

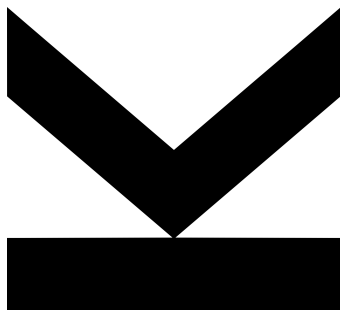
Author
Paul Lehner, BSc BSc
K12145051

Submission
Institute for System
Software

Thesis Supervisor
Univ.-Prof. Dr. Stefan
Marr

September 2026

Partial Compaction in the Garbage-First Garbage Collector



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

Abstract

In programming, it is necessary to allocate memory at runtime. To do that, the object's required space must be allocated. Some programming languages do not automatically monitor allocated memory, leaving memory management to the programmer. Memory no longer used must be freed manually, or it will not be available for further allocation. To avoid manual handling, some programming languages rely on an algorithm called Garbage Collector. Java uses such a Garbage Collector.

This thesis extends the standard Garbage Collector of the Java HotSpot VM, the Garbage-First Garbage Collector (G1). During allocation, sub-areas of the heap, called regions, are selected to be filled. This filling is not perfect, as some memory remains unused at the end. G1 currently accepts the waste. Suitable data structures are implemented to remember holes and to find and provide them for allocation efficiently. These changes invalidate some important assumptions of the G1 Garbage Collector and must therefore also be adjusted. By making these changes, we expect to lower overall pause time during the garbage collection phase, since we can delay running out of memory and therefore trigger a garbage collection cycle.

By evaluating the DaCapo and Renaissance benchmarks, we find that the red-black tree offers the best trade-off between insertion and lookup cost and is therefore selected as the memory data structure. Reusing holes at the end of survivor regions reduces total garbage collection pause time by approximately 19% on average compared to an unmodified baseline. A preliminary evaluation of reusing space at the end of humongous regions shows a smaller but still positive effect, reducing pause time by about 3% on the benchmarks where this mode currently runs. These results show that the memory G1 currently discards as waste is a meaningful and recoverable source of allocation capacity.

Contents

1	Introduction	5
2	Background	7
2.1	Java Virtual Machine	7
2.2	Garbage Collector	8
2.3	Garbage-First Garbage Collector	12
2.3.1	Design of Regions	13
2.3.2	Allocation	15
2.3.3	Concurrent Cycle	16
2.3.4	Remembered Sets	18
2.3.5	Eager Reclaim of Humongous Objects	20
2.3.6	Historic Differences of the G1 Garbage Collector	20
2.4	Data Structure for Managing Holes	23
3	Problem Statement: Holes in Regions Are Underutilized	25
3.1	Survivor Allocation Holes	25
3.2	End of Humongous Region Holes	26
3.3	Fragmentation of Old Region	27
4	Implementation: Utilizing Holes in G1 Regions for Allocations	28
4.1	Integration into the Allocation Path	28
4.1.1	Collecting Young Holes	29
4.1.2	Collecting Holes at the End of Humongous Objects	29
4.1.3	Collecting Old Holes	30
4.1.4	Path of Returning a Fitting Hole	30
4.2	Remembering and Finding Holes	31
4.2.1	Storing Holes in a Balanced Binary Search Tree	33
4.2.2	Storing Holes in a List Structure	35

4.2.3	Alternative Structures for Holes	36
4.3	Integration Into G1	37
4.3.1	Card Alignment of the Tail Region	38
4.3.2	Termination Condition of Concurrent Marking	38
4.3.3	Scan and Scrub of Tail	39
4.3.4	Eager Reclaim of Humongous Objects	40
4.3.5	Fix Reference Scan of Humongous Region	40
4.3.6	Updating Root Regions	42
5	Evaluation	43
5.1	Methodology	43
5.1.1	Benchmarks	44
5.2	Hole Size Analysis	44
5.3	Performance Evaluation	49
5.3.1	Comparison of the Data Structures	49
5.3.2	Final Benchmark Run	51
6	Conclusion and Future Work	58
	Bibliography	64

Chapter 1

Introduction

In modern software systems, efficient memory management is a cornerstone of both reliability and performance. While programming languages such as Java relieve developers from manual memory deallocation through automatic garbage collection, the underlying algorithms and their efficiency are crucial to application throughput and latency. The Java Virtual Machine (JVM), and specifically the HotSpot VM, has continuously evolved its garbage collection techniques, including the development of collectors such as the Garbage-First (G1) Garbage Collector. G1 was designed to balance throughput with predictably low pause times, leveraging a region-based heap layout and concurrent phases to address the varying needs of contemporary applications. Despite these advances, practical challenges remain. One such challenge is the underutilization of memory caused by fragmentation and unused spaces, referred to as “holes”, within heap regions. These holes can occur at the end of survivor regions, in humongous regions, and as internal fragmentation within old regions. Current implementations of G1 ignore this unused memory, leading to wasted resources and increased frequency of garbage collection cycles and longer pause times.

This thesis addresses this gap by proposing and implementing an extension to the G1 Garbage Collector: the active management and reuse of holes within regions. Rather than evacuating all live objects from a region and leaving the entire region empty for future allocations, a process that is expensive. This approach remembers and efficiently reuses the addresses of unused spaces and dead objects inside regions. By integrating suitable data structures for tracking holes, the allocator can satisfy allocation requests directly from these reclaimed segments, thereby reducing fragmentation and making more effective use of available memory. The proposed enhancements necessitate a careful reconsideration of some of G1’s core assump-

tions and mechanisms. For instance, the allocation of old objects after humongous objects, and the allocation of objects into previously fragmented space, require changes to allocation paths, marking invariants, and remembered set handling. This work details these modifications, evaluates alternative data structures for hole management, and analyzes their impact on allocation performance and garbage collection pause times.

The remainder of this thesis is structured as follows: Chapter 2 introduces the relevant background, including the design of the JVM, garbage collection principles, and the architecture of G1. Chapter 3 formulates the problem of underutilized holes in regions. Chapter 4 describes the implementation of the proposed changes, including the integration of new data structures and the resolution of challenges arising from altered G1 invariants. Chapter 5 presents an evaluation of the new approach, comparing performance metrics such as allocation overhead and GC pause times against the unmodified JVM. Finally, Chapter 6 summarizes the findings and discusses their implications for future research and JVM development.

Chapter 2

Background

This chapter presents the essential background information on the concepts and technologies relevant to this thesis. First, this chapter introduces the Java programming language, the Java Virtual Machine, and OpenJDK. Next, the chapter explains garbage collection, with a particular focus on the Garbage-First (G1) garbage collector. Finally, this thesis presents the relevant data structures used throughout.

2.1 Java Virtual Machine

Java is a statically typed, object-oriented programming language [1] and is widely used to develop applications for different platforms [2]. Java source code is compiled into *Java bytecode*. This bytecode is executed on the Java Virtual Machine (JVM) [3]. The JVM is an abstract computing machine [4] that provides a platform-independent execution environment for Java applications. It executes platform-independent Java bytecode through a combination of interpretation and Just-In-Time (JIT) compilation [5, 6]. With this abstraction layer between the operating system and the programming language, Java programs can run on different platforms without platform-specific modifications. Therefore, only the JVM must adapt to the target platform, while the same Java bytecode can run unchanged across different operating systems and hardware architectures.

A concrete JVM implementation is the HotSpot VM [7]. HotSpot is developed as part of the OpenJDK project, an open-source implementation of the Java platform [8]. HotSpot handles

multiple tasks such as loading and executing Java classes, managing memory, and performing garbage collection. Garbage collection is an integral part of the JVM. A central data structure is the *heap*. The heap is a contiguous memory area where the JVM allocates memory for Java objects and arrays. While the JVM specifies, for example, that the heap is shared among all JVM threads, the exact internal organization is implementation-specific [3]. So, while the garbage collector is an integral component of the JVM, the implementation is independent. So all changes for this thesis can be made inside G1 without violating JVM specifications since concepts like heap regions, generations, and remembered sets are provided by the individual garbage collector.

The following section introduces the general concepts and mechanisms of automatic garbage collection before describing the Garbage-First (G1) garbage collector in detail.

2.2 Garbage Collector

In manually managed programming languages, such as C++, the programmer must explicitly release dynamically allocated memory. But here programming errors can happen. If allocated memory is not released, it remains occupied and can not be reused for further allocations. Eventually, the heap runs out of memory, and the application crashes, which is known as *memory leakage* [9]. Even if memory is freed correctly, the programmer may still accidentally dereference a pointer to freed (and possibly reused) memory, leading to unpredictable results. Such a lost hanging pointer is called a *dangling pointer*. Similarly, in the case of freeing a pointer twice, *double free bugs* may occur, where already reassigned memory is unintentionally freed [10]. Automatic memory management addresses this problem by taking over memory management, detecting unused memory, and reclaiming it [11]. A key advantage of automatic memory management is that it reduces the likelihood of memory management programming errors.

Java uses automatic memory management by providing a garbage collector as part of the JVM. The garbage collector identifies objects the application can no longer reach. The reclaimed memory can be reused for new object allocations. Therefore, programmers do not explicitly deallocate Java objects as they would in a manually managed language like C++ with the keyword `delete`. The HotSpot JVM provides multiple garbage collection algorithms with

different trade-offs between throughput, latency, memory consumption, and other performance metrics [12, 13]. The core task of a garbage collector is to determine which objects are reachable and therefore alive. An object is *reachable* or *alive* if it can be accessed from a set of garbage collection roots (GC roots). GC roots are, for example, references held by live threads and VM-internal data structures. An object that cannot be reached from any GC root is considered *unreachable* or *dead* and can be collected. A garbage collection cycle can be divided into two fundamental operations: identifying reachable objects and reclaiming the memory occupied by unreachable objects.

Garbage collectors fall into two fundamental approaches: reference counting collectors and tracing collectors [11]. Reference counting collectors keep track of how many pointers reference each object. If the counter hits zero, the object is considered dead and can be reclaimed. While this approach does not require traversing an object graph, it must handle additional considerations, such as cycles. In a tracing collector, the first operation is commonly called *marking*. Starting from the GC roots, the collector traverses the object graph and identifies reachable objects. Objects that were not visited during the traversal are candidates for reclamation. The exact implementation of the traversal and when the application is stopped depend on the garbage collection algorithm. Some collectors can perform parts of the marking operation concurrently with the application.

The reclamation of unreachable objects can be performed using different strategies: By *non-moving* collectors or *moving* collectors [11]. In a non-moving collector, the memory occupied by dead objects is returned and remembered in a suitable data structure. In contrast, reachable objects remain in place in the heap. A common implementation is a free-list allocator, in which reclaimed memory blocks are maintained in one or more lists and can be reused for new allocations. Although this approach avoids the cost of moving live objects, it can lead to *heap fragmentation*. Fragmentation occurs when free memory is divided into multiple separated regions. Consequently, the total free memory may be sufficient for an allocation. In contrast, no individual contiguous free area is large enough to satisfy the allocation, which is known as *external fragmentation* [14]. Unlike non-moving collectors, moving garbage collectors address fragmentation by relocating reachable objects. During *evacuation*, live objects are copied from selected source areas into different destination areas. After all relevant objects have been evacuated, the entire source areas can be reclaimed. References to the moved objects must be updated to point to their new locations. In the garbage collector G1, the Java heap is divided into equally sized subdivisions called *regions*. Thus, rather than evacuating individual objects

to arbitrary locations in the heap, G1 selects regions for collection and evacuates their live objects into other regions. Once a region is completely evacuated, its memory can be reclaimed and reused for allocation.

A related technique is *compaction*. During compaction, reachable objects are relocated to occupy a more contiguous part of the heap, reducing fragmentation. In a sliding-compaction algorithm [15], for example, live objects are moved toward one side of a memory area while preserving their relative order. The memory following the relocated objects can then be treated as one contiguous free area. Unlike evacuation, compaction does not necessarily require a separate destination region for the entire collection area. So the difference between these two methods is that evacuation copies objects to a specific target area, while compaction moves objects within a larger memory area to reduce fragmentation.

Garbage collectors must perform these operations without consuming much computational time. A garbage collection pause during which application threads are stopped is called a *stop-the-world (STW) pause*. STW pauses simplify garbage collection because the running application cannot modify the object graph. However, long pauses negatively affect application latency and responsiveness. Modern garbage collectors therefore try to reduce the duration and frequency of STW pauses by doing some work concurrently with the application. This introduces additional computational overhead, creating a fundamental trade-off between application throughput and the latency garbage collection introduces to the application [16].

A fundamental observation underlying generational garbage collection is the *Weak Generational Hypothesis* [17], which observes that many objects have a short lifetime and become unreachable shortly after allocation. In particular, most objects do not survive their first garbage collection cycle, and only a small fraction survives multiple garbage collection cycles. Based on this observation, generational garbage collectors [18] divide the heap into two generations: the **Young Generation** and the **Old Generation**. Initially, new objects are allocated in the young generation and, if they survive enough collections, are promoted to the old generation. An object is called **young** iff it is part of the young generation. An object is called **old** iff it is part of the old generation. The number of garbage collection cycles that an object has survived is called the *age* of an object. This idea reduces the pause time, since traversing the entire heap during every garbage collection is computationally expensive and therefore inefficient. By defining two generations, we can collect young objects more frequently than old objects without traversing the entire heap.

Different garbage collectors implement these concepts differently and optimize for different application requirements. The HotSpot JVM provides several collectors, including the Serial, Parallel, G1, ZGC, Shenandoah, and Epsilon collectors [12, 19, 20]:

- **Serial collector**

This collector performs all its tasks on a single thread. No additional synchronization overhead is generated. It does not take advantage of multiprocessor hardware and is therefore suited for single-processor hardware

- **Parallel collector**

The parallel collector is similar to the Serial collector, but uses multiple threads, increasing garbage collection throughput.

- **G1 garbage collector**

G1 is a generational, region-based, mostly concurrent collector designed to balance throughput and predictable pause times [16]. It is the default garbage collector of the HotSpot VM and the target of this thesis.

- **Z garbage collector**

This collector is designed for fast garbage collections with pause times under a millisecond, suited for low-latency applications.

- **Shenandoah garbage collector**

Shenandoah works similarly to the Z garbage collector. It also focuses on low pause times by doing most of the work concurrently with the application.

- **Epsilon collector**

A non-operational garbage collector. It handles memory allocation but does not reclaim memory. It is primarily used for testing and is not designed for production.

Consequently, the different collectors represent different trade-offs between throughput, latency, memory footprint, and implementation overhead. The choice of garbage collector depends on the application's characteristics and requirements. For example, the Parallel collector suits applications where throughput is the primary objective, whereas low-latency collectors such as ZGC and Shenandoah are designed for applications where long garbage collection pauses are

particularly undesirable. G1 aims to balance these objectives and is the focus of the following sections [21].

2.3 Garbage-First Garbage Collector

The Garbage-First (G1) garbage collector [22] is a high-performance implementation of a garbage collector and has been the default garbage collector in the OpenJDK HotSpot Java Virtual Machine (JVM) since JDK 9 [16]. G1 is designed to perform well on systems with multiple processors and large heaps, and it aims to balance application throughput with predictable garbage collection pause times. G1 considers pause-time targets when deciding how much memory to reclaim during individual collection pauses. It is based on splitting the heap into regions, concurrent marking, remembered sets, and incremental evacuation.

Regions are dynamically assigned to either the young generation or old generation, as part of G1's generational garbage collection approach. Regions belonging to a generation therefore do not form a physically contiguous block in the heap and can be distributed across the entire heap. This regional concept allows G1 to select individual regions for collections instead of, for example, collecting the entire old generation at once. The subset of regions selected for reclamation during garbage collection is called the collection set. The regions in the collection set are evacuated during the corresponding garbage collection pause. G1 preferentially selects old generation regions with high collection efficiency, primarily determined by their reclaimable space and connectivity to other regions [16]. They are “collected first”, which gives rise to the name “Garbage-First” collector. This enables incremental space reclamation over multiple collection pauses, avoiding the need to reclaim the entire old generation in a single pause.

