting regions for collection and collecting the selected regions.

Evacuation: The proposed solution will implement an evacuation algorithm to copy live objects from one region to another during a garbage collection cycle.

Concurrent marking: The proposed solution will implement a concurrent marking algorithm to mark live objects in the heap while the application is still running.

The proposed solution will work as follows:

The region manager will track the regions of the heap.

The garbage collection algorithm will select regions for collection.

The evacuation algorithm will copy live objects from the selected regions to a new region.

The concurrent marking algorithm will mark live objects in the heap while the application is still running.

Once all live objects have been copied to the new region, the old region will be reclaimed.

Since we are dealing with objects and memory, we need a policy for memory allocation and deallocation, a memory management policy to be precise. We can make use of Space (org.mmtk.policy) for this purpose. Space is an abstract class that represents a region of memory managed under a specific memory management policy.

G1 GC divides the heap into multiple fixed-size regions. A region can be young, survivor, or old, depending on the GC's needs at any given time and objects are moved between these spaces.

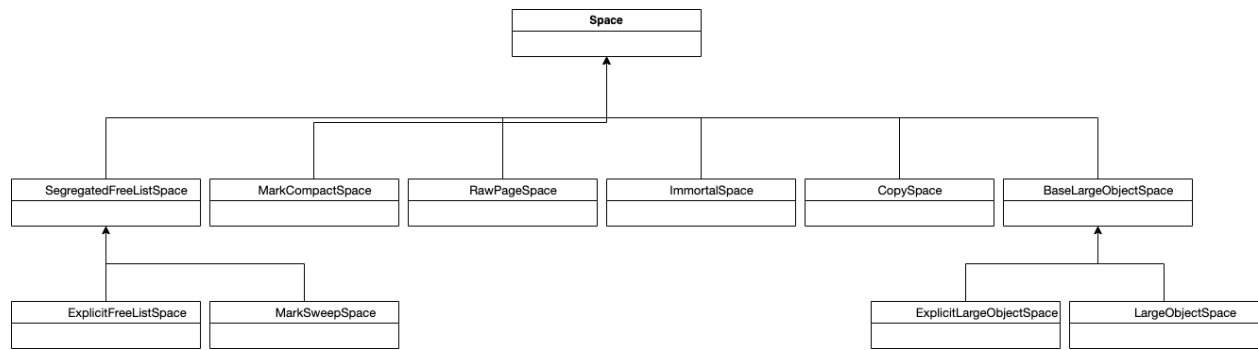
Here's what the Java doc for Space class says:

This class defines and manages spaces. Each policy is an instance of a space. A space is a region of virtual memory (contiguous or non-contiguous) that is subject to the same memory management regime. Multiple spaces (instances of this class or its descendants) may have the same policy (e.g. there could be numerous instances of CopySpace, each with different roles). Spaces are defined in terms of a unique region of virtual memory, so no two space instances ever share any virtual memory.

In addition to tracking virtual memory use and the mapping to policy, spaces also manage memory consumption (used virtual memory).

Our flow of implementation would be to:

- Define regions:
 - Unlike traditional generational GCs that have fixed-size young and old generations, G1 divides the heap into multiple fixed-sized regions (tiles)
 - Each of these regions is an instance of Space. This can be either young or survivor or old space
- Initialize Spaces:
 - We can have a initialization method to instantiate the required spaces
 - Since G1 can have many such small regions, we will maintain a data structure (likely an array) of these Space instances
- Handle Object Allocations:
 - Determine which region (or space) an object should reside in.
- Marking :
 - To help us with the marking phase, we intend to use the methods provided in the Space class, `traceObject` for instance, to assist with the marking phase of G1.
 - This method traces an object as part of a collection and returns the object, which may have been forwarded (if it is a copying collector).
 - But this is something we should implement as it is not present in the generational copying collector provided by MMTk.
- Evacuation & Compaction:
 - G1 performs evacuation of young regions and can also compact old regions that have a significant amount of garbage.
 - We intend to use the *release* method from the Space class to release memory regions that are no longer in use after evacuation or compaction
- Memory management:
 - We see that there are few methods provided by the Space class (`getPagesUsed()`, `getCollectionReserve()`) regarding memory usage.
 - We intend to use these methods to monitor and manage the memory consumption.
- Extend functionality of Space class (as the base functionality defined by the class wouldn't be enough to cater to the requirements of G1 GC).



These are all the spaces that are provided by JikesRVM.

No single space from the image perfectly aligns with the G1 GC's design. However, a combination of spaces might be used:

CopySpace could be adapted for young region management, where objects are frequently copied from one region to another.

MarkCompactSpace could be adapted for old region management, where objects are occasionally compacted.

There are a lot of unknowns at this point of time, how to actually convert this high-level stuff into a low-level design and translate that into working code. But one concrete unknown would be the **performance** of our G1 GC. While G1 GC aims for predictable response times, the actual performance in JikesRVM remains to be seen

Assumptions:

1. We assume that we proceed with an incremental development approach as it is suitable for incremental modification of components in the GenCopy, MarkSweep collector. It will make the testing process easier.
2. We assume that our proposed solutions are technically feasible with the existing architecture of JikesRVM. We assume that there are necessary resources to connect the modified components into the existing system.
3. We assume that our modifications will not introduce any errors in the existing collectors. We will validate it after implementation by testing.

Proposed Solution (Checkpoint 3)

After studying MMTK we decided to implement G1 GC as a region based garbage collector. On a high level we can think of a region in two ways.

1. Region can be a small chunk of memory.
2. Region can be an entire space.

If we deep dive into both the thoughts we get different designs of G1 inspired region based collectors. In this section we will see both of our proposed solutions to build a region based collector.

We also decided to implement the Region based collector in an iterative manner. Instead of focusing on all the phases and complexities at once, we decided to break G1 GC into phases.

In this approach, we decided to use StopTheWorld Plan of MMTK and proceed with complete heap trace. We decided to focus on concurrency in the future.

To conclude, both of our proposed approaches focus mainly on building the generational part of G1 Garbage Collector. The solution focuses on achieving memory allocation into the correct type of generation and achieving collection. Both of the solutions are not taking pause prediction and concurrency in considerations.

Region - as small block of memory (using MarkSweepGC implementation)

When we are thinking of region based collectors, we want each object to be identified as a young object, old object or a survivor object. One way to achieve this is allocating memory to an object and keeping metadata of whether the object is young, old or survivor. In this way the heap will be having discontinuous young space, old space and survivor space. Just to clarify, in this approach we are looking at young space, old space and survivor space as a type of region.

For better code structure, we propose to divide memory into a grid of small memory structures also known as region. Every region will maintain its status - young, old and survivor.

In this approach, we will be creating a logical data structure corresponding to a memory block which will be known as region.

To implement this approach using MMTK we have to maintain the code structure of MMTK.

While introducing any new Garbage collector into MMTK we have to take care of two things -

1. Policy
2. Plan

Policy is something which will see how the memory is managed and how the heap structure will look like. Policy in our case is nothing but Space.

Our policy will contain our basic data structure - Region.

To handle different regions and allocate memory according to Region we will have to use region specific memory allocation methods. Our policy i.e. our space should have methods to deal with Region. One way to achieve this is using existing policy and adding additional methods for region management. Another way to achieve this is creating a new policy which will specifically deal with Region data structure.

Thus in our checkpoint, we propose to introduce a new Region manager i.e. RegionSpace which will be adhering to MMTK structure and deal with managing regions. Its main purpose will be -

1. allocating regions based on requirement.
2. Keeping track of all regions.
3. Keeping track of young, old, survivor spaces and their respective regions.

To implement this RegionSpace, we will have to create a Plan. This step can also be looked in two ways

1. Create a new plan.
2. Use the existing GC plan and incorporate the new RegionSpace policy.

Let's take a look into both the approaches.