G1 combines concurrent phases with Stop-The-World (STW) pauses. During STW pauses, the application thread stops, and the garbage collector performs its operation, for example, copying live objects from the collection sets into other regions. This evacuation process runs in parallel on garbage collector threads. Concurrent to the running application, G1 periodically performs a concurrent marking cycle, which detects which objects inside the old region are still alive and marks them. The marking information identifies regions suitable for the collection set.

The reclamation process also needs a mechanism called the remembered set. Because objects

can reference other objects in different heap regions, evacuating a region requires metadata about where those objects are referenced. Those references need to be changed, since they should point to the object's new position after evacuation. The remembered set provides this information. This data structure contains information about where potential references from outside a region are located. The complete heap is divided into cards, and write barriers perform work when references are changing. With these concepts, G1 can build the remembered set. So instead of traversing the entire heap to find cross-references, G1 can identify and find the correct references much faster. Maintaining remembered sets adds computational overhead, so G1 performs parts of the task concurrently and refines marked references when possible.

The selection of regions for the collection set depends on the kind of garbage collection performed. Regions which are part of the young generation are always in the collection set. Collecting them is called a young garbage collection. In addition, when G1 identifies sparse regions in the old generation, it also adds those regions to the collection set and collects them in a so-called mixed garbage collection. This incremental approach allows G1 to reclaim objects from the old regions over multiple mixed collections rather than performing a full garbage collection, where the entire heap is compacted.

The following describes this mechanism in more detail.

2.3.1 Design of Regions

G1 is a regional garbage collector. Regions are equally sized subdivisions of the Java heap. Figure 2.1 shows an example layout of a Java heap. Regions are either free  or one of four different types: *Eden* , *Survivor* , *Old* , *Humongous* . A used region is either part of the young generation or the old generation. So, instead of dealing with one large memory block per generation, G1 operates on regions to gain more control over memory usage and the young-to-old generation ratio.

- **Eden Region**

Eden region is part of the young generation. All newly allocated objects are inserted into this region, except humongous objects. During garbage collection, the system evacuates all live objects in the eden region into survivor regions, leaving the eden region empty.

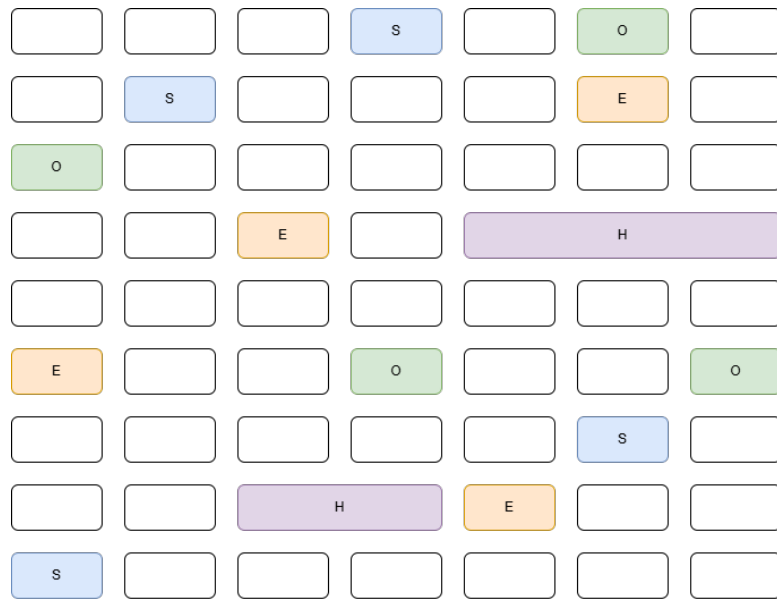


Figure 2.1: Example Heap Layout in G1

- **Survivor Region**

As the name suggests, the survivor region contains objects that survived the first garbage collection of young objects in eden. It is also part of the young generation. If objects survive long enough, they are eventually promoted to the old generation.

- **Old Region**

Old regions contain objects in the old generation. Objects are promoted to these regions if they survive a certain number of garbage collections. During a garbage collection of the young generation, the collector considers old objects alive and therefore ignores them.

- **Humongous Region**

Humongous objects are objects larger than half a region's size. In case an object is larger than a region, it needs multiple contiguous regions, all labeled as humongous.

- **Free Region**

Free regions are currently not assigned to any generation and do not contain allocated objects. They form the pool of available regions from which G1 obtains memory for new allocations and evacuation.

All regions have three pointers: **bottom**, **end**, and **top**. **Bottom** points to the beginning of a region and **end** points to the end of a region. The pointer **top** indicates where new objects can

be allocated. It moves toward `end` after every allocation. Between `bottom` and `top`, the region is assumed to be *parsable*, meaning the range between consists of valid, contiguous objects.

2.3.2 Allocation

In G1, allocation is done on different threads in parallel. Suppose a Java application thread, called `mutator`, wants to allocate an object. G1 selects an empty region from the free list and marks it as the active eden region. All allocation requests are tried to be satisfied by this region. Since multiple mutator threads could race on the same memory, the heap needs to be synchronized across threads. But synchronizing for every allocation is expensive and would take too much time. Therefore, instead of requesting memory just fitting for the object, the mutator thread requests a larger area, a Thread-Local Allocation Buffer (TLAB). Every thread owns its own TLAB and can allocate into it without synchronization. Allocation requires only a pointer increment and a bounds check to ensure the remaining TLAB size is large enough for the object. When the TLAB is exhausted, G1 requests a new TLAB from the allocator. If the active eden region can no longer provide the requested TLAB sizes, the allocator tries to allocate the object directly to use as much available space as possible. If this path eventually fails, the region retires, and the allocator selects a new, empty region. If a humongous object is allocated, the allocation follows a different path, where multiple consecutive regions may be used. This process continues until a certain threshold of filled eden regions is met. To provide new space, G1 triggers a young garbage collection, where all live objects from all eden regions are evacuated into survivor regions.

During a garbage collection cycle, G1 uses a similar allocation technique. Instead of having TLABs, the garbage collection worker thread uses PLABs (Promotion Local Allocation Buffer). PLABs solve the same synchronization problem as TLABs did. When multiple worker threads are copying live objects into the active survivor region, instead of synchronizing on allocation, each worker allocates into its own PLAB.

2.3.3 Concurrent Cycle

When G1 determines that space reclamation in the old generation is required, it performs concurrent marking (CM) to determine the number of live objects in the old regions. During CM, G1 traces the object graph, starting from a set of root references. Every object visited during traversal is marked as live. The resulting information is used to determine which regions are suitable candidates for reclamation [23].

Concurrent marking uses a *Snapshot At The Beginning* (SATB) algorithm [23]. The fundamental idea of SATB is to consider the object graph as it existed at the beginning of the marking cycle. Objects that are alive at the beginning are treated as live objects, even if they become unreachable while marking is in process. On the other hand, objects allocated after the snapshot are considered alive, even if they die during marking. Those objects need to be marked during the next concurrent marking. SATB used by G1 takes a virtual snapshot. For every region in the snapshot, G1 remembers the current value of the **top** pointer of the region in a **TAMS** (Top-At-Mark-Start) pointer. The **top** pointer marks the end of the last allocated memory area in a region. The memory between **bottom** and **TAMS** marks the part that belongs to the snapshot. When an object is allocated after the snapshot, **top** advances beyond **TAMS**. These newly allocated objects are now to the right of **TAMS**, outside the snapshot, and are implicitly considered live for the current marking cycle.

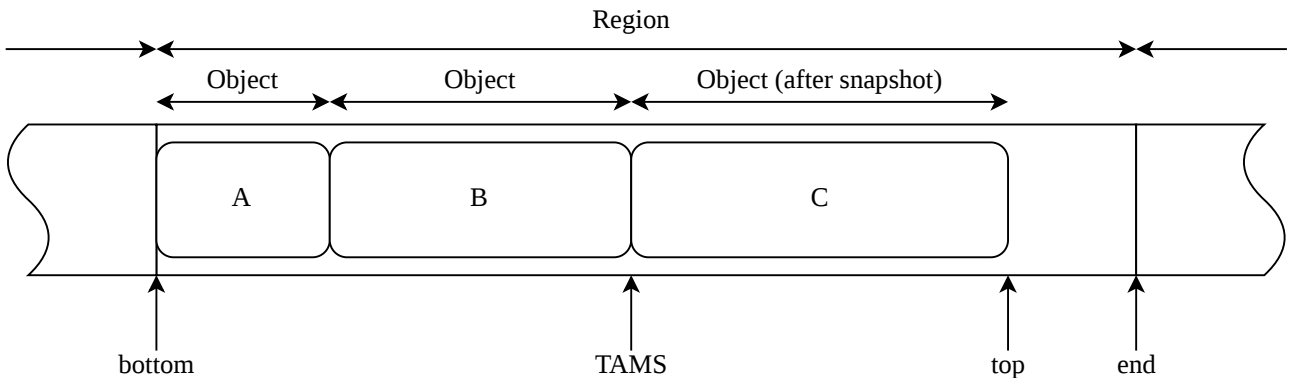


Figure 2.2: Example of SATB on a region. Object A and B are snapshotted, and **TAMS** was set to **top**. After SATB, object C was allocated, and **top** has moved

For example, assume objects A and B are already allocated when the concurrent cycle starts. Figure 2.2 shows the example region. G1 sets the **TAMS** to the current **top** value. Now, object C is allocated, moving the **top** pointer. Object C is therefore outside the snapshot and is

considered alive for this cycle, which does not necessarily mean that object C is still reachable. But G1 does not attempt to reclaim C based on the current snapshot. G1 can reclaim object C only in the next concurrent marking cycle.

The SATB invariant must be maintained while the application continues to modify the heap. To prevent reference changes during marking, G1 uses a *pre-write barrier*. Each time the application changes the value of a pointer during the current marking, it records the previous reference. The marking process treats those recorded references as root references. This ensures that an object that was part of the snapshot cannot be lost during marking simply because the application changed the reference that previously kept it reachable.

The concurrent cycle consists of several phases, from traversing the object graph to restoring the data structure after marking. The workflow is presented in Figure 2.3. The cycle starts with the *Concurrent Start* pause, which initializes marking. *Concurrent Clear Claimed Marks* performs some work that was left over by the previous phase. The following *Concurrent Scan Root Regions* phase scans the recorded root regions concurrently. This phase finds references from the young generation into the part of the old generation that belongs to the snapshot. Objects reachable through these references are marked on the bitmap. After scanning, the object graph is traced during *Concurrent Mark*. The actual marking is done on a bitmap. When an object is marked, the bit indicates the start of the object in the heap. Since an object may start at an arbitrary position in the heap, aligned by a `HeapWord` (8 bytes), one bit must represent 8 bytes of the heap. Thus, the bitmap requires approximately $\frac{1}{64}$ of the Java heap, or 1.5625% of its size. In *Concurrent Mark From Root*, starting from the roots, the object graph is marked. G1 uses the *tri-color marking algorithm* [24]. Objects are conceptually divided into three sets: *white*, *grey*, and *black*. White objects have not yet been discovered and may therefore be unreachable. Grey objects have been discovered as reachable but have not yet been fully scanned for references to other objects. Black objects have been fully scanned, meaning that all objects reachable through their references have also been discovered. At the beginning of marking, all objects are white. The collector starts from a set of root references and marks the objects they reference as grey. A grey object is then selected and scanned. After all references of the object have been processed, the object is considered black, while newly discovered objects are added to the grey set. This process continues until no grey objects remain. At this point, all reachable objects are black, while the remaining white objects are unreachable and can be reclaimed. If a reference points to an object within the snapshot area, G1 attempts to mark the referenced object. In addition to tracing the object graph, G1 calculates the amount of

live data in each region. G1 later uses this information to determine which regions from the old generation are suitable candidates for evacuation. In the *Concurrent Preclean* phase, some special objects are processed for what G1 can determine, whether further processing is required in *Remark* or not. *Remark* completes marking and initializes rebuilding the remembered sets. In rare cases, the marking process can be restarted until a fix point is reached. When marking is finally complete, *Concurrent Rebuild Remembered Sets and Scrub Regions* is called. For the selected candidates for the collection, the remembered set is rebuilt by scanning live objects and recording those that point into the region. Additionally, the area of all dead objects is scrubbed with filler objects. Those objects make the region parsable again since someone can skip them by an offset of the filler size. Finally, the *Cleanup* pause completes the preparation for old generation reclamation. G1 refines the set of candidate regions and removes regions it considers inefficient to collect. The following *Concurrent Cleanup for Next Mark* phase clears the marking bitmap and resets the TAMS pointers, preparing the data structures for the next concurrent marking cycle. With this phase completed, the concurrent marking cycle has finished.

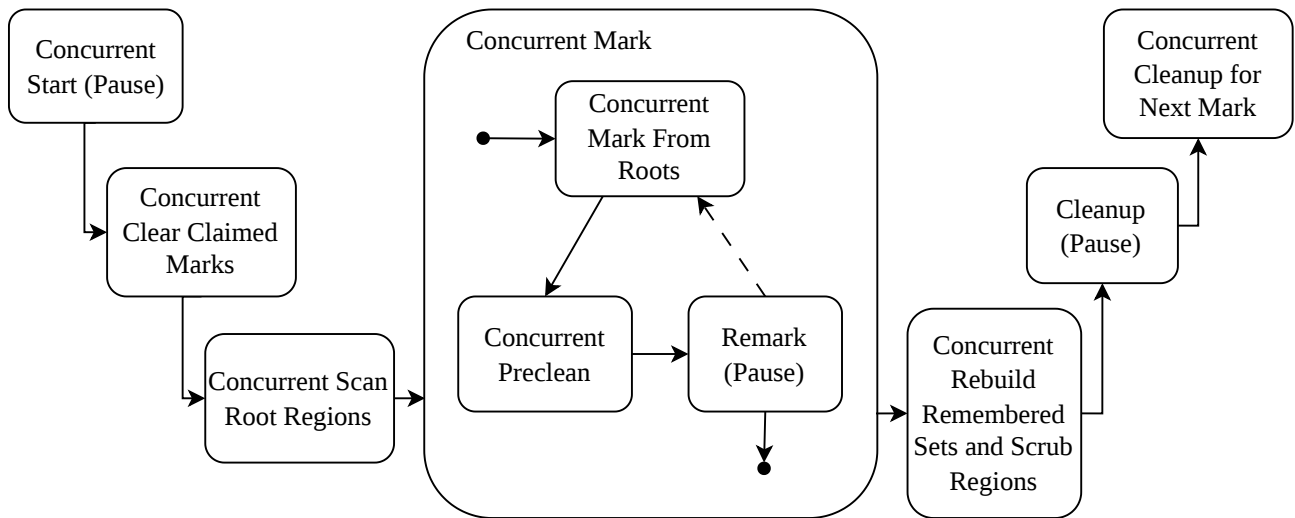


Figure 2.3: Flow chart of concurrent marking phases

2.3.4 Remembered Sets

Region-based collection introduces an important problem. Objects in one region can reference objects in a different region. In case an object is collected, for example, through evacuation into another region, the garbage collector must identify all references to this object that are not part

of the collection set, since these references need to be updated to the object's new location [25]. Scanning all regions outside of the collection set would be computationally expensive, as the collection set is assumed to consist of a small subset of the entire heap [26]. G1 solves this issue by using remembered sets. To identify potentially interesting references, G1 uses a card table.

G1 uses a card table to detect heap memory modifications. The Java heap is divided into fixed-size cards. One card represents a part of the heap, typically 512 bytes [27]. The card table holds those cards in a byte array, where each entry contains a card. The table does not store the references directly. Instead, a card marks an area where a potentially interesting reference for garbage collection is located. An interesting reference is a reference from an area of the heap that will not be collected to an area that will be collected. The write barrier consists of additional code the application executes when it writes a reference. References within the same region do not need to be recorded, since they cannot create references between different regions. Similarly, assignments of `null` do not create references between objects and therefore do not need to mark the corresponding card. A card that is already marked does not need to be marked again, since it is already scheduled for processing.