1. Create a new plan:

- a. Define Base class
 - i. Extends the Stop-The-World approach for garbage collection – initial mark, remark, cleanup, copying.
 - ii. Data Structures: Utilizes data structures like Region, Trace, and SharedDeque. It manages memory spaces, traces, and other resources.
 - iii. Manages memory regions through the RegionSpace class.
 - iv. Defines memory spaces like eden, survivor and old.
 - v. Orchestrates the different phases of the G1 GC cycle.
 - vi. Decides current phase of G1 GC cycle.
- b. Define Mutator class
 - i. Extends StopTheWorldMutator.java
 - ii. Allocates memory to objects.
 - iii. Methods present – alloc()
 - iv. Allocation is done through different kind of allocators -BumpPointers, SegregatedFreeList, LargeObjectAllocator
- c. Define Collector class
 - i. Extends StopTheWorldCollector.java
 - ii. Type of GC that will stop-the-world during the garbage collection
 - iii. Methods present – collect(), concurrentCollect(), allocCopy() and postCopy()
- d. Define TraceLocal class
 - i. Implementing core functionality for a transitive closure over the heap graph during the marking phase.
 - ii. Tracing and marking objects, updating references and handling remembered sets.
 - iii. Tracing and evacuating objects, updating references and handling remembered sets.
 - iv. Manages object relocation within the G1 region space and ensures proper handling of remembered sets

2. Use existing GC plan and incorporate the new RegionSpace policy

- a. Decision on existing plan - There are different plans in MMTK like - Marksweep, Marklocal, CopyGC, Generational Garbage collector. Each plan has a specific implementation, we found that the marksweep plan has a base structure, we can add additional methods and it will be the suitable plan to incorporate the regionSpace policy.
- b. How to modify MarkSweep Plan - Lets say that we are allocating memory to objects, in a new modification we were thinking of allocating memory to objects via region. In this plan, instead of dealing with memory directly, we will be dealing with it via Regions. We can say that we memory operations are abstracted, and all operations are done on regions.
- c. How will regions be allocated, data structures, allocators ? - Marksweep plan deals with memory using a segregatedFreeList allocator. SegregatedFreeList is allocating memory, using Address data structure. Now to deal with regions, we have to allocate memory to Regions. To do this, we can modify the SegregatedFreeList allocator to deal with regions. The elegant way to handle this is to extend a subclass from the SegregatedFreeList Allocator class, which will be RegionAllocator.
- d. To incorporate regionAllocator we modify the MarkSweepMutator class. As RegionAllocator is using SegregatedFreeList, there won't be much changes. We have to modify the alloc method - this method will by default assign objects into the Young kind of region, if the region is filled - we will create a new Young region.
- e. We have to modify the marksweep plan to implement different phases. These all phases can be defined in MarkSweep base class. In this solution we will incorporate additional phases to the MarkSweep.
- f. How will the collection work ? Collection will work in almost similar ways. In addition to collection, we will incorporate moving young objects to survivor regions. But as all the objects are in the same space, we don't need to move any regions, instead we change the type of regions and update the metadata of each region, which will enable us to achieve generational garbage collection feature. At the same time we will keep track of count and address of regions of each type-young, old, survivor.
- g. To deal with different phases, we have to trace objects of different types - young, old and survivor. This will be done in MarkSweepTraceLocal class. We will have to modify this class such that it will mark young type regions, old type regions and survivor type regions separately.
- h. To deal with the newly added complex phases, we will have to modify the existing base class of the MarkSweep plan. This is also the class where we will deal when the collection will trigger. Whenever total memory occupied with the young region crosses the threshold, the collector phase will be triggered.

After comparing both the approaches for the plan we decided to use the existing MarkSweep plan with the newly proposed RegionSpace.

This is our first summarized proposed solution:

1. Define Region Data structure.
2. Define RegionSpace policy to deal with regions.
3. Incorporate RegionSpace into MarkSweep plan.
4. Modify/Create a memory allocator to deal with region memory management.
5. Change the plan classes so that we will have basic region based garbage collectors.

Assumptions we are making for this solution:

1. The proposed solution assumes that we can achieve all the phases of a region based collector using markswEEP as a base plan.
2. It assumes that a segregated free list will be enough and efficient for memory allocation and won't have significant performance issues.

Region - as a Space (Using CopyGC Implementation)

We can implement region based garbage collectors by allocating three spaces - young, old and survivor into contiguous memory allocation instead of the previous approach. This approach contradicts the basis of G1 GC which is having all three spaces in discontiguous memory.

Our base thought of this approach was having three memory spaces, and moving objects from one space to another space so that we can achieve generational garbage collectors.

Policy: We thought of defining three spaces and defining a policy which will define methods to move objects from one space to another space.

After studying multiple garbage collectors from MMTK, we came to the conclusion of using CopyGC as a base plan.

The CopyGC plan is dealing with two spaces, nursery and mature space. In our case we needed three spaces which would be regions.

Our proposed solution contains defining three spaces - young space, survivor space and old space by taking analogy from the CopyGC plan.

1. How to modify an existing base class ? We thought of using base class as an orchestrator. It is a class which will define the three memory spaces.
2. How will the memory be allocated ? This approach is a bit simpler, as we will always allocate memory to young space.
3. What will happen when young space is full ? It will trigger initial collection and move the objects to survivor space.
4. Possibility of young space and survivor spaces filling : Whenever both the spaces will be filled, the collection will be triggered. And we will move objects from survivor space to old space, similarly objects of young space to survivor space.
5. How to deal with three different spaces and tracing of objects ? Tracelocal class has all three instances of memory spaces. So whenever required we will use those instances to trace objects.
6. How will collection happen and where will the object movement be triggered ? In the Collector class of copygc we will handle both the young collection and old collection. At the same time, we will handle the movement of objects from one space to another space.

To summarize, we proposed a second solution to achieve region based garbage collectors. In this solution we have three different spaces corresponding to young, old and survivor space.

To make this easy, we thought of using CopyGC as there are a lot of existing implementations for both the policy and plan.

Assumptions:

1. We are assuming that our incremental approach will work and be able to create three spaces.
2. We are assuming that moving objects from one space to another won't cause performance issues and won't create longer pause times.

What has changed in the proposed solution ?

1. In checkpoint 2 we thought of using an incrementing approach, but with a lot of unknowns. In checkpoint 3 we explored ways to achieve the expectations from checkpoint2.
2. We explored the existing GC and came to the conclusion that we can use existing GCs to build a region based garbage collector.
3. We acknowledged the fact that we can't achieve many phases from G1 GC, but we can use existing infra from MarkSweep GC and Copy GC to build a basic Region based GC.
4. Our above solution contains details of our approach, which can lead us to build G1 GC in an incremental way.

Implementation: (Code Review)

Implementing MarkSweep Garbage Collector

We have built the markswEEP collector in JikesRVM. We performed changes to the Tutorial class to perform both allocation and collection according to the Mark-Sweep policy. There are two steps involved in this process:

1. Changing the allocation from bump-allocation to free-list allocation.
2. Addition of mark-sweep policy to perform collection.

Let's take a look at the steps in detail:

- a. **Free-list Allocation**: This step is responsible for changing the simple collector from using the bump pointer to free-list

- i. Updated the TutorialConstraints class with the parameters used in the MarkSweep system. Following are the functions that were changed:

1. `getHeaderBits()` -modified to return `MarkSweepSpace.LOCAL_GC_BITS_REQUIRED`
2. `gcHeaderWords()` - modified to return `MarkSweepSpace.GC_HEADER_WORDS_REQUIRED`.

Reason for change: These methods are changed to reflect the constraints of a MarkSweep system.

- ii. Replaced the ImmortalSpace with a MarkSweepSpace:

1. `noGCspace` is renamed to `msSpace`.
2. `NOGC` is renamed to `MARK_SWEEP`.
3. The type and static initialization of `msSpace` is changed to:

```
public static final MarkSweepSpace msSpace = new
MarkSweepSpace("ms", VMRequest.discontiguous());
```

4. All the redundant imports are removed like `ImmortalSpace`. `MarkSweepSpace` is imported in this step.

- iii. The `ImmortalLocal` which is related to the bump-pointer is replaced with `MarkSweepLocal` (a free-list allocator) in the `TutorialMutator` class:

1. We made changes to the type of `nogc` and changed the static initializer accordingly.
2. The import statement of `ImmortalLocal` is replaced with `MarkSweepLocal`.
3. Replaced the name of `nogc` with `ms`.

- iv. The `postAlloc()` method is fixed to initialize the mark-sweep header:

```
if (allocator == Tutorial.ALLOC_DEFAULT) {
    Tutorial.msSpace.postAlloc(ref);
```

```

} else {
    super.postAlloc(ref, typeRef, bytes, allocator);
}

```

- v. Changed “default” to “ms” in the tutorial line in the PlanSpecificConfig class.

In **summary**, in this step the changes were made to reflect a transition from bump pointer to free-list allocator without introducing any garbage Collection yet. Naming conventions and import statements were adjusted accordingly to match the new classes. One issue that we can see in this is that the free list allocator might be slightly less efficient in space utilization than the bump pointer allocator. The implications of this should be considered, and potential optimizations can be explored.