Refinement is the concurrent process in which G1 re-examines and reclassifies cards previously marked by the application. When the write barrier marks a card, it places the card's location in a refinement buffer so refinement threads can process it later. A refinement thread examines the corresponding area of the heap and determines whether the card currently contains interesting references. If it finds an interesting reference, the card adds the referenced object's region to its remembered set. Since the contents of a card can change between the original write and its refinement, a card that was marked by the application may no longer contain an interesting reference when it is examined. Refinement therefore avoids unnecessary scanning during a subsequent garbage collection pause by removing obsolete card marks and classifying the remaining ones according to their target regions.

Remembered sets store cards of special interest for collection. In G1, each region has its own remembered set, since it can collect arbitrary subsets of regions. A region's remembered set stores cards that may contain references from outside the region into the region. Thus, after evacuating objects from a region, instead of traversing the entire heap to check whether any reference to an evacuated object must be updated, G1 only needs to examine the cards contained in the remembered set of the evacuated region.

2.3.5 Eager Reclaim of Humongous Objects

Normally, G1 determines the liveness of humongous objects during concurrent marking and reclaims unreachable objects during the subsequent cleanup phase. G1 additionally supports *eager reclamation*, which attempts to identify dead humongous objects during a young garbage collection and reclaim their regions immediately. This allows the regions occupied by such objects to be reused without waiting for the next concurrent marking cycle. Eager reclamation is an optimization that lets G1 reclaim some humongous objects before the regular concurrent marking cycle identifies them as dead [28]. But since not all humongous objects can be reclaimed like that, concurrent marking is still needed for humongous objects.

2.3.6 Historic Differences of the G1 Garbage Collector

Detlefs et al. [22] first presented the Garbage-First (G1) garbage collector. Since its original publication, G1 has undergone substantial changes. While its fundamental region-based heap organization and incremental evacuation strategy remain central to the current collector, several mechanisms described in the original paper have been replaced, significantly revised, or removed altogether. The following sections highlight the most important differences between the original G1 design and its current implementation.

1. Heap organization and allocation

G1's heap organization has remained largely unchanged since the original design. The Java heap is divided into equally sized regions, and humongous objects are allocated across multiple contiguous regions. Similarly, general allocation still relies on regions and allocation buffers (TLAB, PLAB).

2. Remembered Sets

The purpose of remembered sets remained the same. In the original design, every worker thread tracks a bitmap at the card level for every target region. The original G1 paper proposed grouping regions by *affinity*, based on how strongly regions reference each other [22]. However, this was never implemented. Instead, modern G1 has progressively introduced mechanisms to avoid maintaining unnecessary remembered set information. The remembered set data structure was also redesigned as a fully lock-free, adaptive

structure that selects an appropriate representation based on the number of cards stored, reducing memory overhead. Furthermore, G1 does not continuously track remembered sets for all regions. Through remembered set scrubbing and rebuilding during concurrent marking, G1 selectively maintains remembered sets only for groups worth tracking [29, 30]. These changes shift from continuously maintained, per-region remembered sets to selectively maintained, more compact representations. The design is conceptually related to adaptive bitmap representations such as Roaring bitmaps [31], but G1 uses its own data structure.

3. Write Barriers

The *pre-write* barrier remains essentially the same as in the original paper. However, the *post-write* barrier changed completely. The original G1 post barrier synchronously marked a shared card table and enqueued newly dirty cards for concurrent refinement. The new design uses two independently written card tables that periodically swap roles, so the barrier only performs filtering and a card mark. At the same time, refinement discovers marked cards by sweeping the inactive table [32, 33, 23].

4. When to Start a Concurrent Cycle

The old G1 used a static threshold for old generation occupancy. When the threshold was exceeded, it triggered a concurrent cycle. Now, the threshold is adaptive based on marking duration, allocation needs, and humongous object pressure. Marking is started early enough to produce old candidates before allocation pressure becomes problematic.

5. Marking

The TAMS pointer serves the same purpose as in the original paper, but we reduced the number of TAMS to one per region. Evacuation during concurrent marking still cooperates with concurrent marking in essentially the same way. Still, modern G1 reduces the cost of determining object liveness by representing evacuated holes as filler objects, as mentioned in Section 2.3.3.

6. Hot-Card Handling

The original G1 design attempted to identify cards that were modified particularly frequently and treat them specially, so called *Hot-Cards* [22, 34]. Later measurements did not show sufficient benefit in the context of the other optimizations introduced into G1. Consequently, this mechanism was removed.

7. Popular Objects

The original paper introduced the concept of *popular objects*. Popular objects are objects which are referenced from many locations. A special portion of the heap, which will not be considered to be part of the collection set, was reserved for those objects. Therefore, the location, and so the reference, remains stable. G1 now handles such objects implicitly. Popular objects are by definition very costly, so they will not be selected for remembered set building. No explicit handling is done anymore.

8. Non-Generational Mode

The original paper describes G1 operating in a *non-generational mode*. Modern G1 no longer provides this mode and is nowadays a pure generational collector. Zhao and Blackburn support this decision, as their measurement shows a significant reduction of GC pauses using generational mode [34].

9. Full Collections

Compared with the original collector, modern G1 has a largely parallel full-collection implementation, improving GC performance. This is not really a change. It improves the original implementation.

10. Class and Code Unloading

Class unloading was not described as part of the original G1 design. Early G1 could perform class unloading during full GC, while later versions integrated it into concurrent marking [35].

11. JNI and Pinned Objects

The treatment of native code has also changed. Originally, the G1 implementation used a global *GCLocker* mechanism to prevent garbage collection on regions while native code could potentially hold references into the Java heap. Modern G1 instead supports *region pinning* [36]. Regions containing pinned objects that must remain at stable addresses can be pinned while other regions can meanwhile be collected.

2.4 Data Structure for Managing Holes

This chapter briefly describes the data structures used in this thesis to efficiently store and retrieve holes.

This thesis uses the basic concept of single-linked lists, as described in the literature [37]. A list consists of nodes, where each node has an **element** and a **next** field. The **element** field stores the content of the node, and the **next** field points to the next node in the list. The first element of a list is called **head** and is the access point to the list, and therefore the only node you need to remember when storing a list. Starting at the **head**, you can traverse the list by following the **next** field. Inserting in a list is fast. To insert a new node at the beginning, the new node becomes the **head** and its **next** pointer points to the previous **head**. This is done in $\mathcal{O}(1)$ time complexity. Deletion works similarly: we bend the **next** pointer around the node we want to delete so it points to the next node.

The thesis also uses the concept of a binary tree (BT). A binary tree is described as a directed graph where each node has at most two successor nodes, called children, and at most one predecessor, called the parent. The only node with no parent is called the **root** node and is the entry point to the data structure. A tree node therefore has an **element** field for the content, a **parent** field, and two fields, **left** and **right**, for the children. A binary search tree (BST) is a binary tree with a special ordering property based on each node's value. During insertion, when the new node's value is less than or equal to the current node's value, insert the node in the left subtree. If it is greater, insert it in the right subtree. This ordering ensures that every value in a node's left subtree is smaller than the node's value, and every value in its right subtree is greater. During insertion, we traverse through the tree until we find a node where the needed child is **null**. In the worst case, we traverse the full height of the tree, h , leading to a time complexity of $\mathcal{O}(h)$.

A red-black tree (RBT) is a self-balancing binary search tree that assigns each node a color, either red or black. In addition to the ordering property of a BST, the nodes must satisfy a set of color and structural properties that limit the height h of the tree. In particular, every path from the root to a leaf contains the same number of black nodes, and a red node cannot have a red child. These properties ensure that the height of the tree remains logarithmic in the number of nodes, resulting in $\mathcal{O}(\log n)$ time complexity for searching, insertion, and deletion. After an insertion or deletion, the tree may violate its balancing properties and is therefore

rebalanced using recoloring and tree rotations.

In the context of RBTs, AVL trees should also be considered, as they are frequently compared with red-black trees in the literature [38]. Although AVL trees provide stricter balancing, red-black trees generally require fewer rotations, making them better suited for workloads involving frequent insertions and deletions. This characteristic makes red-black trees a potentially better fit for the hole structure considered in this project. Moreover, the HotSpot VM already provides a red-black tree implementation that could be adapted for this project.

Chapter 3

Problem Statement: Holes in Regions Are Underutilized

This chapter addresses low-level insights into the problem: holes can occur at the end of regions as well as inside them. As already mentioned above, the core concept is to make this unused memory available to the running application. In detail, three different kinds of problems can create holes. The first problem is called the *Survivor Allocation Holes*. Here, a hole can occur at the end of a survivor region. Similarly, the *End of Humongous Region Holes* occurs when humongous regions can also have unused memory at the end of the region. The third problem is in old regions and is called *Fragmentation of Old Region*. Inside an old region, objects can be marked as dead by the garbage collector and therefore freed. The following gives a more detailed description of the three different problems.

3.1 Survivor Allocation Holes

Survivor regions, together with the eden region, are part of the young generation. Conceptually, only young objects should live in young regions. During a young GC, live objects from the eden regions get copied into survivor regions. This evacuation process is done in parallel on multiple threads. As described above, to avoid multithreading conflicts, each thread is assigned a PLAB to allocate memory into. Figure 3.1 shows an example of evacuation. Three different threads have their own PLAB regions. When a thread requests another PLAB, and either a fourth thread exists or the thread's currently assigned PLAB is full, the request fails because there

is not enough space. In that case, the current active survivor region is retired. During this process, a new, empty survivor region is assigned as the new active region. Then the PLAB is allocated into the new region. As long as the PLABs of the retired region have enough space for the allocation request, the objects of those threads are still allocated into the retired region. Still, at some point, all PLABs are full and allocated into a new region. The retired region is no longer available for allocation, so the unused space at the end of the region, illustrated in Figure 3.1, is wasted and not visible to the allocator. The goal is to provide this hole so the allocator can assign the unused memory to an object of a fitting size. Those holes should be available to the mutator thread so that the memory can be used more efficiently.

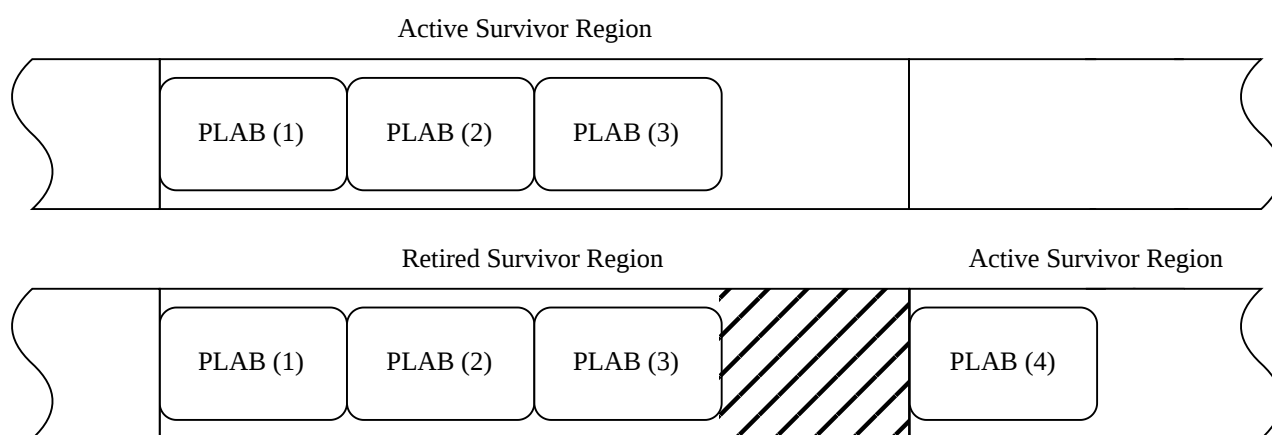


Figure 3.1: Allocation into active survivor region during garbage collection

For completeness, when there is not enough space for an entire PLAB but there is enough space to allocate the object, the region provides memory instead of retiring the active region, to use as much memory as possible. The same technique is applied to old regions.

3.2 End of Humongous Region Holes

For humongous objects, a similar problem occurs. When a humongous object is allocated, it receives its own humongous region(s). Naturally, there will be a gap at the end of the last region since it is unlikely that the object uses an exact multiple of region sizes. Figure 3.2 shows such a case.

From a practical point of view, this is indeed a real problem. It is common in practice to

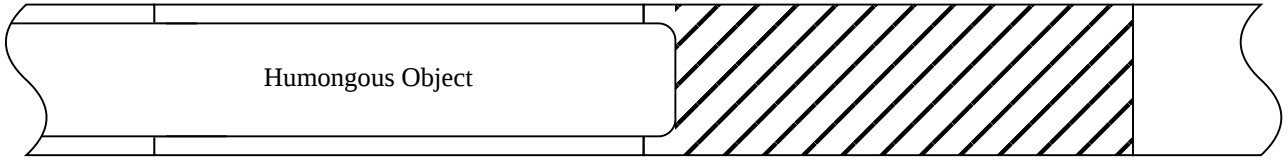


Figure 3.2: Unused memory at the end of the last region of a humongous object

allocate large objects of payload 2^n , which would be perfectly fit into 2^{n-m} regions of size 2^m [39]. But due to the object header at the beginning of each object, which is a couple of bytes, this would imply that the object would dip slightly into the next region, using $2^{n-m} + 1$ regions, where the last region is basically completely unused, as illustrated in Figure 3.2. By design, no other objects can be allocated into an occupied region, as G1 assumes that only one object exists in humongous regions. Making this tail available for allocation to old objects would increase available space in the old generation and improve heap-usage efficiency. This would make humongous regions a hybrid type, where both kinds of old objects can exist in the same region. By doing so, we violate assumptions currently made by G1, which need to be updated.

3.3 Fragmentation of Old Region

The *Fragmentation of Old Region* problem generalizes the two problems above. As described for *Survivor Allocation Hole*, we get a hole at the end of old regions when the active old region retires. But during the liveness analysis, objects can now be marked as dead, leaving a hole inside the region behind, as shown in Figure 3.3. This leaves the region fragmented, where the holes are not visible to the allocator. The unused space can only be reclaimed by evacuating the live objects into another region and defragmenting the region in the process. Instead, we could remember the objects we mark during the concurrent marking phase and present them to the allocator if memory is requested.

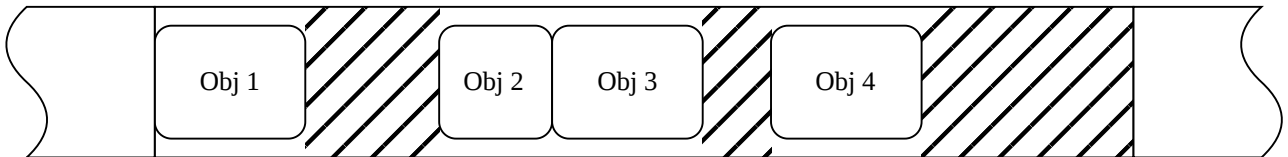


Figure 3.3: Fragmentation of old region

Chapter 4

Implementation: Utilizing Holes in G1 Regions for Allocations

In this chapter, we explain the implementation details for this thesis. We need to find and remember the location in the existing codebase where holes occur. The task is also to investigate the existing allocation path and identify the correct place to store remembered holes. For that, we use different approaches. How holes are remembered is crucial, depending on which criteria we want to optimize. To investigate whether, for example, fast insertion or fast hole search is more important, we test the project on different benchmarks to determine the best approach. We will present and evaluate approaches for remembering holes in the heap and finding them for allocation. Since we extend the existing implementation and change existing assumptions, we also need to address those changes. We also explain the problems that arise in detail. In addition, we describe alternative approaches and benchmark them against the chosen solution.