- b. **Mark-Sweep Allocation:** The next step in the process was to perform Mark and sweep collection whenever the heap gets exhausted. Other MMTk components call the poll() method of plan class at suitable intervals to inquire about the need for a collection from the plan.

- i. Modifications were made in the TutorialConstraints class so that it inherits the constraints from the extending plan.

```
public class TutorialConstraints extends StopTheWorldConstraints
```

- ii. The plan should be aware of the process of garbage collection. The garbage collection is done in phases which are coordinated by the data structures defined in the StopTheWorld class. They also have global and thread-local components. We ensured that the global components were working collectively.

- 1. The tutorial class is changed by extending the StopTheWorld class instead of extending the Plan class.

```
public class Tutorial extends StopTheWorld
```

- 2. Renamed the trace variable to ms.
- 3. Made some code changes so that the tutorial performs correct global collection phases in collectionPhase.
 - a. Removed assertions that were not required for the code.
 - b. After completing the preparation steps related to the superclasses, we added the prepare phase, which prepares the space (msSpace) and the global tracer (msTrace).

```

if (phaseId == PREPARE) {
    super.collectionPhase(phaseId);
    nurserySpace.prepare(true);
    msTrace.prepare();
    msSpace.prepare(true);
}

```

```
return;  
}
```

- c. We included the closure phase and got the global tracer (msTrace) ready once again.

```
if (phaseId == CLOSURE) {  
    msTrace.prepare();  
    return;  
}
```

- d. Before completing the release phases connected to the superclass, we added the release phase and released the global tracer (msTrace) and the space (msSpace):

```
if (phaseId == RELEASE) {  
    nurserySpace.prepare(true);  
    msTrace.release();  
    msSpace.release();  
    super.collectionPhase(phaseId);  
    return;  
}
```

- e. Lastly, We confirmed that the phases are assigned to the superclass in all other cases by uncommenting the following after the conditionals mentioned above:

```
super.collectionPhase(phaseId);
```

4. Created a new way of calculating how much space a collection must make for changes. The method, `getPagesUsed`, is changed to override the one already in the `StopTheWorld` class.

```
@Override  
public int getPagesUsed() {  
    return super.getPagesUsed() + msSpace.reservedPages();  
}
```

5. Added a new method that informs whether there will be any movement during the collection.

```
@Override  
public boolean willNeverMove(ObjectReference object) {  
    if (Space.isInSpace(MARK_SWEEP, object))  
        return true;  
}
```



```
return super.willNeverMove(object);  
}
```

iii. In the next step we ensured that the Tutorial correctly performs the local collection phase.

1. The TutorialCollector class is modified to extend the StopTheWorld class. After extending and importing the StopTheWorld, we removed some methods that were implemented by the StopTheWorld class: like concurrentCollect() and concurrentCollectionPhase().
2. Code changes were made to perform the correct global collection phases.
 - a. Removed the unnecessary assertions in the code.
 - b. After completing the preparation steps related to the superclasses, we added the prepare phase and prepared the local tracer (trace). Assigned the following values to the phaseId == PREPARE clause in TutorialCollector.collectionPhase().

```
if (phaseId == Tutorial.PREPARE) {  
    super.collectionPhase(phaseId, primary);  
    trace.prepare();  
    return;  
}
```

- c. After completing the local tracer, we added the closure phase.

```
if (phaseId == Tutorial.CLOSURE) {  
    trace.completeTrace();  
    return;  
}
```

- d. Before carrying out the release phases connected to the superclass, we added the release phase and released the local tracer (trace).
 - e. Lastly, We confirmed that the phases are assigned to the superclass in all other cases by uncommenting the following after the conditionals mentioned above:

```
super.collectionPhase(phaseId, primary);
```

iv. In the final step, we ensured that the tutorial correctly performed local mutator-related collection activities.

1. Modified the TutorialMutator class to extend the StopTheWorldMutator. Imported and extended the required classes in this step.
2. Updated the phases in the mutator-side collection
 - a. Removed unnecessary assertions from the code.
 - b. Added the prepare phase to collectionPhase(). This prepares the mutator-side data structures for the start of a collection.

```

if (phaseId == Tutorial.PREPARE) {
    super.collectionPhase(phaseId, primary);
    ms.prepare();
    return;
}

```

- c. Release phase was added to collectionPhase() which re-initializes the data structures of the mutator after the end of the collection.

```

if (phaseId == Tutorial.RELEASE) {
    ms.release();
    super.collectionPhase(phaseId, primary);
    return;
}

```

- d. As a final step all other classes were delegated to the superclass.

```

super.collectionPhase(phaseId, primary);

```

In **summary**, in this step the changes made introduce a mark-and-sweep collection to the system, handling both global and local collection phases. The code was organized to follow the structure of a stop-the-world garbage collection plan.

- c. **Optimized Mark-sweep Collection** : One special ability of MMTk is that it enables the specialization of the performance-critical scanning loop. This is especially useful for collectors that have multiple modes of collection (generational collectors, for example), as each path for collection is specifically specialized at build time, eliminating conditionals from the hot part of the tracing loop at the collector's core.

It took two simple steps to enable this:

- i. Identified and modified the number of specialized scanning loops. The numSpecializedScans() method is overridden and a variable named SCAN_MARK is used to indicate the only specialized scan in the Tutorial.

```

@Override
public int numSpecializedScans() { return 1; }

public static final int SCAN_MARK = 0;

```

- ii. The specialized method is registered.
 - 1. The following line is added to the registerSpecializedMethods() method in the tutorial.

```
TransitiveClosure.registerSpecializedScan(SCAN_MARK,
TutorialTraceLocal.class);
```

2. Tutorial.SCAN_MARK is added as the first argument to the superclass constructor of the TutorialTraceLocal.

```
public TutorialTraceLocal(Trace trace) {
    super(Tutorial.SCAN_MARK, trace);
}
```

The main goal was to implement the Mark Sweep system. The changes were made to the tutorial to support both mark-sweep allocation and collection. After making all the necessary changes we then tested it to ensure the performance. We created a simple ‘HelloWorld’ program and observed the effects of garbage collection during the runtime. We also performed testing by changing the heap size. Through these steps we got a better understanding on garbage collection effects and management of heap size in the JVM.

```
vis@PeerOfPeer:~/Documents/CSE605/GC/31kesRV$ dlist/BaseBaseTutorial_x86_64-linux/rvm -X:gc:verbose=3 HeapTest
Key: (I)mmortal (N)onmoving (D)iscontiguous (E)xtent (F)raction
HEAP_START 0x0000200000000000
AVAILABLE_START 0x00002000dc000000
boot IN 0x0000200000000000->0x000003ffffffff E 0x0000020000000000
Immortal IN 0x0000400000000000->0x000005ffffffff E 0x0000020000000000
meta N 0x0000600000000000->0x000007ffffffff E 0x0000020000000000
los N 0x0000800000000000->0x000009ffffffff E 0x0000020000000000
sanity N 0x0000a00000000000->0x00000bffffffff E 0x0000020000000000
non-moving N 0x0000c00000000000->0x00000dffffffff E 0x0000020000000000
sn-code N 0x0000e00000000000->0x00000ffffffff E 0x0000020000000000
lg-code N 0x0001000000000000->0x000011ffffffff E 0x0000020000000000
ms N 0x0001200000000000->0x000013ffffffff E 0x0000020000000000
nursery 0x0001e00000000000->0x00001ffffffff E 0x0000020000000000
AVAILABLE_END 0x0000200000000000
HEAP_END 0x0000200000000000
Setting default thread count for MMtk to minimum of default thread count 2 and maximal thread count 2147483647 supported by current GC plan.
New default thread count value is 2
Setting actual thread count for MMtk to minimum of desired thread count 2 and maximal thread count 2147483647 supported by current GC plan.
New actual thread count is 2
Collection 1: reserved = 19 MB (5107 pgs) used = 10 MB (2683 pgs) total = 20 MB (5120 pgs)
Before Collection: used = 10.69 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 0.83 Mb + sanity 0.00 Mb + non-moving 0.17 Mb + sn-code 0.21 Mb + lg-code 0.00 Mb + ms 0.00 Mb + nursery 9.46 Mb
SOFT references: 0 -> 0
WEAK references: 20 -> 5
PHANTOM references: 0 -> 0
After Collection: used = 19.94 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 1.15 Mb + sanity 0.00 Mb + non-moving 0.17 Mb + sn-code 0.85 Mb + lg-code 0.00 Mb + ms 9.89 Mb + nursery 9.46 Mb
reserved = 19 MB (5093 pgs) used = 19 MB (5093 pgs) total = 20 MB (5120 pgs)
Collection time: 285.50 ms
GCWarning: Live ratio greater than 1: 1.46
Live ratio 1.00
GCloud 0.36
Heap adjustment factor is 1.36
GC Message: Heap changed from 20480KB to 30720KB
Collection 2: reserved = 29 MB (7678 pgs) used = 20 MB (5174 pgs) total = 30 MB (7680 pgs)
Before Collection: used = 20.26 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 1.15 Mb + sanity 0.00 Mb + non-moving 0.18 Mb + sn-code 0.85 Mb + lg-code 0.00 Mb + ms 9.89 Mb + nursery 9.78 Mb
SOFT references: 0 -> 0
WEAK references: 40 -> 19
PHANTOM references: 0 -> 0
After Collection: used = 20.54 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 1.15 Mb + sanity 0.00 Mb + non-moving 0.18 Mb + sn-code 0.85 Mb + lg-code 0.00 Mb + ms 9.37 Mb + nursery 9.78 Mb
reserved = 30 MB (7750 pgs) used = 20 MB (5246 pgs) total = 30 MB (7680 pgs)
Collection time: 139.88 ms
GCWarning: Live ratio greater than 1: 1.00
Live ratio 1.00
GCloud 0.88
Heap adjustment factor is 1.50
GC Message: Heap changed from 30720KB to 46080KB
vis@PeerOfPeer:~/Documents/CSE605/GC/31kesRV$
```