4.1 Integration into the Allocation Path

This section provides an overview of how, in the current allocation mechanism, the different holes described in the problem statement can be collected. To collect the holes in the young generation, the method shown in Listing 4.1 is used. For humongous and old holes, the method is conceptually the same. The method expects a pointer `region` pointing to the region the hole is part of, a `word` pointer indicating the start address of the hole, and `size_in_words` of the

```

1  bool G1RegionFreeSpaceTracker::add_potential_survivor_hole(
2      G1HeapRegion* region, HeapWord* word, size_t size_in_words) {
3      if (size_in_words < _min_hole_size_young) {
4          return false;
5      }
6      MutexLocker x(G1YoungDataStructure_lock);
7      add_hole_list(_holes_young, region, word, size_in_words);
8      if (use_tree()) { add_hole_tree(_root_young, word, size_in_words); }
9      return true;
10 }

```

Listing 4.1: Implemented method for inserting in holes in data structure

hole. In line 3, the hole gets rejected if it is smaller than the minimal hole size. In line 6, the data structures are synchronized by holding a lock during the insertion. The hole is inserted into the list structure in line 7 and, when the tree is enabled, also into the tree structure in line 8.

4.1.1 Collecting Young Holes

As described in the problem statement, holes at the end of survivor regions occur when the active region retires. G1 calls the `retire` method in case no further allocation is possible. The method is shown in Listing 4.2. In line 5, the region gets retired. During this process, we assign a new `_alloc_region`. Since the current active region is needed to compute the hole size in line 7, the region is stored in line 3. During retirement, the `top` pointer is also moved, which marks the start of the hole. G1 provides the `top` pointer before the retirement, which is accessed via `pre_dummy_top()`. The region, the start of the hole, and the hole size are passed to the `add_potential_survivor_hole` method in line 9.

4.1.2 Collecting Holes at the End of Humongous Objects

Humongous objects have special allocation paths. Even when newly allocated, they are assigned directly to regions in the old generation. Because they use multiple consecutive regions, additional metadata must be set, like a reference to the region where the humongous object starts. This is done in the `set_humongous_metadata` method. At the end of the last region,

```

1  size_t G1AllocRegion::retire(bool fill_up) {
2      ...
3      G1HeapRegion* alloc_region = _alloc_region;
4      if (alloc_region != _dummy_region) {
5          waste = retire_internal(alloc_region, fill_up);
6          if (G1UseSurvivorHoles && alloc_region->is_survivor()) {
7              actual_waste = pointer_delta(alloc_region->end(),
8              alloc_region->pre_dummy_top());
9              _g1h->add_potential_survivor_hole(alloc_region,
10              alloc_region->pre_dummy_top(), actual_waste);
11          }
12          reset_alloc_region();
13      }
14      ...
15  }

```

Listing 4.2: Collecting young holes during retirement

it inserts a filler object. All that needs to be done is to pass the last region, the end of the humongous object, and the size of the filler object to the `add_potential_humongous_hole` method.

4.1.3 Collecting Old Holes

Fragmentation in old regions is currently handled by concurrent marking. The implemented logic is shown in Listing 4.3. In line 5, the object is skipped, whereas in line 8, dead objects are filled with filler objects. Those filler objects are added to the `add_potential_old_hole` method in line 10.

4.1.4 Path of Returning a Fitting Hole

Complementary to finding the correct place to insert a hole, we also need to find a suitable place for a hole and return its address to the allocator. For young objects, we follow the allocation path. The allocator tries to provide memory from retained regions. If this fails, the allocator tries a second attempt to provide memory from the active allocation region. If this attempt also fails, the allocator attempts the allocation on the slow path. In Listing 4.4 some parts of

```

1 HeapWord* scan_or_scrub(G1HeapRegion* hr, HeapWord* addr,
2   HeapWord* limit){
3   if (_bitmap->is_marked(addr) ||
4       _cm->check_left_allocated_objects(addr)) {
5       return addr + scan_object(hr, addr);
6   } else {
7       HeapWord* scrub_end = _bitmap->get_next_marked_addr(addr, limit);
8       hr->fill_range_with_dead_objects(addr, scrub_end);
9       if (hr->is_continues_humongous() || hr->is_old()) {
10          G1CollectedHeap::heap()->add_potential_old_hole(hr, addr,
11              pointer_delta(scrub_end, addr));
12      }
13      return scrub_end;
14  }
15 }

```

Listing 4.3: Collecting old holes during scan or scrub

`attempt_allocation_slow` is shown. In line 8 we get a mutex lock. The next step is to start an allocation attempt in line 13, where the attempt to open a new region is made. But before attempting this, the allocator asks the data structure whether a suitable hole is available in line 10. If it fails, a new region opens anyway. When there is not enough space for a new region, in 16 a collection pause is scheduled.

For old objects, we access the `old_attempt_allocation` method, shown in Listing 4.5. This method attempts to allocate memory for an old object during garbage collection. In line 5, the active region is asked for memory. If the allocation has failed, we try to satisfy the request by finding a fitting hole in line 11. When this also fails, we fall back to the old allocation path in line 14.

4.2 Remembering and Finding Holes

Remembering and providing holes is the core of this thesis. Since holes are, by definition, unused memory, we can use that space to store the fields needed to build the data structure inside the heap. With this approach, we do not need to allocate additional memory because we reuse the existing heap. But we must consider the minimum hole size. When the size of a hole is smaller than the sum of all necessary fields, the hole can not be remembered since

```
1 HeapWord* G1CollectedHeap::attempt_allocation_slow(uint node_index,
2   size_t min_word_size,
3   size_t word_size,
4   size_t* actual_word_size) {
5   ...
6   for (uint try_count = 1; /* we'll return */; try_count++) {
7     {
8       MutexLocker x(Heap_lock);
9
10      result=find_young_hole(min_word_size, word_size, actual_word_size);
11      if (result != nullptr) { return result; }
12    }
13    result=_allocator->attempt_allocation_locked(node_index, word_size);
14    ...
15  }
16  result = do_collection_pause(...);
17  ...
18  return nullptr;
19 }
```

Listing 4.4: Slow allocation attempt for young objects

```
1 HeapWord* G1Allocator::old_attempt_allocation(size_t min_word_size,
2   size_t desired_word_size,
3   size_t* actual_word_size) {
4   ...
5   HeapWord* result = old_gc_alloc_region()->attempt_allocation(
6     min_word_size,
7     desired_word_size,
8     actual_word_size);
9   if (result == nullptr && !old_is_full()) {
10     ...
11     result = _g1h->find_old_hole(min_word_size,desired_word_size,
12       actual_word_size);
13     if (result == nullptr && !old_is_full()) {
14       result = old_gc_alloc_region()->attempt_allocation_locked(
15         min_word_size,
16         desired_word_size,
17         actual_word_size);
18       ...
19     }
20     return result;
21 }
```

Listing 4.5: Allocation attempt of old object during garbage collection

there is too little space. In practice, this does not matter, since the minimum hole size is only a couple of bytes, depending on the fields needed per approach, and holes of that size are not relevant and not worth remembering. The Java heap currently fills holes with filler objects. This is necessary because HotSpot requires every committed heap region to be parsable from bottom to top. With these objects, the gap can be skipped during traversal, since the object size is encoded in the header. Therefore, this header must remain untouched and counts toward our minimum hole size. After the header, we can store information freely as long as we do not exceed the filler object size. The following describes the different data structures.

4.2.1 Storing Holes in a Balanced Binary Search Tree

A binary search tree (BST) is a widely used data structure in computer science [40]. The structure makes it possible to search for specific values efficiently. To find the value 5 in the example shown in Figure 4.1, we start at the root. Since 4 is smaller than 5, we continue searching on the right branch. Since 6 is greater than 5, the search continues on the left branch, where we successfully find the correct node. Because of the structure of binary trees, the search space is cut in half at each branch, assuming the tree is balanced, which makes searching elements very fast.

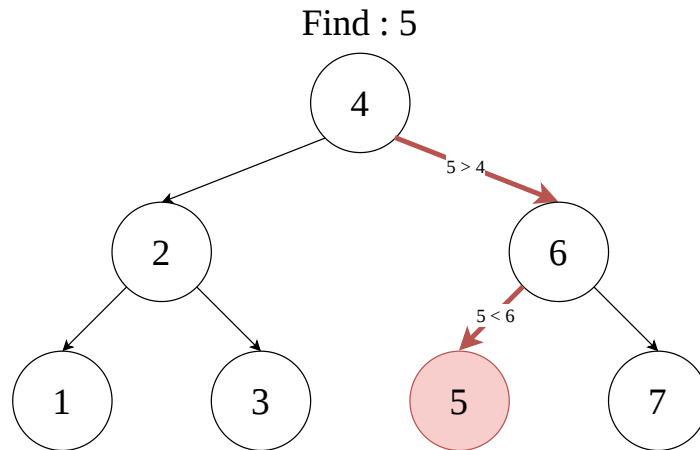


Figure 4.1: Example of searching for the number 5 in a binary tree

To provide fitting holes, it is not necessary to find a hole that fits exactly, but one that is at least as large as the minimum required hole size. To find the best-fitting hole, we define the *fit* function to find the smallest hole that fits.

Definition 4.2.1 (Fit function). $fit(x)$ is defined as the smallest hole that is larger than or equal to x .

$$fit(x) = \min\{h \in H | x \leq |h|\}$$

where H is the set of all holes and $|h|$ denotes the size of hole $h \in H$.

Binary trees naturally provide an efficient algorithm to find the smallest node greater than or equal to x . Listing 4.6 shows the code. We start at the root node and start traversing the tree. As long as the current node `curr` is not `null`, we compare the node key with the hole size. In line 7, when the hole size is exactly the minimum size `min_size`, we return immediately since we can not find a better hole. If not, we check in line 9 if the size of `curr` is smaller than `min_size`. If so, we continue searching for larger holes in the right subtree. In the other case, we check in the lines 12-13 two conditions: The region of the current hole should not be part of the collection set, since we should not provide a hole that the garbage collector will collect soon. The second condition checks whether the hole is splittable, meaning the remaining part of the found hole is large enough to fit a filler object. If not, we cannot use this hole, as the remaining part is not parsable. If both conditions are met, we store in line 14 `curr` as the best hole found so far. Since the hole size of all successors in the right subtree of `curr` is greater, we can ignore the subtree, as we can not find a better hole than the current best. However, we must traverse the left subtree if a hole smaller than `curr` exists but is still greater than `min_hole`, so we assign the left child in line 16. We continue traversing the tree until we reach a leaf node with `null` as its successor child and return `best_fitting_hole` in line 19. Therefore, we return either the best fitting hole in the tree for `min_size`, or `null` if there does not exist a hole greater than or equal to key.

So in the case of a balanced tree, we perform this algorithm in $\mathcal{O}(\log(n))$. This assumes the tree is indeed balanced, which we do not guarantee at any point. In case we inserted by chance holes in ascending order, the tree structure degenerates conceptually to a linked list, where each node only has a right child, which leads to a worst-case performance of $\mathcal{O}(n)$. To counter that, we extend the binary tree to a red-black tree (RB tree) [37]. This data structure assigns each node either the color red or black. The tree must fulfill specific rules based on those colors. If a rule is violated, we rebalance the tree to keep it balanced. By doing this, the height is $h = \log(n)$, where n is the number of nodes in the tree. Therefore, we cut our search space in half after every node, leading to an average search time of $\mathcal{O}(\log(n))$. The heap layout of the holes is shown in Figure 4.2. After the filler object header, we store the hole size. The `next` field is needed for the list structure, explained in the following section. The RBT requires four

```

1 HeapWord* find_best_fitting_hole(HeapWord* &root, size_t min_size) {
2     HeapWord* curr = root;
3     HeapWord* best_fitting_hole = nullptr;
4     while (curr != nullptr) {
5         size_t hole_size = get_size(curr);
6         if (hole_size == min_size) {
7             return curr;
8         }
9         if (hole_size < min_size) { curr = get_right(curr); }
10        else {
11            G1HeapRegion* region = _g1h->heap_region_containing(curr);
12            if (!region->in_collection_set() &&
13                isSplittable(min_size, hole_size)) {
14                best_fitting_hole = curr;
15            }
16            curr = get_left(curr);
17        }
18    }
19    return best_fitting_hole;
20 }

```

Listing 4.6: Algorithm for finding best fitting hole in BST

fields : `parent`, `left`, `right` and `color`. A root pointer refers to the entry of the tree. The pointers to the child nodes are stored inside the holes. By linking them together according to the RBT properties, we get an efficient structure for storing our holes.

4.2.2 Storing Holes in a List Structure

A simpler approach is to link all holes in one region together. For this, we allocate an array of size equal to the number of regions. Each array element points to the starting hole of the region or to `null` if a region has no holes. Each hole has a `size` field and a `next` field. The size stores the hole size, and the next pointer points to the next hole in the same region. The last hole points again to `null`. Figure 4.3 shows an example of this linking with two regions.

Inserting in a list is fast. Determining the correct list head is an array look-up. Inserting a new hole requires setting the `next` pointer to the current header and redirecting the header pointer of the array to the new hole, which is overall a constant $\mathcal{O}(1)$ operation. Deletion works the same way: bend the previous hole's `next` pointer to the successor hole, again in $\mathcal{O}(1)$. But

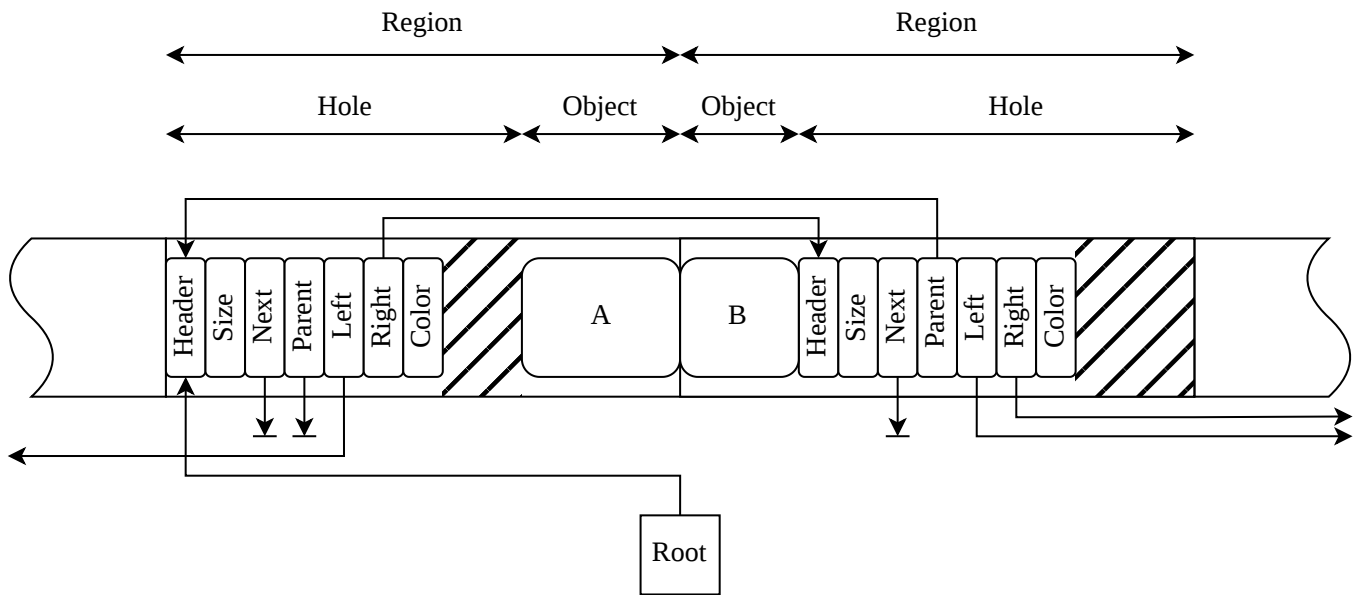


Figure 4.2: Linking holes together in a tree structure

searching a fitting hole is problematic.