We'll take these learnings along with us into building G1 GC. We built a barebones version of Region Manager leveraging these.

Region based GC implementation by extending MarkSweep GC

To implement Region based GC using MarkSweep GC, we started by defining Region data structure which logically represents a memory block.

To define what kind of Region it is, we define three integers as an identifier. Region can be of young, survivor or old kind

```
public static final int EDEN = 0;
public static final int SURVIVOR = 1;
public static final int OLD = 2;
```

In the region, we define memory constraints for the region, its identifiers and meta data of the region.

```
public static final int LOG_PAGES_IN_REGION = 8;
public static final int PAGES_IN_REGION = 1 << LOG_PAGES_IN_REGION;
public static final int LOG_BYTES_IN_REGION = LOG_PAGES_IN_REGION +
LOG_BYTES_IN_PAGE;
public static final int BYTES_IN_REGION = 1 << LOG_BYTES_IN_REGION;
public static final Word REGION_MASK = Word.fromIntZeroExtend(BYTES_IN_REGION
- 1);
public static int PREV = 0;
public static int CURSOR = PREV;
```

We have written methods to identify the type of region. Methods to find region name and method checking the meta data. `getGenerationName` will give us generation of the region.

```
public static Address allocate(final Address region, final int size) {
    if (VM.VERIFY_ASSERTIONS) VM.assertions._assert(isAligned(region));
    Address regionEnd = region.plus(BYTES_IN_REGION);
    // get the current address of the cursor
    Address currAddress = getAddressOf(region, CURSOR);
    Address addressTill = currAddress.plus(size);
    if (AddressTill.GT(regionEnd)) return Address.zero(); // if address goes out of bounds of the
determined scope
    CURSOR += size;
    return currAddress;
}

static String getGenerationName(Address region) {
    int gen = getAddressOf(region, CURSOR).loadInt(); // but this only gets generation of
the current region where the cursor is at. But we might/ will need this at several instances
where the objects must be moved from one space to other. So there is a necessity to maintain
metadata as well.
    switch (gen) {
        case EDEN:    return "eden";
        case SURVIVOR: return "survivor";
        case OLD:     return "old";
        default:      return "???";
    }
}
```

To manage region data structures we define the space - `RegionSpace`.

This is the code of the `regionSpace` constructor and initial configuration. The memory allocated is discontinuous memory space.

```
int nurseryRegions = 0;
public int regionsInUse = 0;

public RegionSpace(String name) {
    super(name, true, false, true, VMRequest.discontiguous());
    pr = new FreeListPageResource(this, Region.METADATA_PAGES_PER_CHUNK);
}
```

```
}
```

We included methods to deal with generation in RegionSpace. `acquireRegion` method gives us the address of a new Region of given generation.

```
public Address acquireRegion(int generation) {
    // Acquire pages for the new region
    int pagesReserved = pr.reservePages(Region.PAGES_IN_REGION);
    Address region = pr.getNewPages(pagesReserved, Region.PAGES_IN_REGION,
zeroed);
    if (region.isZero()) {
        return Address.zero(); // Failed to allocate new region
    }
    // Initialize the new region
    if (generation != Region.OLD) {
        nurseryRegions += 1;
    }
    regionsInUse += 1;
    Region.register(region, generation);
    Region.set(region, Region.CURSOR, headRegion);
    headRegion = region;
    return region;
}
```

Apart from that RegionSpace also has methods like `releaseRegion`, which triggers Region class to clear metadata. Whenever we release a region, we release it from the free list as well.

RegionSpace also has methods to check if an object is live, if it is reachable so that we can trace it while marking and remarking.

```
@Inline
public void releaseRegion(Address region) {
    // Clear metadata -- write unregister method in Region
    Region.unregister(region);

    // Release memory
    ((FreeListPageResource) pr).releasePages(region);
}
```

```

@Override
public boolean isLive(ObjectReference object) {
    return ForwardingWord.isForwarded(object);
}

@Override
public boolean isReachable(ObjectReference object) {
    return !fromSpace || ForwardingWord.isForwarded(object);
}

```

Region based GC implementation by extending CopyGC

To implement a region based Garbage collector using CopyGC we created a new plan. We started by making a copy of the CopyGC package to SimGC.

Once we were done, we made changes to CopyGC to create three generations - young, old, survivor.

This is how we initialized all three spaces in the base class. Our young space was an instance of CopySpace, on the other hand survivor and old space was an instance of MarkSweepSpace. We defined the fraction factor for young space, which can later be used in copy and evacuation phase.

```

public static final int NURSERY = 0;
public static final int SURVIVOR = 1;
public static final int OLD = 2;

public static final CopySpace youngSpace = new CopySpace("young", true,
VMRequest.highFraction(0.15f));
public static final MarkSweepSpace survivorSpace = new MarkSweepSpace("survivor",
VMRequest.discontiguous());
public static final MarkSweepSpace oldSpace = new MarkSweepSpace("old",
VMRequest.discontiguous());

public static final int ALLOC_YOUNG = ALLOC_DEFAULT;
public static final int ALLOC_SURVIVOR = ALLOC_DEFAULT + 1;
public static final int ALLOC_OLD = ALLOC_DEFAULT + 2;

```

This is how our collection phase in base class looked like, which orchestrated the preparation of all spaces. It triggers collection using phaseId.

```
if (phaseId == PREPARE) {
    super.collectionPhase(phaseId);
    trace.prepare();
    survivorSpace.prepare(true);
    oldSpace.prepare(true);
    return;
}
if (phaseId == CLOSURE) {
    trace.prepare();
    return;
}
if (phaseId == RELEASE) {
    trace.release();
    survivorSpace.release();
    oldSpace.release();
    switchNurseryZeroingApproach(youngSpace);
    super.collectionPhase(phaseId);
    return;
}
super.collectionPhase(phaseId);
```

We defined the collectionRequired method, to see when to trigger collection. If any of the space is full or beyond threshold we trigger the collection.

```
public final boolean collectionRequired(boolean spaceFull, Space space) {
    boolean youngFull = youngSpace.reservedPages() > Options.nurserySize.getMaxNursery();
    boolean survivorFull = survivorSpace.reservedPages() >
Options.nurserySize.getMaxNursery();
    boolean oldFull = oldSpace.reservedPages() > Options.nurserySize.getMaxNursery();

    return super.collectionRequired(spaceFull, space) || youngFull || survivorFull;
}
```

Apart from that we modified all the methods of base class according to our requirement.