We could also link all holes into one large linked list, so an array of head pointers to the first hole per region is not necessary. But by doing so, we lose information about which holes belong to which region. If a region is freed, we must remove all holes belonging to that region. A large linked list has to traverse all elements and decide for each hole whether it needs to be removed. In the per-region approach, we need to set the region's head pointer to `null`, which is much more efficient. Although we use a tree structure to remember the holes, the list is still necessary. If a region is freed, we need to remove all holes we remembered for that region. In a tree structure, we would only get this information by traversing the entire tree and checking every hole to see whether it belongs to the freed region. Instead, we also link all holes per region together, so we can traverse the list and remove each hole from the tree. Note that we need to distinguish between holes in the young generation and the old generation, so we need two separate lists and therefore two head-pointer arrays.

4.2.3 Alternative Structures for Holes

Using the linked list would be conceptually sufficient. All information about a hole, such as its size and location, can be found by traversing all lists in all regions. But to find the best-fitting

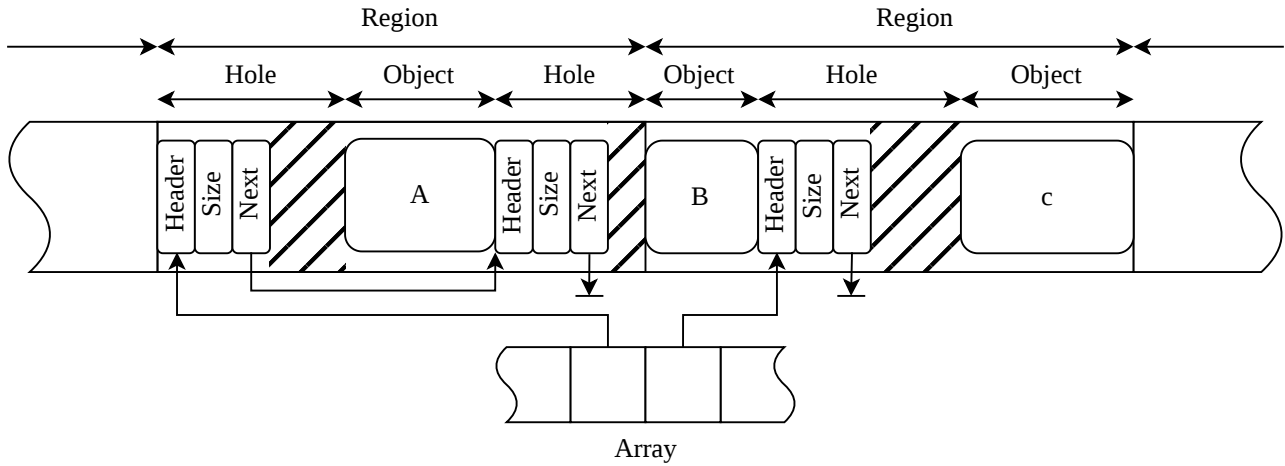


Figure 4.3: Linking holes together with a linked list

hole, we need to traverse all regions to be sure we found the best fit, since we do not insert holes in order. So we rely on this data structure with a First-Fit approach, returning the first hole big enough to fit the object. This not only wastes memory by leaving potentially very small, unusable space behind, but when no suitable hole exists, it is still, in the worst case, a $\mathcal{O}(n)$ operation, where n is the sum of all holes across all regions. By introducing structure and inserting holes by size, we could search more efficiently for a given size. But even if we do, we still need to traverse all linked lists if we want, for example, to find the largest hole in our list. So the worst-case time complexity remains $\mathcal{O}(n)$.

Instead of rebalancing the tree, we can use the binary search tree as it is. To compensate for the missing balance, we introduce a termination criterion. Instead of finding the best-fitting hole by traversing the entire tree, the search stops after k steps and returns the best hole found so far. While the found hole is not guaranteed to be the best-fitting hole anymore, performance is bounded by $\mathcal{O}(k)$.

4.3 Integration Into G1

Providing a new allocation path violates some assumptions of the existing implementation. Therefore, the new features, like old objects that can now occur after a humongous object, must be updated so HotSpot can handle them.

4.3.1 Card Alignment of the Tail Region

In theory, old objects can be allocated directly after the humongous object. But it is more convenient to align the beginning of the old tail region with the card table. This makes refinement and building remembered sets easier. To do so, a gap is computed after the humongous object exactly as large as the distance to the card table border, as shown in Figure 4.4. A filler object needs to be inserted into the gap. If the alignment to the next card is smaller than the minimum filler object size, a second card is added. If the remaining space is large enough, the `_old_objects_start` pointer is set to indicate where the tail region begins. The remaining space is collected as a hole. When old objects are allocated into the tail, `top` moves accordingly. So the region between `_old_objects_start` and `top` indicates the region of allocated old objects. The check of whether a humongous region has an old object tail region is done by checking `_old_objects_start != top`.

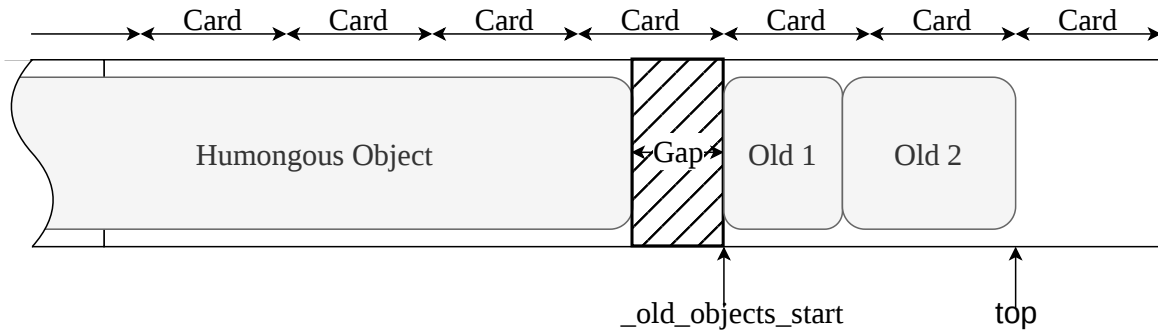


Figure 4.4: Aligning the old region after a humongous object with the card table

4.3.2 Termination Condition of Concurrent Marking

During concurrent marking, G1 uses the SATB algorithm. Objects allocated after the snapshot are considered live and are not marked. The check is a simple address check, since objects can only be allocated at an address higher than the `TAMS` pointer. Now, because of the additional path that uses holes for allocation, a new object may be allocated during concurrent marking, which violates the SATB invariant. To address this, we need to remember which regions of memory addresses our hole data structure provides. If marking encounters an object whose start address falls within such a region, we skip the object since SATB considers it live. To remember a region of addresses, we store it in a red-black tree. Since we assume no regions

```

1 bool G1ConcurrentMark::check_left_allocated_objects(HeapWord* word) {
2     RBNode<HeapWord*, MemRegion>* node =
3         _left_allocated_objects_tree.closest_leq(word);
4     return node != nullptr && node->val().contains(word);
5 }

```

Listing 4.7: Checking if an address is inside a requested hole used for allocation

```

1 void scan_and_scrub_humongous_region(G1HeapRegion* hr,
2                                     HeapWord* const pb) {
3     ...
4     oop humongous = cast_to_oop(hr->humongous_start_region()->bottom());
5     HeapWord* humongous_end =
6         hr->humongous_start_region()->bottom() + humongous->size();
7     MemRegion mr(hr->bottom(), MIN2(hr->top(), humongous_end));
8     scan_large_object(hr, humongous, mr);
9     if (hr->has_humongous_tail()) {
10        HeapWord* start = hr->old_objects_start();
11        scan_and_scrub_region(hr, start, pb);
12    }
13 }

```

Listing 4.8: Iterate over tail region to visit references

overlap, we search for the closest start address of any region that is less than or equal to the address to be checked, as shown in Listing 4.7. The check succeeds, if the address is contained inside this memory region.

4.3.3 Scan and Scrub of Tail

We need to introduce the tail region to the concurrent marking mechanism. Listing 4.8 shows the `scan_and_scrub_humongous_region` method, which scans humongous objects. In line 8, the humongous object inside the region is scanned. In case the humongous region has a tail, we scan and scrub over the tail in line 11 from the beginning of the tail region to the parsable bottom.

4.3.4 Eager Reclaim of Humongous Objects

Humongous objects are a special case. Because of their size, they have multiple consecutive regions, as explained in Section 2.3.1. G1 currently assumes that in the last region of the humongous object nothing follows. The current implementation puts a filler object at the end of the last region to fill up the area. But with the possibility of providing this tail for allocation, the humongous object is not alone anymore. In case a humongous object is reclaimed, the method `free_humongous_region` is called, shown in Listing 4.9. In line 6, the metadata of the humongous region is cleared. So we need to store a flag indicating whether the region had a tail with old objects, and the start of this tail, in lines 4 and 5. If the region has a tail, we cannot simply free it. Instead, we set the region to an old region in line 10 and add it to the set of old regions in line 13. To remain parsable, we fill the beginning of the region, where the end of the humongous object reached into, with a dummy object in line 15. Since this space is now unused, we add it to our hole structure in line 16. But during scan and scrub of concurrent marking, this hole is inserted again. To avoid this, we insert the hole also in our allocation tree for the SATB invariant fix in line 18. As a result, the scan skips this hole and does not insert it again. The skip condition can be seen in Listing 4.3 in line 4.

4.3.5 Fix Reference Scan of Humongous Region

The reference scanning of humongous objects for finding references pointing into other regions is shown in Listing 4.10. The method `do_oops_on_memregion_in_humongous` iterates inside the given memory region `mr` over all objects and returns the address where the iteration has ended. Since after a humongous object no other objects could previously live, there was no attempt to visit them. In case we have a humongous tail region, we first check if the memory region starts after the tail in line 6. In this case, we iterate over the old objects in line 7 and return in line 9 the end of the memory region, or the end of the last object scanned, depending on which address is larger. If the memory region does not start in the tail, but ends in the tail, we need to split the work. We first scan the start of the tail to the end of the memory region in line 11. After that, we fall through to the existing handling of the humongous object in line 15 to line 21 to complete the scan for the humongous part.

```
1 void G1CollectedHeap::free_humongous_region(G1HeapRegion* hr,  
2                                             G1FreeRegionList* free_list) {  
3     assert(hr->is_humongous(), "this is only for humongous regions");  
4     bool has_tail = hr->has_humongous_tail();  
5     HeapWord* old_objects_start = hr->old_objects_start();  
6     hr->clear_humongous();  
7     if (has_tail) {  
8         assert(old_objects_start != nullptr, "must be");  
9         size_t begin_size = pointer_delta(old_objects_start, hr->bottom());  
10        hr->set_old();  
11        {  
12            MutexLocker x(G1OldSets_lock, Mutex::_no_safepoint_check_flag);  
13            _old_set.add(hr);  
14        }  
15        hr->fill_with_dummy_object(hr->bottom(), begin_size);  
16        add_potential_old_hole(hr, hr->bottom(), begin_size);  
17        if (concurrent_mark()->cm_thread()->in_progress()) {  
18            concurrent_mark()->add_to_allocation_tree(  
19                MemRegion(hr->bottom(), begin_size));  
20        }  
21    } else { free_region(hr, free_list); }  
22 }
```

Listing 4.9: Handling of tail after freeing humongous region

```

1 HeapWord* G1HeapRegion::do_oops_on_memregion_in_humongous(
2                                     MemRegion mr,
3                                     Closure* cl) {
4     HeapWord* hum_tail_scan_end = mr.end();
5     if (has_humongous_tail()) {
6         if (mr.start() >= old_objects_start()) {
7             hum_tail_scan_end =
8             oops_on_memregion_iterate<Closure, in_gc_pause>(mr, cl);
9             return MAX2(mr.end(), hum_tail_scan_end);
10        } else if (mr.end() > old_objects_start()) {
11            hum_tail_scan_end =
12            oops_on_memregion_iterate<Closure, in_gc_pause>(
13                MemRegion(old_objects_start(), mr.end()), cl);
14        }
15    if (obj->is_objArray() || (sr->bottom() < mr.start())) {
16        obj->oop_iterate(cl, mr);
17        return hum_tail_scan_end;
18    } else {
19        size_t size = obj->oop_iterate_size(cl);
20        return MAX2(cast_from_oop<HeapWord*>(obj) + size, hum_tail_scan_end);
21    }

```

Listing 4.10: Iterate over tail region to visit references

4.3.6 Updating Root Regions

Before the changes, only one root range per region was possible. So the number of root regions was bounded by the number of regions, and a fixed-size array was used to store them. Since providing old holes can introduce new root regions, we may need to insert an arbitrary number of non-contiguous regions. So we replace the array with a dynamically growing data structure. Should the array run out of space, it dynamically increases its size.

Chapter 5

Evaluation

This chapter evaluates the performance impact of the implemented changes, investigating both their potential benefits and the overhead introduced by the new functionality. In particular, the evaluation examines their effects on application execution time, memory utilization, and garbage collection performance. We compare the modified JVM against an unmodified JDK 26 build to assess the tradeoffs associated with the additional functionality. This comparison allows us to quantify both its potential benefits, such as reduced garbage collection pause times, and its associated performance and overhead.

To ensure consistent and reproducible measurements, all benchmarks were executed automatically on a machine reserved for benchmarking, with no other user applications running concurrently. The experiments were conducted on the machine identified as *yuria*, equipped with an Intel® Xeon® E5-2620 v3 processor running at 2.40 GHz. The benchmark execution was integrated into the GitLab CI pipeline to automate the experiments and ensure that the same benchmark procedure was applied consistently.

5.1 Methodology

We first characterized the distribution of hole sizes to determine which hole sizes are relevant for bookkeeping. During benchmark execution, we logged every allocation request handled by the hole data structure. Each benchmark configuration was executed twice: once with the minimum heap size recommended for the benchmark and once with twice that heap size. The

resulting statistics provide insights into the frequency and distribution of hole sizes and inform the design and evaluation of the implemented data structures. Next, we tested the three implemented data structures. For every allocation request, we measured how long it took to fulfill it. We also logged whether the search was successful. We also measured insertion time for each data structure. For this, we ran each benchmark test four times: once for each data structure and once for the baseline, to measure the overhead of collecting the holes. Based on the results, we fix the data structure. We repeat all benchmarks under two settings: baseline and the *Survivor Allocation Holes* problem. We run each benchmark five times and average the logged pause times. For the benchmarks, we set `-XX:+DisableExplicitGC`. The option prevents explicit garbage collection requests, such as calls to `System.gc()` from triggering garbage collection. This setting reduces the influence of benchmark-induced explicit GC requests on the measured pause times and allows us to focus on garbage collection initiated by the JVM during normal application execution. However, this configuration does not represent workloads that intentionally trigger garbage collection through explicit requests. Since the evaluation focuses on the impact of the implemented changes on regular garbage collection behavior, we consider this limitation acceptable.

5.1.1 Benchmarks

We test the implementation against the DaCapo benchmarks [41] and the Renaissance benchmarks [42]. Table 5.1 and Table 5.2 show the benchmarks used, along with the recommended minimal heap sizes. Note that for some benchmarks (*) the minimal heap size is adjusted, since they struggled with the limited memory and run out of memory or triggered garbage collections over and over again. We increased the heap so the benchmark runs in under 60 seconds. Also, the DaCapo benchmarks `cassandra`, `h2o`, `tradebeans`, and `tradesoap` no longer run on JDK26 and therefore are excluded from the benchmarking.

5.2 Hole Size Analysis

In Figure 5.1, the distribution of holes inserted into the young generation data structure is visualized across all benchmarks. The upper plot shows the normalized frequency of each hole size as a histogram, while the lower plot shows the cumulative frequency of young generation

benchmark	min heap size	benchmark	min heap size
avroa	5m	kafka	250m*
batik	175m	luindex	29m
biojava	150m*	lusearch	30m*
eclipse	135m	pmd	191m
fop	20m*	spring	55m
graphchi	175m	sunflow	33m
h2	700m*	tomcat	21m
jme	1000m*	xalan	13m
python	50m*	zxing	183m

Table 5.1: DaCapo benchmark with minimal heap sizes

benchmark	min heap size	benchmark	min heap size
akka-uct	512m	movie-lens	48m
als	512m	naive-bayes	1536m
chi-square	512m	neo4j-analytics	1536m
db-shootout	96m	page-rank	768m
dec-tree	512m	par-mnemonics	48m
dotty	64m	philosophers	16m
finagle-chirper	1024m	reactors	512m
finagle-http	96m	rx-scrabble	48m
fj-kmeans	128m	scala-doku	48m
future-genetic	16m	scala-kmeans	32m
gauss-mix	512m	scala-stm-bench7	512m
log-regression	512m	scrabble	128m
mnemonics	48m		

Table 5.2: Renaissance benchmark with minimal heap sizes

holes, normalized to 1. A vertical line indicates the region size of 1MB. The minimum hole size supported by each data structure is also indicated. The list structure has a minimal hole size of 32 bytes, while the tree structure need at least 64 bytes.

For both data structures, almost no holes need to be rejected, as only a small number are smaller than the respective minimum hole size. Note that the horizontal axis is logarithmic to represent the wide range of hole sizes better. Although it appears that most holes have a size of approximately 512KB, the logarithmic scaling means that the interval between 512KB and 1024KB represents half of the possible hole sizes between 0 and the region size.

Figure 5.2 visualizes the distribution of old generation hole sizes in the same manner. In

contrast to the young generation holes, almost all old generation holes fall within the range from 16 bytes to 2KB. The heap fragmentation appears to consist mainly of small gaps, likely because the benchmark predominantly allocates small objects. Both data structure must reject about 50% of insertion requests because of their minimum supported hole size.

The distribution of holes associated with humongous objects is shown in Figure 5.3. Most holes are at least 512KB bytes in size, indicating that approximately half of a humongous region remains unused. This observation supports the claim that the unused space at the end of humongous regions can constitute a significant source of memory waste.

The hole sizes are additionally presented as boxplots in Figure 5.4. The medians of the young generation and humongous holes are both in a higher order of magnitude than those of the old generation holes. Young generation holes exhibit a wider distribution, with a minimum hole size of 8 bytes, whereas humongous holes show comparatively low variance. In contrast, old generation holes are relatively small, as shown in the corresponding histogram. We omit outliers from the boxplot for clarity.

The amount of wasted and unused PLAB space for each generation was also recorded, with the results shown in Table 5.3. Here, “waste” refers to the amount of memory remaining in a PLAB when the PLAB is closed or retired. The respective PLAB can no longer use this memory. “Unused” refers to the amount of memory that remains available in currently active PLABs. For both generations, the average amount of wasted memory across all benchmarks accounts for less than one percent of the allocated memory.

	Wasted	Unused	Total
Young PLABs	0.36%	7.48%	7.83%
Old PLABs	0.28%	7.80%	8.08%

Table 5.3: PLAB statistics of wasted and unused memory during benchmarking

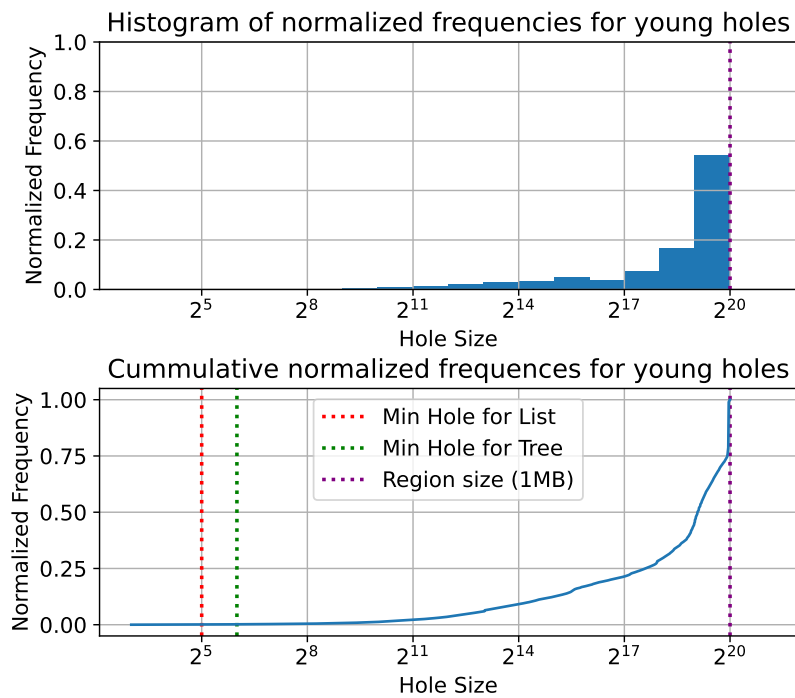


Figure 5.1: Distribution of young hole sizes collected during benchmarking

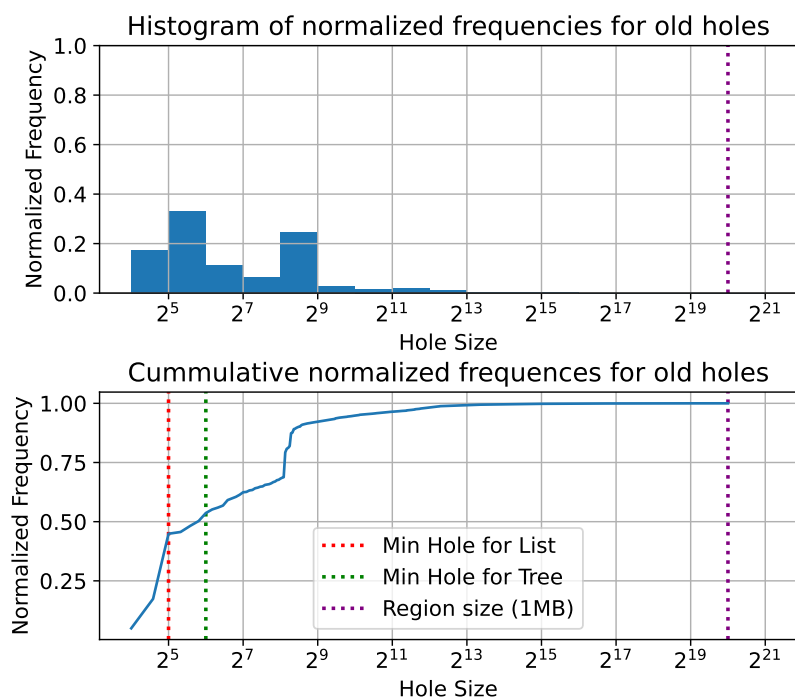


Figure 5.2: Distribution of old hole sizes collected during benchmarking

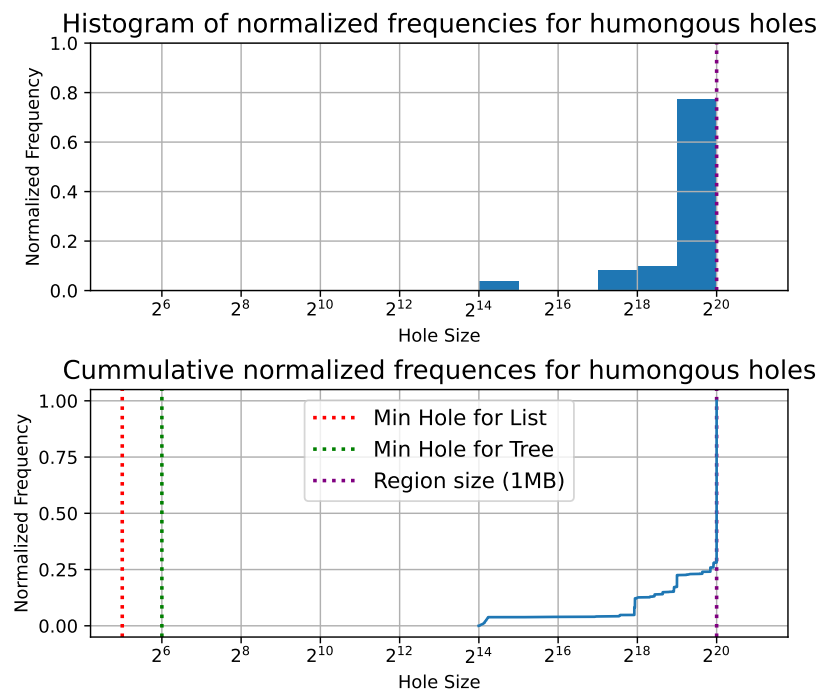


Figure 5.3: Distribution of humongous hole sizes collected during benchmarking

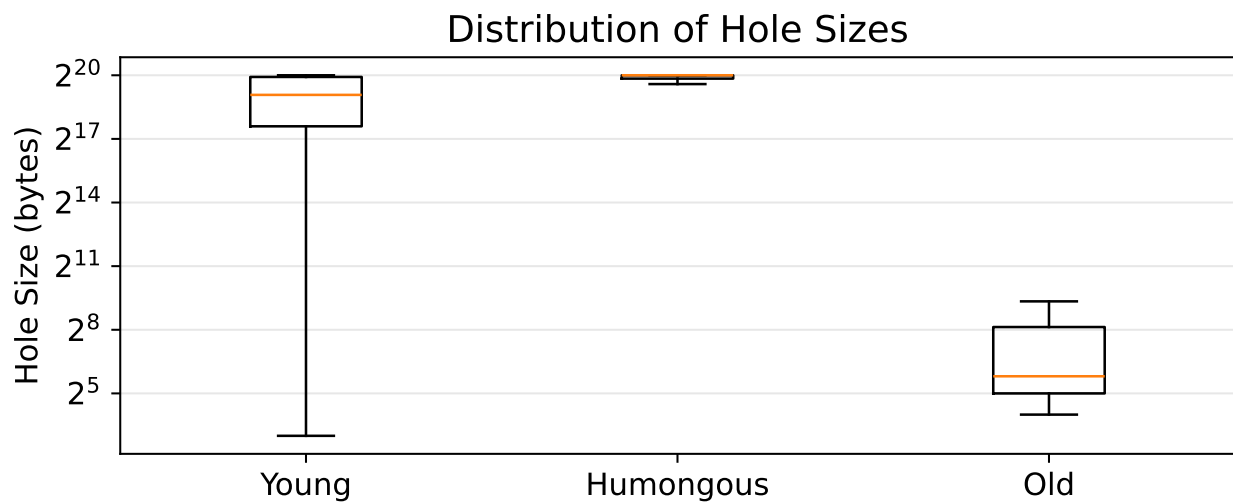


Figure 5.4: Distribution of hole sizes

5.3 Performance Evaluation

5.3.1 Comparison of the Data Structures

During a complete benchmark run, we measure all insertion and lookup operations for the three proposed data structures. We sum the insertion and lookup times separately for each setup and compute the corresponding averages. We therefore do not distinguish between individual benchmarks, treating all runs as one large combined run in order to simplify the measurement of the operations. The results are presented in Table 5.4.

For insertion operations, the list data structure performs best, with an average insertion time of 0.95ns. The tree-based implementations require slightly more than 1ns on average. However, for finding a hole, the list requires approximately 600ns, while the binary search tree is approximately 43% faster. The red-black tree achieves an even greater improvement of about 49% over the list. Based on these measurements, we select the red-black tree as the data structure for storing holes. We also visualize the individual timings as boxplots in Figures 5.5 and 5.6.

We also record the success rates of both hole insertion and hole lookup in the young generation. As expected, almost all young holes are inserted into the young generation data structure. When requested by the allocator, the data structure provides a hole from the young generation about 57% of the time. This is likely because holes are requested shortly after a young garbage collection, when the data structure has not yet had time to accumulate holes and is therefore empty.

		List		BST		RBT	
		Young	Old	Young	Old	Young	Old
Inserting	Hitrate	99.63%	-	99.56%	-	99.56%	-
	Avg. Time [ns]	0.95		1.19		1.16	
	Total Time [s]	0.06		0.07		0.07	
Finding	Hitrate	59.51%	-	58.24%	-	56.63%	-
	Avg. Time [ns]	593.71		334.93		304.05	
	Total Time [s]	35.51		20.03		18.18	

Table 5.4: Performance comparison of insert and find operations over the different data structures: list, binary tree, and red-black tree

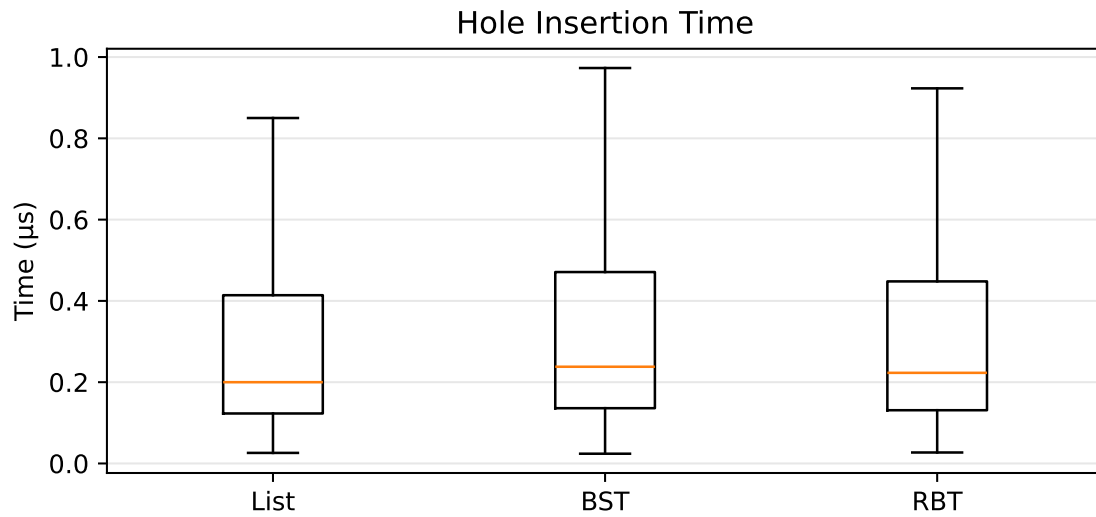


Figure 5.5: Boxplot of timing the insertion of holes in the three data structures

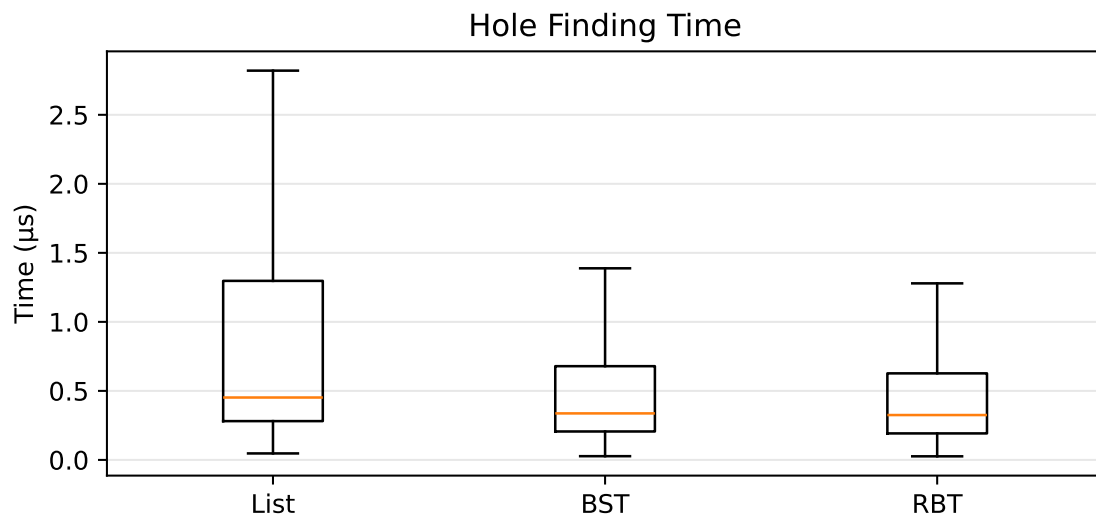


Figure 5.6: Boxplot of timing the finding of holes in the three data structures

5.3.2 Final Benchmark Run

In Figure 5.7, the young garbage collection pause times for the survivor-hole problem with the minimal heap size are presented. The pause times are normalized to the baseline configuration, so the baseline has a normalized pause time of 1, and the survivor setting pause times are given relative to the baseline. For young garbage collections, the best-performing benchmark is **zxing**, which reduces pause time by approximately 79% compared to the baseline. In contrast, **luindex** performs worst, with a pause time approximately twice as long as the baseline. Figure 5.8 shows the corresponding total garbage collection pause times, combining both young and full garbage collection pauses. Again, **zxing** achieves the best performance and **luindex** performs worst. For the doubled heap size, the results for each benchmark are presented in Figures 5.9 and 5.10. The benchmark **avrrora** performs the best in terms of young garbage collection pause time, while **batik** has the lowest total garbage collection pause time. For both pause time metrics, **neo4j-analytics** achieves the highest garbage collection pause time. Across all benchmarks and both heap sizes, the survivor setting achieves an average reduction in garbage collection pause time of approximately 19% compared to the baseline.