We modified our **SimCollector** to deal with all generations. This is how our constructor looked like.

```
public SimCollector() {
    los = new LargeObjectLocal(Plan.loSpace);
    youngSpace = new CopyLocal(Sim.youngSpace);
    survivorSpace = new MarkSweepLocal(Sim.survivorSpace);
    oldSpace = new MarkSweepLocal(Sim.oldSpace);
    trace = new SimTraceLocal(global().trace);
}
```

We modified our allocCopy to accommodate the copying and evacuation phase.

```
public final Address allocCopy(ObjectReference original, int bytes,
    int align, int offset, int allocator) {
    if (VM.VERIFY_ASSERTIONS)
        VM.assertions._assert(Allocator.getMaximumAlignedSize(bytes, align) >
            Plan.MAX_NON_LOS_COPY_BYTES);
    if (allocator == Sim.ALLOC_YOUNG) {
        return youngSpace.alloc(bytes, align, offset);
    } else if (allocator == Sim.ALLOC_SURVIVOR) {
        return survivorSpace.alloc(bytes, align, offset);
    } else if (allocator == Sim.ALLOC_OLD) {
        if (VM.VERIFY_ASSERTIONS) {
            VM.assertions._assert(bytes <= Plan.MAX_NON_LOS_COPY_BYTES);
        }
        return oldSpace.alloc(bytes, align, offset);
    } else if (allocator == Plan.ALLOC_LOS) {
        if (VM.VERIFY_ASSERTIONS)
            VM.assertions._assert(Allocator.getMaximumAlignedSize(bytes, align) >
                Plan.MAX_NON_LOS_COPY_BYTES);
        return los.alloc(bytes, align, offset);
    } else {
        // Default to the old space
        if (VM.VERIFY_ASSERTIONS) {
            VM.assertions._assert(bytes <= Plan.MAX_NON_LOS_COPY_BYTES);
        }
        return oldSpace.alloc(bytes, align, offset);
    }
}
```

```
}  
}
```

We modified post copy to deal with all three generations.

```
public final void postCopy(ObjectReference object, ObjectReference typeRef,  
    int bytes, int allocator) {  
    if (allocator == Sim.ALLOC_YOUNG) {  
        // Young space does not require post-copy actions  
    } else if (allocator == Sim.ALLOC_SURVIVOR) {  
        // Survivor space does not require post-copy actions  
    } else if (allocator == Sim.ALLOC_OLD) {  
        Sim.oldSpace.postCopy(object, true);  
    } else if (allocator == Plan.ALLOC_LOS) {  
        Plan.loSpace.initializeHeader(object, false);  
    } else {  
        // Default to the old space  
        Sim.oldSpace.postCopy(object, true);  
    }  
}
```

We modified our collectionPhase method to do collection according to our new defined phases.

```
public final void collectionPhase(short phaseId, boolean primary) {  
    if (phaseId == Sim.PREPARE) {  
        super.collectionPhase(phaseId, primary);  
        survivorSpace.prepare();  
        oldSpace.prepare();  
        trace.prepare();  
        return;  
    }  
    if (phaseId == Sim.CLOSURE) {  
        trace.completeTrace();  
        return;  
    }  
    if (phaseId == Sim.RELEASE) {  
        survivorSpace.release();  
        oldSpace.release();  
        trace.release();  
    }  
}
```

```

        super.collectionPhase(phaseId, primary);
        return;
    }

//    super.collectionPhase(phaseId, primary);
}

```

We introduced changes to SimMutator - so that object allocation will happen in the correct space. These are our changed methods in alloc() and postAlloc() methods.

```

public Address alloc(int bytes, int align, int offset, int allocator, int site) {
    if (allocator == Sim.ALLOC_YOUNG)
        return nursery.alloc(bytes, align, offset);
    else if (allocator == Sim.ALLOC_SURVIVOR)
        return survivor.alloc(bytes, align, offset);
    else if (allocator == Sim.ALLOC_OLD)
        return old.alloc(bytes, align, offset);
    else
        return super.alloc(bytes, align, offset, allocator, site);
}

```

```

public void postAlloc(ObjectReference ref, ObjectReference typeRef,
    int bytes, int allocator) {
    if (allocator == Sim.ALLOC_YOUNG)
        return;
    else if (allocator == Sim.ALLOC_SURVIVOR)
        Sim.survivorSpace.initializeHeader(ref, true);
    else if (allocator == Sim.ALLOC_OLD)
        Sim.oldSpace.initializeHeader(ref, true);
    else
        super.postAlloc(ref, typeRef, bytes, allocator);
}

```

We additionally incorporated some utility methods to make these changes work:

```

public Allocator getAllocatorFromSpace(Space space) {
    if (space == Sim.youngSpace) return nursery;
    else if (space == Sim.survivorSpace) return survivor;
    else if (space == Sim.oldSpace) return old;
    return super.getAllocatorFromSpace(space);
}

```

SIMTrace class - We modified it to trace objects in young, survivor and old region space.

```

public boolean isLive(ObjectReference object) {
    if (object.isNull()) return false;
    if (Space.isInSpace(Sim.NURSERY, object)) {
        return Sim.youngSpace.isLive(object);
    }
    if (Space.isInSpace(Sim.SURVIVOR, object)) {
        return Sim.survivorSpace.isLive(object);
    }
    if (Space.isInSpace(Sim.OLD, object)) {
        return Sim.oldSpace.isLive(object);
    }
    return super.isLive(object);
}

```

```

public ObjectReference traceObject(ObjectReference object) {
    if (object.isNull()) return object;
    if (Space.isInSpace(Sim.NURSERY, object))
        return Sim.youngSpace.traceObject(this, object, Sim.ALLOC_SURVIVOR);
    if (Space.isInSpace(Sim.SURVIVOR, object))
        return Sim.survivorSpace.traceObject(this, object);
    if (Space.isInSpace(Sim.OLD, object))
        return Sim.oldSpace.traceObject(this, object);
    return super.traceObject(object);
}

```

Evaluation Proposal:

In general, there are many effective techniques for comparing the relative performance of garbage collection algorithms over a wide range of parameter values. Measuring the time taken to traverse the heap during garbage collection cycles in mark-sweep is a relevant metric. It assesses the efficiency of heap traversal, which is a fundamental step in mark-sweep garbage collection. This step involves scanning the entire heap to identify live objects and reclaim memory from unreachable objects. The Mark-Sweep garbage collector will demonstrate competitive traversal times due to its focused approach on identifying and marking live objects and subsequently reclaiming memory from unreachable ones.

Measurements for benchmarks and hypothesis for each evaluation metric:

1. Micro Benchmarks:

- Pause Time Measurements:
 - To measure how long the garbage collector pauses the application during its cycles, timers or profiling tools can be used to record the duration of stop-the-world events caused by GC. Passing the '-XX:MaxGCPauseMillis' argument with a preferred pause time goal, this argument sets a target value for maximum pause time. The default is 200 milliseconds, G1 GC algorithm tries its best to reach this goal.
 - The G1 GC has a pause time target that it tries to meet. During the young collection, the G1 GC size of the young generation is adjusted in order to meet the real-time target, which includes the New and Survivor regions. Due to this, it is generally recommended to set the pause time target and let the GC change the heap as needed. It is important to note that the new generation size should not be set unless required.
 - Hypothesis: G1 GC shows significantly shorter pause times compared to traditional GC methods. Mark-Sweep may showcase shorter pause times compared to certain other traditional garbage collection methods due to its ability to perform both marking and sweeping concurrently in a single pause.
- Heap Traversal Time:
 - Benchmarking the time taken to traverse the heap for marking and sweeping operations.
 - Hypothesis: The mark-sweep collector may have a longer traversal time due to the need to scan the entire heap for marking and identifying unreachable objects.
- Memory Fragmentation Evaluation:

- The amount of memory utilized can be assessed and if fragmentation occurs over time, by monitoring metrics like heap occupancy, free space distribution, and compactness, G1 GC could potentially reduce memory fragmentation compared to other strategies.
 - Hypothesis: G1 GC demonstrates reduced memory fragmentation compared to other GC techniques. Mark-sweep might lead to higher fragmentation due to the nature of sweeping and deallocating memory in non-contiguous chunks. Mark-Sweep may potentially mitigate fragmentation by deallocating memory in non-contiguous chunks, leading to improved memory utilization compared to other algorithms, especially in scenarios with a higher rate of short-lived objects.
- Garbage Collector Overhead Analysis and Locality of Reference:
 - Total throughput is an important performance measure, related to two metrics: CPU Overhead and locality of reference.
 - CPU Overhead: The time needed to traverse and preserve reachable objects; the additional time required to maintain garbage collection invariants with each reference, (for generation collection, the time to maintain a list of intergenerational pointers; and for incremental collection, the time to transparently transport objects from freespace is included); the additional execution time caused by a choice of object representation convenient for garbage collection. This is the CPU overhead of garbage collection. Specifically, the mark-sweep algorithm represents this in the reference.
 - Reference Locality: The locality of reference is investigated at two levels: macro locality in the main memory and micro locality in the data cache. This contributes significantly to the performance of the GC. Traditionally, macro locality has strongly influenced the design of garbage collection algorithms. the cache miss ratio is the fraction of references to the cache that require access to the main memory.
 - Measuring the extra computational resources consumed by the GC by monitoring CPU utilization, memory usage, and GC-related threads' activity during application execution. G1 GC may demonstrate lower overhead due to its optimized collection strategy.
 - CPU Overhead Hypothesis: Mark-Sweep may demonstrate relatively higher CPU overhead due to its two-phase nature (marking and sweeping) involving full heap traversal and potential non-contiguous memory deallocation, causing increased computational resources compared to

some other modern GC algorithms. G1 GC might showcase lower CPU overhead due to its optimized collection strategy, which includes incremental and concurrent phases. G1's ability to perform collection concurrently with application execution could reduce the computational resources dedicated solely to garbage collection.