Additionally, we have already partially evaluated the performance of humongous tail allocation on the subset of benchmarks for which the implementation was able to run. The results for total garbage collection pause time are shown in Figures 5.11 and 5.12. For the minimal heap configuration, **fj-kmeans** performs best with a reduction of around 44%, whereas **chi-square** requires approximately four times as long. For the doubled heap size, **avrrora** performs best, while **jme** performs worst. Across all benchmarks, we were able to reduce the total pause time by 3%. Particularly interesting is that the allocation achieves a 100% hit rate, indicating that the data structure is always able to provide memory on request.

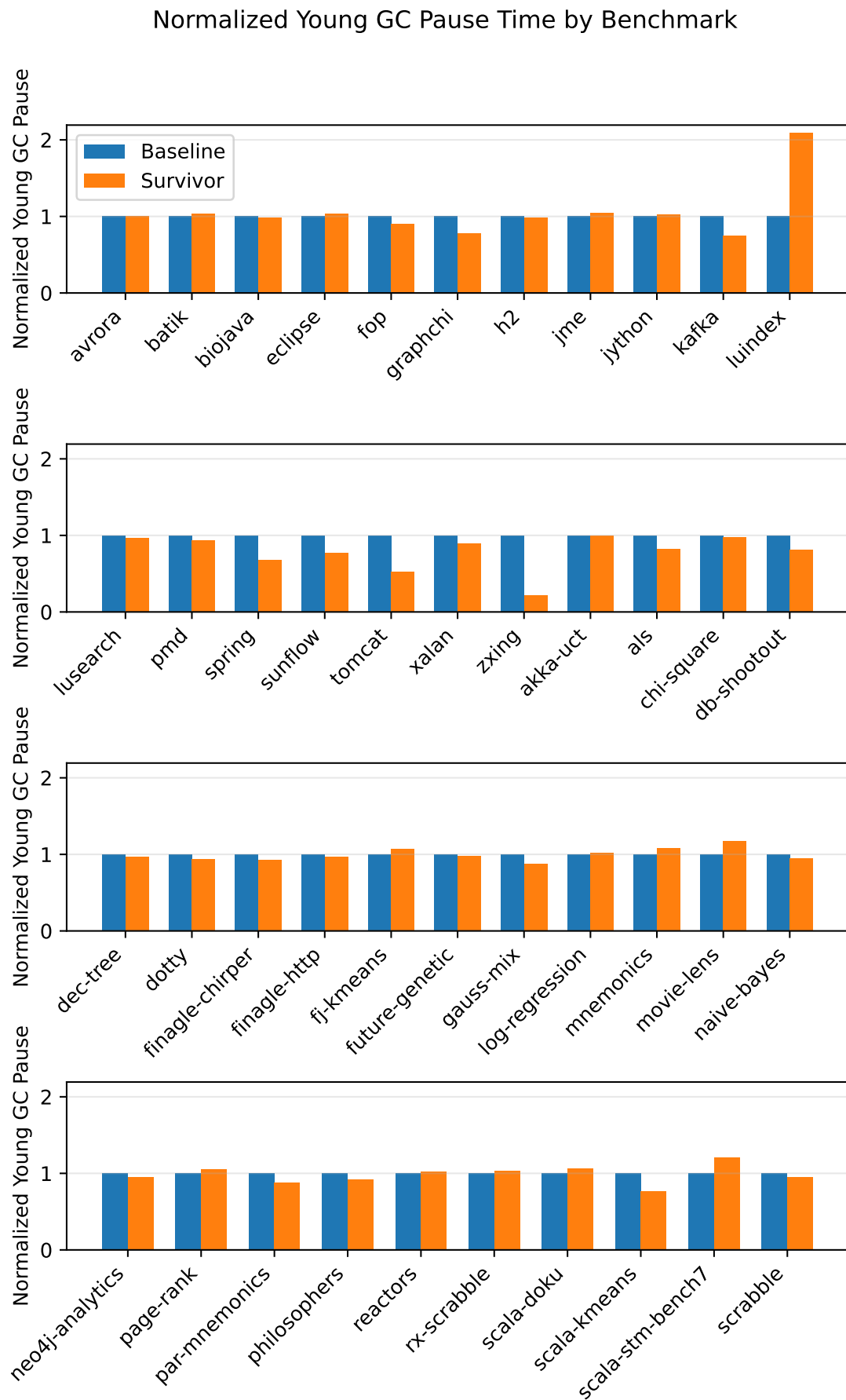


Figure 5.7: Young GC pause time normalized to baseline with minimum heap sizes for survivor holes

Normalized Total GC Pause Time by Benchmark

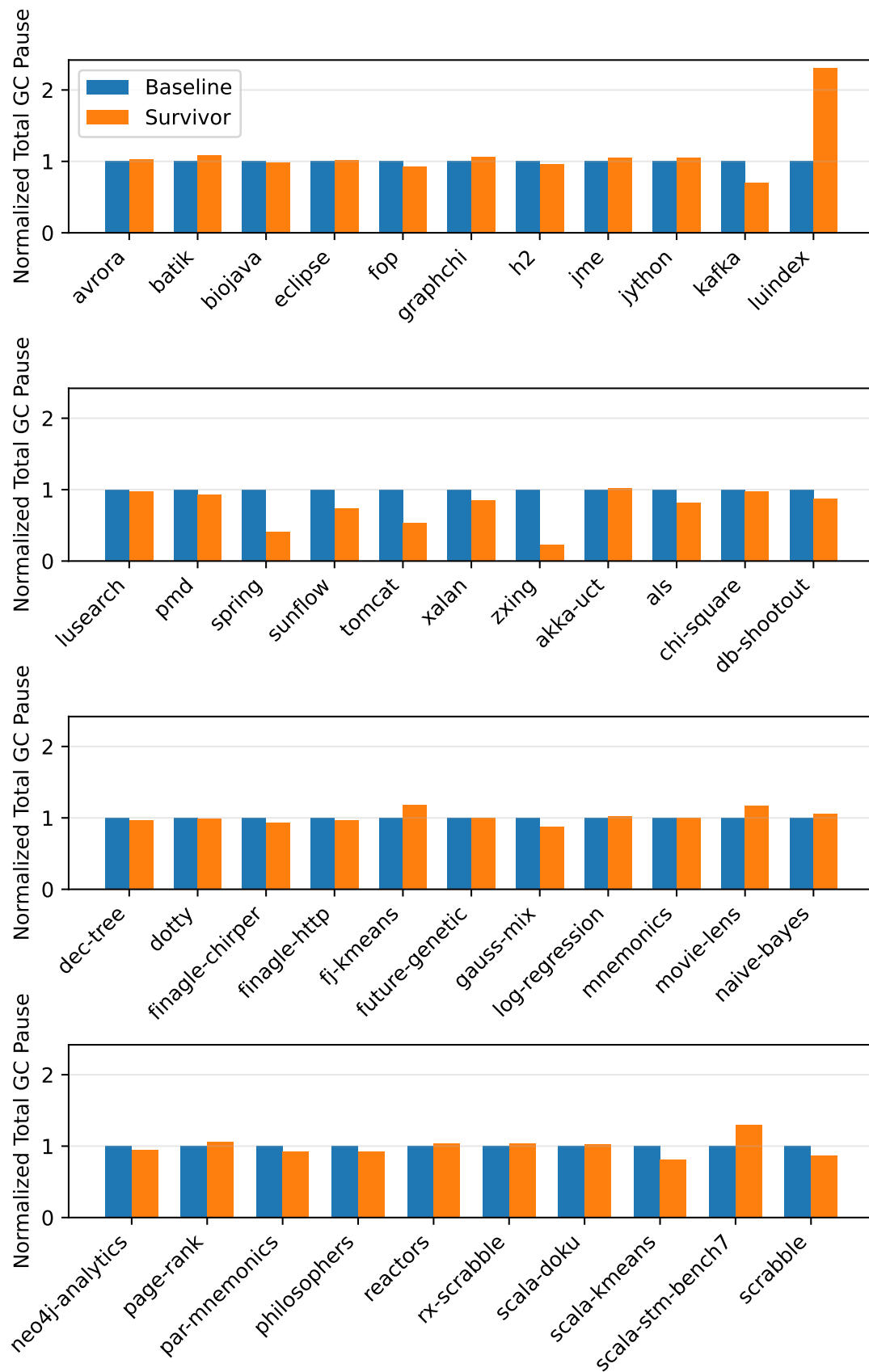


Figure 5.8: Total GC pause time normalized to baseline with minimum heap sizes for survivor holes

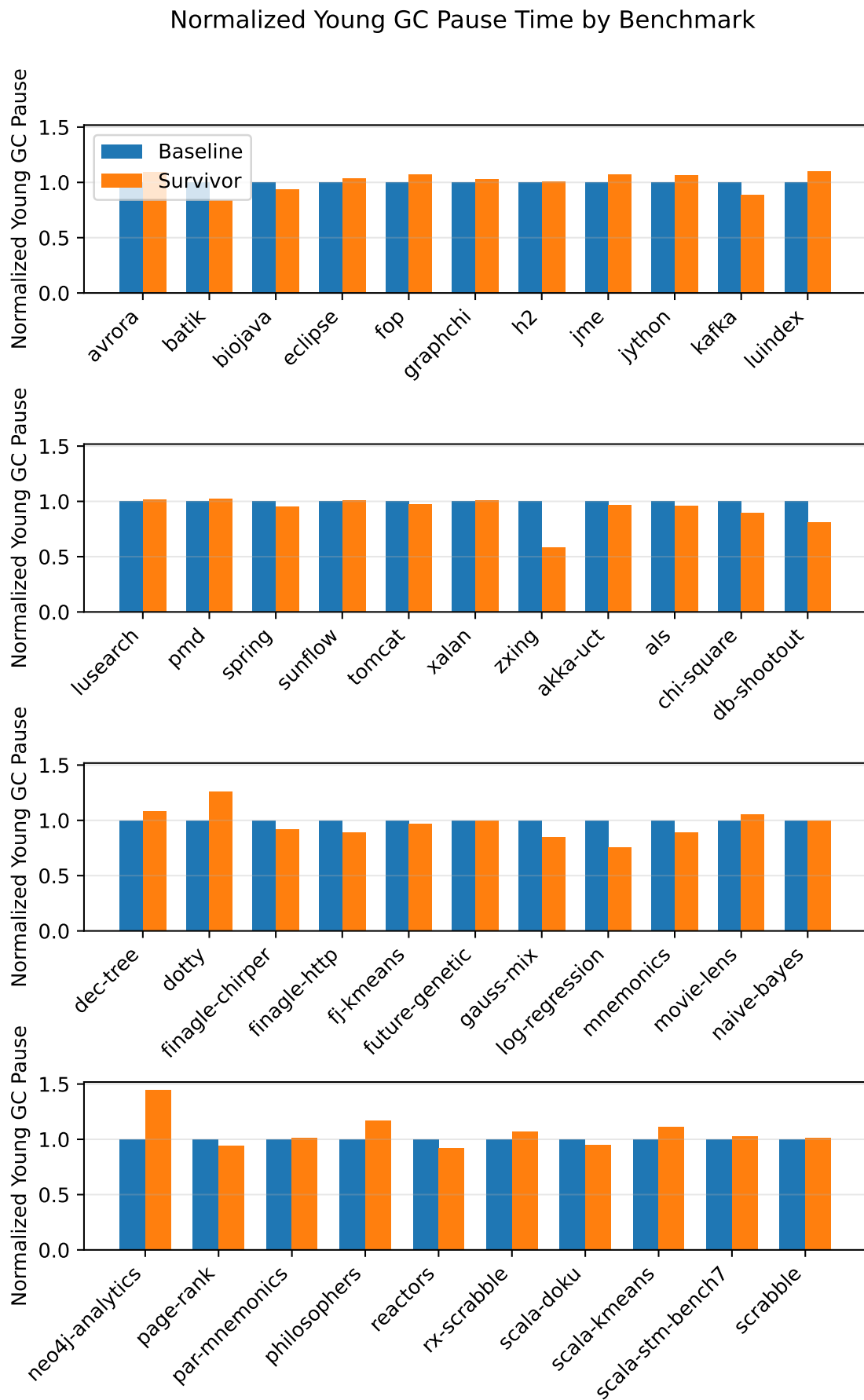


Figure 5.9: Young GC pause time normalized to baseline with doubled heap sizes for survivor holes

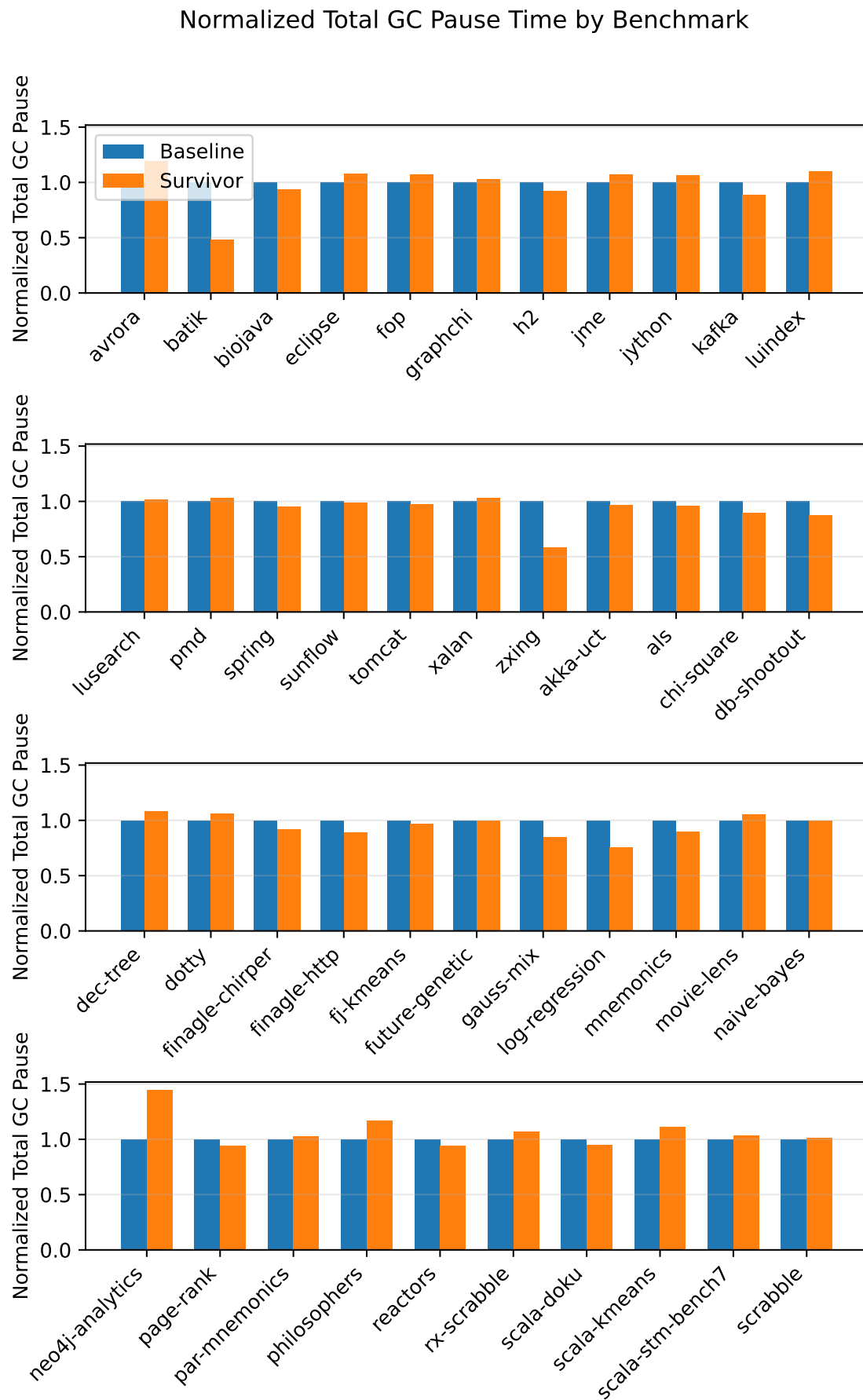


Figure 5.10: Total GC pause time normalized to baseline with doubled heap sizes for survivor holes