- Reference Locality Hypothesis: Mark-Sweep might exhibit limitations in optimizing reference locality, especially at the micro-level (data cache). Its approach of sweeping non-contiguous memory areas may impact cache performance, leading to a higher cache miss ratio, and affecting overall GC performance. G1 GC may demonstrate improved reference locality, particularly at the macro level in the main memory. G1's focus on regions and adaptability in managing memory could contribute to enhanced macro locality, potentially reducing the cache miss ratio and positively impacting overall GC performance.

2. Macro Benchmarks:

- Analysis of throughput:
 - Evaluating the ability of the application to perform the tasks within a specified time frame by executing stress tests or batch tasks to measure the number of completed operations per unit of time. In this way, G1 GC might show equal or improved throughput due to its concurrent nature.
 - Hypothesis: G1 GC exhibits either similar or higher throughput compared to other GC approaches. The mark-sweep collector might demonstrate lower throughput and slightly higher latency under moderate to heavy loads compared to other collectors due to its stop-the-world nature during the sweeping phase.
- Application-level Performance:
 - The overall impact of GC on the application's performance can be assessed by executing the application under real-world conditions and monitoring response times, latency, and resource utilization such that G1 GC might maintain or enhance the application's responsiveness compared to other GC methods.
 - Hypothesis: G1 GC maintains or improves the application's responsiveness compared to traditional GC methods. Mark-Sweep might demonstrate increased resource utilization and longer pause times during garbage collection cycles, leading to higher latency and potentially affecting the overall responsiveness of the application.

Evaluation and Results so far:

Results of extending MarkSweep GC to RegionBased GC:

After integrating RegionSpace into marksweep plan and making corresponding changes to MarkSweep Plan, we were able to successfully compile the JikesRVM code. Unfortunately, at the time, we did not take any screenshots.

However when we tried to execute a sample java program, we were encountered with a runtime exception.

After going through the code, we realized that our GC was failing at allocation. After carefully reviewing the code we noticed that our RegionSpace was not extending SegregatedFreeListSpace, we were directly extending the Space class.

We made the changes into class, but we were unable to make necessary changes in the implementation of SegregatedFreeList to give support for Region memory allocation.

Thus our first approach failed at allocation level itself and we did not reach the collection phase.

Results of extending CopyGC GC to RegionBased GC:

After creating three spaces, and making all necessary changes to CopyGC Plan we were able to successfully compile the JikesRVM code.

We created the test java code HeapTest.java to check the implementation of Simple Region Based Garbage Collector.

When executing the code, we again faced the runtime exception. However, this time we moved a step further. On our console we were able to see the different spaces created.

From the image below we can observe the memory address of HEAP_START, AVAILABLE_START and different spaces like immortal space, meta space, large object space, and the spaces which we created for region based garbage collection i.e. young, survivor and old.

We can also see the AVAILABLE_END and HEAP_END memory address.

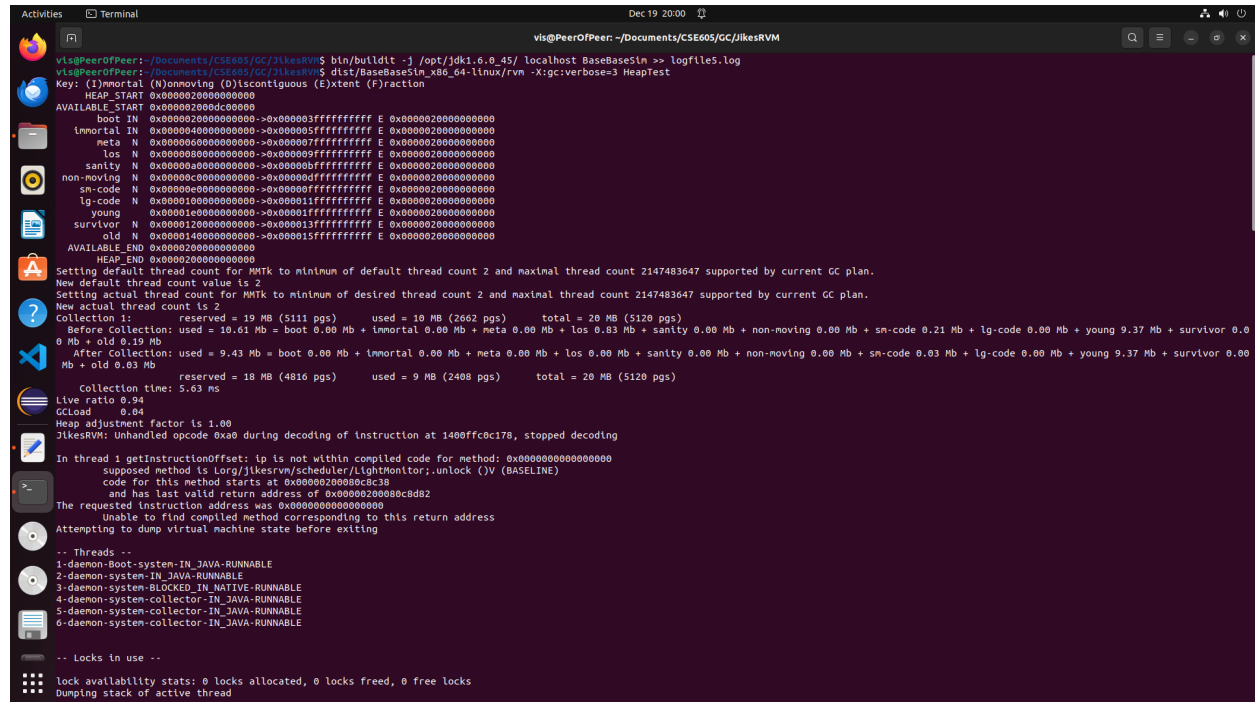
From the console message we were able to deduce that our code started executing and memory was allocated.

There are two types of messages where we see BEFORE_COLLECTION and AFTER_COLLECTION. Initially the memory used was 10.61Mb - 2662 pages, and after collection the memory used was 9.43 Mb-2408 pages.

We can also observe the memory allocated in different spaces before and after collection - in young, survivor and old spaces.

Later the code fails and dumps all the stack traces. According to our understanding, it was unable to understand instructions.

We deduced that our code execution was initiated and memory allocation happened. However, we were not able to conclusively pin point at point of failure.



```
Activities Terminal Dec 19 20:00
vls@PeerOfPeer: ~/Documents/CSE605/GC/jikesRVM
vls@PeerOfPeer: ~/Documents/CSE605/GC/jikesRVM$ bin/buildit -j /opt/jdk1.6.0_45/ localhost BaseBase$in >> logfile5.log
vls@PeerOfPeer: ~/Documents/CSE605/GC/jikesRVM$ dist/BaseBase$in_x86_64-linux/rvm -Xgc:verbose=3 HeapTest
Key: (I)mmortal (N)onmoving (D)iscontiguous (E)xtent (F)raction
HEAP_START 0x0000020000000000
AVAILABLE_START 0x0000020000000000
  boot IN 0x0000020000000000->0x000003ffffffffffff E 0x0000020000000000
  immortal IN 0x0000040000000000->0x000005ffffffffffff E 0x0000020000000000
  meta N 0x0000060000000000->0x000007ffffffffffff E 0x0000020000000000
  los N 0x0000080000000000->0x000009ffffffffffff E 0x0000020000000000
  sanity N 0x00000a0000000000->0x00000bffffffffffff E 0x0000020000000000
  non-moving N 0x00000c0000000000->0x00000dffffffffffff E 0x0000020000000000
  sn-code N 0x00000e0000000000->0x00000ffffffffffff E 0x0000020000000000
  lg-code N 0x0000100000000000->0x000011ffffffffffff E 0x0000020000000000
  young N 0x0000120000000000->0x000013ffffffffffff E 0x0000020000000000
  survivor N 0x0000140000000000->0x000015ffffffffffff E 0x0000020000000000
  old N 0x0000160000000000->0x000017ffffffffffff E 0x0000020000000000
AVAILABLE_END 0x0000020000000000
HEAP_END 0x0000020000000000
Setting default thread count for MMTk to minimum of default thread count 2 and maximal thread count 2147483647 supported by current GC plan.
New default thread count value is 2
Setting actual thread count for MMTk to minimum of desired thread count 2 and maximal thread count 2147483647 supported by current GC plan.
New actual thread count is 2
Collection 1: reserved = 19 MB (5111 pgs) used = 10 MB (2662 pgs) total = 20 MB (5120 pgs)
Before Collection: used = 10.61 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 0.03 Mb + sanity 0.00 Mb + non-moving 0.00 Mb + sn-code 0.21 Mb + lg-code 0.00 Mb + young 9.37 Mb + survivor 0.00 Mb + old 0.19 Mb
After Collection: used = 9.43 Mb = boot 0.00 Mb + immortal 0.00 Mb + meta 0.00 Mb + los 0.00 Mb + sanity 0.00 Mb + non-moving 0.00 Mb + sn-code 0.03 Mb + lg-code 0.00 Mb + young 9.37 Mb + survivor 0.00 Mb + old 0.03 Mb
Collection time: 5.63 ms
Live ratio 0.94
GCload 0.04
Heap adjustment factor is 1.00
jikesRVM: Unhandled opcode 0xa0 during decoding of instruction at 1400ffcb178, stopped decoding
In thread i getInstructionOffset: ip is not within compiled code for method: 0x0000020000000000
  supposed method is Long/jikesRVM/scheduler/LightMonitor;unlock (JV (BASELINE)
  code for this method starts at 0x0000020000000000
  and has last valid return address of 0x0000020000000000
The requested instruction address was 0x0000020000000000
Unable to find compiled method corresponding to this return address
Attempting to dump virtual machine state before exiting

-- Threads --
1-daemon-boot-system-IN_JAVA-RUNNABLE
2-daemon-system-IN_JAVA-RUNNABLE
3-daemon-system-BLOCKED_IN_NATIVE-RUNNABLE
4-daemon-system-collector-IN_JAVA-RUNNABLE
5-daemon-system-collector-IN_JAVA-RUNNABLE
6-daemon-system-collector-IN_JAVA-RUNNABLE

-- Locks in use --
lock availability stats: 0 locks allocated, 0 locks freed, 0 free locks
Dumping stack of active thread
```

Benchmark Evaluation on MarkSweep GC, CMS and Generational GC:

We chose ANTLR benchmark from DaCapo suite to evaluate the GCs. ANTLR (Another Tool for Language Recognition) is known for its parser generation workload, was chosen to assess how each GC strategy copes with the memory management demands of a real-world, memory-intensive Java application.

Garbage Collectors evaluated:

1. **Mark-Sweep (MS):** A traditional GC approach, characterized by its stop-the-world behavior during both the marking and sweeping phases.
2. **Concurrent Mark-Sweep (CMS):** Designed to minimize pause times by performing most of the GC work concurrently with the application threads.
3. **Generational Mark-Sweep (GenMS):** Incorporates generational GC concepts, aiming to optimize garbage collection by segregating objects based on their life spans.

Results (for small and large workloads):

- **Mark-Sweep (MS):**
 - **Execution Time:** 873 ms (small), ~25000 ms (large)
- **Concurrent Mark-Sweep (CMS):**
 - **Execution Time:** 1332 ms (small), ~41000 ms (large)
 - **Observations:** Although designed to reduce pause times, CMS showed longer total GC times for this workload. This could be due to the overhead of managing concurrent GC operations and the specific memory allocation pattern of ANTLR.
- **Generational Mark-Sweep (GenMS):**
 - **Execution Time:** 678 ms (small), ~13500 ms (large)
 - **Observations:** Demonstrated the best performance among the three, indicating that the generational approach effectively managed the allocation pattern of ANTLR. Likely benefited from optimizing collections for younger generations, where most objects were short-lived.

Analysis:

The Generational Mark-Sweep GC outperformed the other two strategies in the context of the ANTLR benchmark. This suggests that the generational approach, which prioritizes collecting short-lived objects, aligns well with the memory allocation behavior of ANTLR. Conversely, the CMS, despite its concurrent nature, incurred additional overheads that did not translate into performance benefits for this specific workload. The traditional Mark-Sweep, while simpler, was less efficient than GenMS but more so than CMS in this scenario.

Conclusion:

The results underscore the importance of choosing a GC strategy that aligns with the application's specific memory allocation and usage patterns. While Generational Mark-Sweep showed superior performance in this test, different applications and workloads might favor other GC strategies. These findings also highlight the complexities and trade-offs inherent in garbage collection mechanisms within Java virtual machines.

```
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseMarkSweep_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small antlr
===== DaCapo antlr starting =====
Running antlr on grammar antlr/java.g
ANTLR Parser Generator Version 2.7.2 1989-2003 jGuru.com
===== DaCapo antlr PASSED in 873 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseConcMS_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small antlr
===== DaCapo antlr starting =====
Running antlr on grammar antlr/java.g
ANTLR Parser Generator Version 2.7.2 1989-2003 jGuru.com
===== DaCapo antlr PASSED in 1332 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseConcMS_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small bloat
===== DaCapo bloat starting =====
Optimized with: EDU.purdue.cs.bloat.optimize.Main -only EDU.purdue.cs.bloat.trans -pre -dce -diva -prop -stack-alloc -peel-loops all -f EDU.purdue.cs.bloat.trans.ValueNumbering ./scratch/optimizedcode
===== DaCapo bloat PASSED in 9767 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseMarkSweep_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small bloat
===== DaCapo bloat starting =====
Optimized with: EDU.purdue.cs.bloat.optimize.Main -only EDU.purdue.cs.bloat.trans -pre -dce -diva -prop -stack-alloc -peel-loops all -f EDU.purdue.cs.bloat.trans.ValueNumbering ./scratch/optimizedcode
===== DaCapo bloat PASSED in 4703 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$
```

```
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseGenMS_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small bloat
===== DaCapo bloat starting =====
Optimized with: EDU.purdue.cs.bloat.optimize.Main -only EDU.purdue.cs.bloat.trans -pre -dce -diva -prop -stack-alloc -peel-loops all -f EDU.purdue.cs.bloat.trans.ValueNumbering ./scratch/optimizedcode
===== DaCapo bloat PASSED in 3751 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$ dist/BaseBaseGenMS_x86_64-linux/rvm -Xmx512M -jar $BENCHMARK_ROOT/dacapo/dacapo-2006-10.jar -s small antlr
===== DaCapo antlr starting =====
Running antlr on grammar antlr/java.g
ANTLR Parser Generator Version 2.7.2 1989-2003 jGuru.com
===== DaCapo antlr PASSED in 678 msec =====
vis@PeerOfPeer: ~/Documents/CSE605/GC/3ikesRVMS$
```

Checkpoint 1 vs Checkpoint 2: Key Differences and Reasons for Change:

1. Memory Allocation Policy

- Checkpoint 1 Proposal: The initial approach focused on using the Space class from org.mmtk.policy for memory management, with a basic idea of how regions are managed in G1 GC.
- Checkpoint 2 Changes:
 - Large Object Allocation Strategy: Handling objects that are too large for Thread-Local Allocation Buffers. Allocation directly in the old generation or a dedicated large object space. Allocation in Young and Old Generations: Detailed strategy for balancing allocations between young and old generations based on object age and size.
- Reason for Change: The initial assumptions about memory allocation were too broad and lacked the specific strategies necessary for G1 GC's efficient operation. We recognized the need for more sophisticated memory allocation strategies to optimize GC performance and reduce overhead.

2. Handling Fragmentation:

- Checkpoint 1 Proposal: Checkpoint 1 did not explicitly address fragmentation, especially in the old generation.
- Checkpoint 2 Changes:
 - Region Compaction Strategy: Dynamically compacting regions to reduce fragmentation, particularly in the old generation.
 - Concurrent Mark-Compact Algorithm: Implementing a concurrent algorithm to reduce application pause times while handling fragmentation.
- Reason for Change: Fragmentation, especially in the old generation, impacts the efficiency of GC.

3. Low-Level Design and Implementation Details:

- Checkpoint 1 Proposal: We gave a high-level design without delving into specifics.
- Checkpoint 2 Changes:
 - Region Management: Data Structure: Implementation of a dynamic array or linked list to manage heap regions. Each element in this structure represents a region and stores metadata such as region type (young, old, survivor), current occupancy, and garbage density.
 - Concurrent Algorithms Implementation:
 - Concurrent Marking: Development of a multi-threaded marking algorithm that runs concurrently with the application threads. This includes the use of atomic operations for marking objects and synchronization mechanisms to handle concurrent modifications. A concurrent-markable bit array or bitmap for each region allows the marking phase to quickly determine live objects.

- Concurrent Compaction: Integration of a concurrent compaction algorithm to reduce fragmentation, particularly in old regions. This involves identifying contiguous free spaces and moving live objects to compact the heap. Use of a forwarding table or similar structure to manage the relocation of objects during compaction.
- Reason for Change: The complexity of implementing a G1 GC requires a detailed understanding of each component's interaction, necessitating a more granular design approach.

4. Testing and Validation Strategy:

- Checkpoint 1 proposal: checkpoint 1 lacked a detailed testing strategy
- Checkpoint 2 Changes:
 - Unit Tests: To validate the functionality and performance of individual components of the G1 GC. Writing test cases for each module, such as region management, allocation, evacuation, and marking algorithms. Using tools to assess performance metrics like execution time and memory usage for each unit.
 - Integration Tests: To verify that different components of the GC work cohesively and interact correctly. Testing the interaction between the GC and other JVM components like the JIT compiler and thread management system. Validating the correct functioning of the GC in different heap configurations and under various workload patterns.
 - Stress Tests: To evaluate the robustness and stability of the GC under extreme conditions and high-load scenarios. Introducing artificial load and stress conditions, such as rapid object creation and destruction, to test the GC's responsiveness and performance under pressure. Monitoring for issues like memory leaks, excessive pause times, or crashes under stress.
- Reason for Change: Need for Comprehensive Testing: The complexity and critical nature of a garbage collector require a thorough testing strategy to ensure its reliability and efficiency.

Limitations

Potential Limitations (Checkpoint 1 and 2):

Mark-Sweep Garbage Collector (GC):

- Stop-the-World Pauses: Mark-Sweep GC involves stop-the-world pauses during both marking and sweeping phases, causing interruptions in application execution, and potentially leading to noticeable latency for interactive applications.
- Fragmentation and Memory Management: The nature of Mark-Sweep can lead to memory fragmentation, especially when deallocating memory in non-contiguous chunks, potentially impacting memory utilization and overall system performance.
- Limited Scalability: Mark-Sweep might face challenges in scalability when handling large heaps or high-throughput systems due to its stop-the-world nature, which might prolong pause times significantly with increasing data volumes.

G1 Garbage Collector (GC):

- Overhead Variability: While G1 aims for reduced pause times, it might encounter variability in actual pause durations due to factors like heap size, live data distribution, and application characteristics, impacting real-time applications.
- Complexity in Tuning: Configuring G1 GC parameters for optimal performance might be challenging. Fine-tuning heap size, pause time goals and other parameters requires deep understanding and extensive experimentation.
- High Memory Overhead: G1 might exhibit higher memory overhead compared to simpler collectors due to its concurrent nature and the need for additional bookkeeping data structures, impacting memory utilization.
- Initial Collection Overhead: During the initial stages or warm-up periods, G1 might experience longer pause times as it adapts and optimizes its internal regions and sizes to match the application's behavior.

Limitations (Checkpoint 3):

For this checkpoint we tried to implement a simple region based garbage collector. This was a step towards building GarbageFirst Collector.

We implemented both the region based solutions we proposed. In our first approach we failed before memory allocation.

Using our second approach, we were able to see some sort of allocation and collection happening. However, the main limitation of this approach was we are allocating space for each kind of generation, which contradicts with the basis of Garbage first collector, where each tile has its own generation.

Overall, we are looking at two kinds of limitations here:

1. Functional limitations
2. Implementation limitations

Functional limitations:

1. We were set to build a Garbage first collector, but at the end we were able to reach a simple region based garbage collector, which is the basic implementation of G1 GC.
2. In G1 GC, there are multiple phases like - concurrent marking, copying, evacuation, pause prediction analysis. We were not able to achieve all of them.
3. All the phases which are part of Garbage first collector are not implemented to its truest nature, instead they are implemented around the structure of MMTK.

Implementation limitations:

1. While implementing region based G1 GC which we were set to do, we faced issues while defining region based implementation in the first approach.
2. Although our approach sounded very close to Garbage's first collector, policy definition was not entirely accurate.
3. Our second approach was a bit successful in implementation, but our program did not terminate well.
4. It lacks the basis of garbage first, thus it is hard to extend it to complete working GC.
5. Both of the solutions adhere to the Stop the world Plan and need major work on concurrency.
6. We also deduce that although we were close to defining a working GC, its phases and implementation of each phase did not have clear division of responsibility.

What can be improved ?

1. We can improve the existing implementation of approach 1 by building an allocator supporting region allocation.
2. We believe if we are able to rectify our first approach, we can introduce other features as plugins and can achieve working Garbage First collector.
3. The additional parts will be working on concurrency, pause predictor, removing dependency from stop the world event.
4. For the second approach, we can make it work by clearly defining phases suitable to three spaces and add additional features like features in an iterative manner.

Group work statement

1. Vinuthna Reddy -
 - a. Reading material, understanding all types of GC and their differences.
 - b. Report preparation
 - c. NoGC building and Region based GC solutioning.
 - d. Presentation preparation and delivery.
2. Faiz -
 - a. Reading material, understanding all types of GC and their differences.
 - b. Report Preparation
 - c. MarkSweep GC building and Region based GC solutioning.
 - d. Presentation preparation and delivering.
3. Viswa -
 - a. Reading material, understanding all types of GC and their differences.
 - b. Working on region based collectors (extending Marksweep GC and NoGC to Region based collector)
 - c. Evaluation strategies
4. Harshith -
 - a. Reading material, understanding all types of GC and their differences.
 - b. Report preparation
 - c. MarkSweep GC, GenCopy GC building and Region based GC solutioning.
 - d. Code review, presentation preparation
5. Niharika -
 - a. Reading material, understanding all types of GC and their differences.
 - b. Working on region based collectors (extending Marksweep GC and NoGC to Region based collector)
 - c. Presentation preparation and delivery

We would like to make a note that all commits are made by Viswa, as we were able to test the changes locally in his machine.

We have two separate PRs for our two approaches, and we did not merge them in the main branch for our convenience.

Github: <https://github.com/ImVis10/G1GC/>

References:

1. <https://github.com/JikesRVM/JikesRVM>
2. Paul R. Wilson. 1992. Uniprocessor Garbage Collection Techniques. In Proceedings of the International Workshop on Memory Management (IWMM '92). Springer-Verlag, Berlin, Heidelberg, 1–42.
3. [G1 Garbage Collector](#)
4. <https://chat.openai.com/> - to understand a few parts of the code.
5. A [LinkedIn excerpt](#) about garbage collection
6. [Mark-and-Sweep: Garbage Collection Algorithm](#)
7. [Micro benchmarks Vs Macrobenchmarks](#)
8. [Performance Testing, Load Testing & Stress Testing](#)
9. [Maximum GC Pause Time](#)
10. [Best practice for JVM Tuning with G1 GC](#)
11. [The Impact of Garbage Collection on Application Performance](#)
12. Comparative Performance Evaluation of Garbage Collection Algorithms by Benjamin G. Zorn
13. <https://engineering.linkedin.com/garbage-collection/garbage-collection-optimization-high-throughput-and-low-latency-java-applications>
14. <https://github.com/wenyuzhao/JikesRVM-G1>
15. <https://users.cecs.anu.edu.au/~steveb/pubs/papers/g1-vee-2020.pdf>
16. [Memory Efficient Hard Real-Time Garbage Collection By Tobias Ritzau](#)
17. [Fast multiprocessor memory allocation and garbage collection Author: Hans- J Boehm](#)
18. [Profile-guided Proactive Garbage Collection for Locality Optimization: Author: Wen-ke Chen, Sanjay Bhansali, Xiaofeng Gao, Weihaw Chuang](#)
19. [Garbage-First Garbage Collection David Detlefs, Christine Flood, Steve Heller, Tony Printezis](#)
20. [Deconstructing the Garbage-First Collector' paper by Wenyu Zhao and Stephen M. Blackburn from the Australian National University](#)