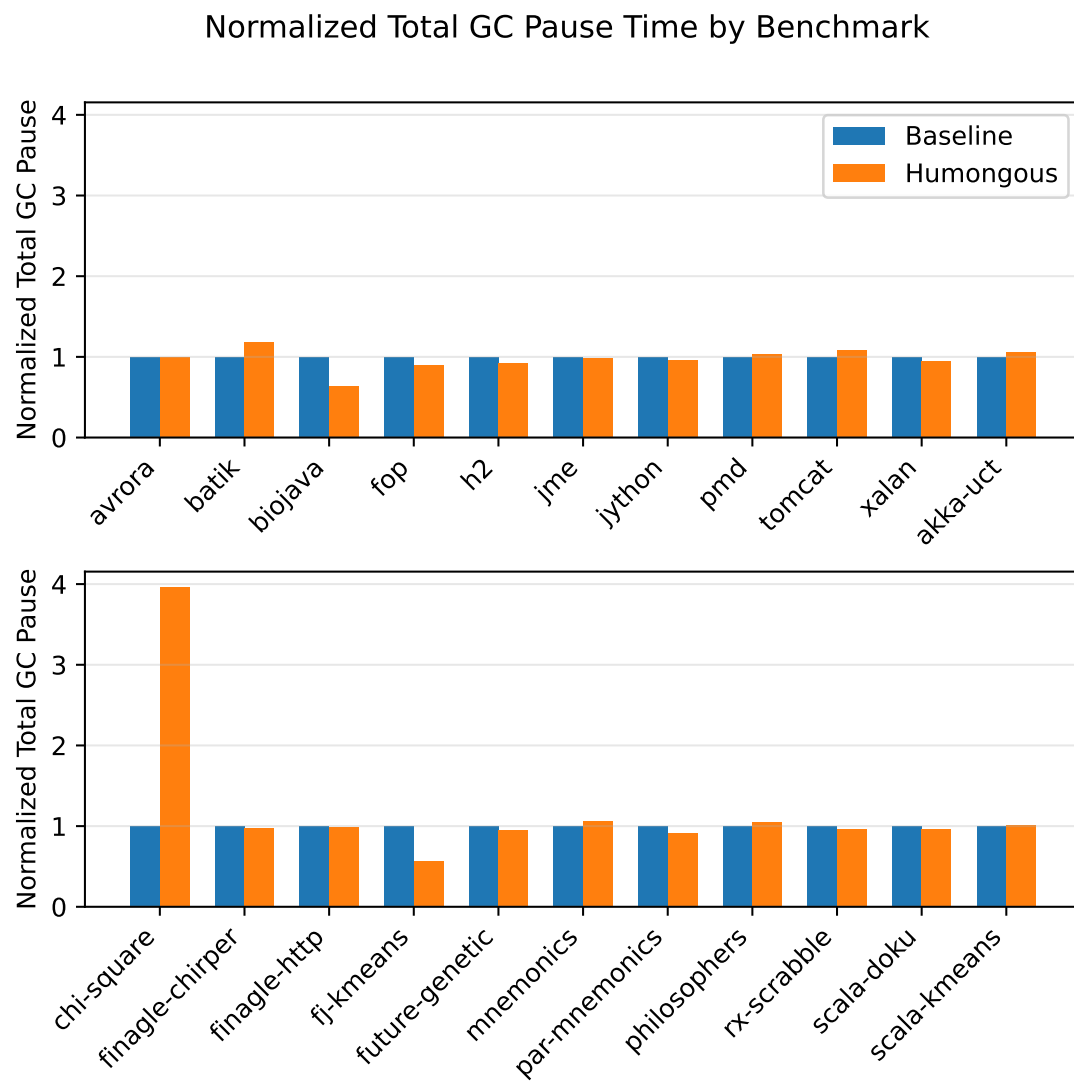


Figure 5.11: Total GC pause time normalized to baseline with minimum heap sizes for humongous holes

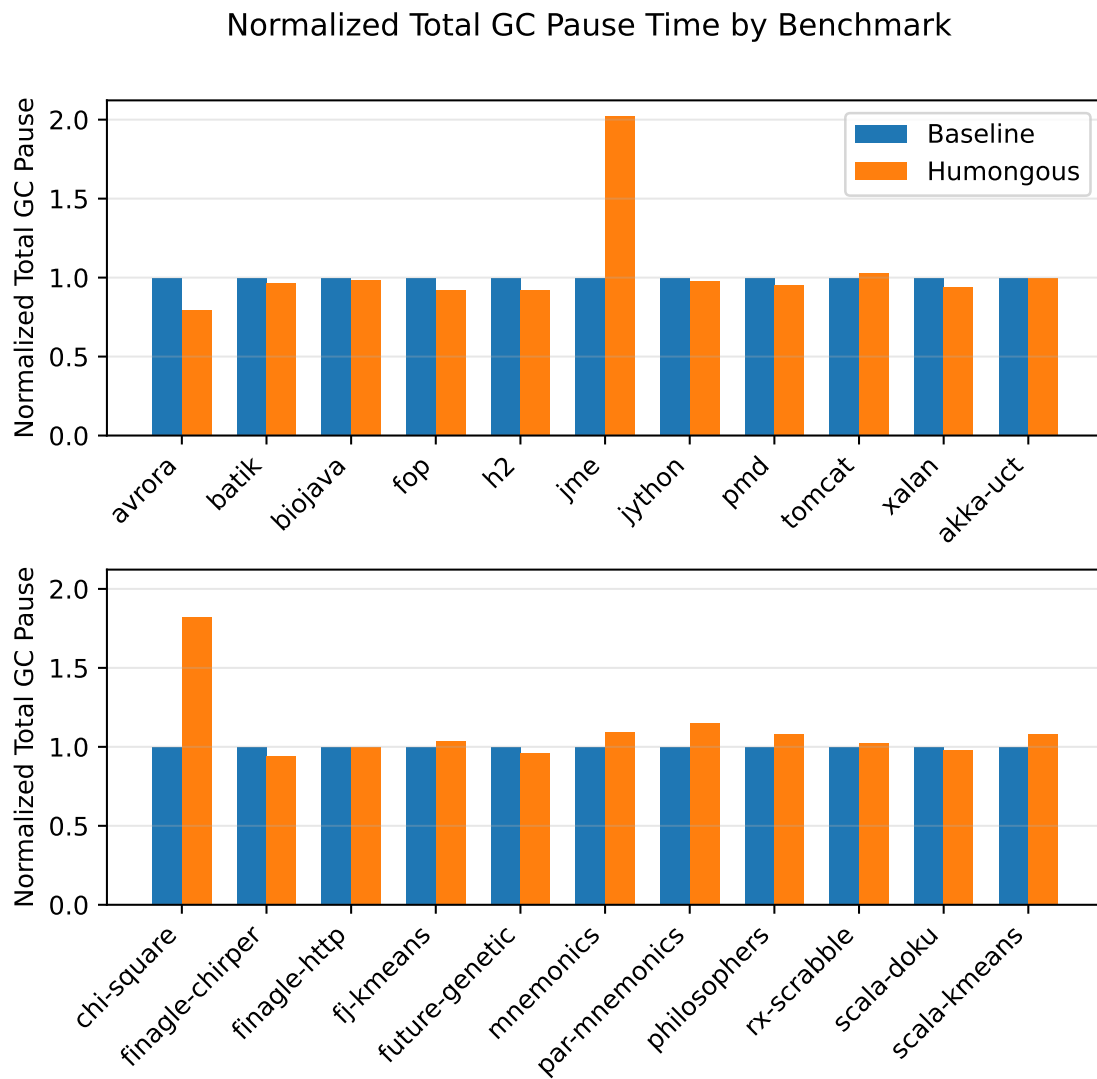


Figure 5.12: Total GC pause time normalized to baseline with doubled heap sizes for humongous holes

Chapter 6

Conclusion and Future Work

This thesis laid the groundwork for making memory that G1 currently discards as waste available to the allocator. We identified three sources of underutilized memory in G1 regions: survivor allocation holes, the unused tail of humongous regions, and fragmentation inside old regions. We implemented mechanisms to remember, locate, and reuse them without allocating additional heap memory by embedding the necessary bookkeeping directly into the holes themselves. We evaluated three candidate data structures and found that a red-black tree offers the best balance of insertion and lookup performance, making it the structure of choice for the final implementation. Enabling hole reuse required rethinking several core assumptions of G1, including the SATB marking invariant, remembered set construction, and the region layout around humongous objects. Our evaluation on the DaCapo and Renaissance benchmark suites shows that reusing holes in the survivor region reduces total garbage collection pause time by approximately 19% on average, with individual benchmarks improving by as much as 79%. A preliminary evaluation of the humongous-region tail shows a smaller but still positive average reduction of approximately 3%. The observed 100% hit rate whenever the allocator requested memory from this space indicates that, when available, it provides reliable benefits. However, it was not evaluated as extensively as the survivor case.

Working with the G1 codebase remains a major challenge. Beyond acquiring theoretical knowledge of the collector, one also needs to develop specific domain knowledge in order to identify the relevant parts of the codebase. Knowing which assumptions are violated by a given change is crucial. Working with the tools required to debug the code is a further challenge in itself.

Addressing the `End of Humongous Region Holes` problem and the `Fragmentation of Old`

Region problem appears promising. Early tests indicate that almost all holes could be used for allocation. While this thesis presents the fundamental groundwork, further adjustments to the G1 mechanism are still required. Additionally, it may be worth considering remembering holes left over at the end of a PLAB to make them available for allocation.

List of Figures

2.1	Example Heap Layout in G1	14
2.2	Example of SATB on a region. Object A and B are snapshotted, and TAMS was set to top . After SATB, object C was allocated, and top has moved	16
2.3	Flow chart of concurrent marking phases	18
3.1	Allocation into active survivor region during garbage collection	26
3.2	Unused memory at the end of the last region of a humongous object	27
3.3	Fragmentation of old region	27
4.1	Example of searching for the number 5 in a binary tree	33
4.2	Linking holes together in a tree structure	36
4.3	Linking holes together with a linked list	37
4.4	Aligning the old region after a humongous object with the card table	38
5.1	Distribution of young hole sizes collected during benchmarking	47
5.2	Distribution of old hole sizes collected during benchmarking	47
5.3	Distribution of humongous hole sizes collected during benchmarking	48
5.4	Distribution of hole sizes	48
5.5	Boxplot of timing the insertion of holes in the three data structures	50
5.6	Boxplot of timing the finding of holes in the three data structures	50
5.7	Young GC pause time normalized to baseline with minimum heap sizes for survivor holes	52
5.8	Total GC pause time normalized to baseline with minimum heap sizes for survivor holes	53
5.9	Young GC pause time normalized to baseline with doubled heap sizes for survivor holes	54
5.10	Total GC pause time normalized to baseline with doubled heap sizes for survivor holes	55

5.11	Total GC pause time normalized to baseline with minimum heap sizes for humongous holes	56
5.12	Total GC pause time normalized to baseline with doubled heap sizes for humongous holes	57

List of Tables

5.1	DaCapo benchmark with minimal heap sizes	45
5.2	Renaissance benchmark with minimal heap sizes	45
5.3	PLAB statistics of wasted and unused memory during benchmarking	46
5.4	Performance comparison of insert and find operations over the different data structures: list, binary tree, and red-black tree	49

Listings

4.1	Implemented method for inserting in holes in data structure	29
4.2	Collecting young holes during retirement	30
4.3	Collecting old holes during scan or scrub	31
4.4	Slow allocation attempt for young objects	32
4.5	Allocation attempt of old object during garbage collection	32
4.6	Algorithm for finding best fitting hole in BST	35
4.7	Checking if an address is inside a requested hole used for allocation	39
4.8	Iterate over tail region to visit references	39
4.9	Handling of tail after freeing humongous region	41
4.10	Iterate over tail region to visit references	42

Bibliography

- [1] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.
- [2] Jonathan C Erb. A decade of programming languages: Trends in popularity and influence. 2026.
- [3] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-wesley, 2013.
- [4] Oracle. Java garbage collection basics. <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>. Accessed: 2026-08-30.
- [5] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *Ieee micro*, 17(3):36–43, 1997.
- [6] Andreas Krall. Efficient javavm just-in-time compilation. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pages 205–212. IEEE, 1998.
- [7] Oracle. The java hotspot performance engine architecture. <https://www.oracle.com/java/technologies/whitepaper.html>. Accessed: 2026-08-02.
- [8] OpenJDK. Openjdk. <https://openjdk.org>. Accessed: 2026-08-11.
- [9] James Clause and Alessandro Orso. Leakpoint: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 515–524, 2010.

-
- [10] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.
 - [11] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. CRC press, 2023.
 - [12] Oracle. Available collectors. <https://docs.oracle.com/en/java/javase/26/gctuning/available-collectors.html>. Accessed: 2026-08-02.
 - [13] SUN Microsystems. Memory management in the java hotspot™ virtual machine, 2006. Accessed: 2026-07-24.
 - [14] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Hoboken, NJ, 10 edition, 2018.
 - [15] Bruce K Haddon and William M Waite. A compaction procedure for variable-length storage elements. *The Computer Journal*, 10(2):162–165, 1967.
 - [16] Oracle. Garbage-first (g1) garbage collector. <https://docs.oracle.com/en/java/javase/26/gctuning/garbage-first-g1-garbage-collector1.html>. Accessed: 2026-08-05.
 - [17] Barry Hayes. Using key object opportunism to collect old objects. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 33–46, 1991.
 - [18] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.
 - [19] OpenJDK Wiki. Shenandoah gc. <https://wiki.openjdk.org/spaces/shenandoah/pages/25002018/Main>. Accessed: 2026-08-11.

-
- [20] OpenJDK. Jep 318: Epsilon: A no-op garbage collector (experimental). <https://openjdk.org/jeps/318>. Accessed: 2026-08-11.
 - [21] OpenJDK. Make g1 the default garbage collector in all environments. <https://openjdk.org/jeps/523>. Accessed: 2026-08-11.
 - [22] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, 2004.
 - [23] Thomas Schatzl. Concurrent marking in g1. <https://tschatzl.github.io/2022/08/04/concurrent-marking.html>. Accessed: 2026-07-23.
 - [24] Edsger W Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth FM Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.
 - [25] Thomas Schatzl. Welcome 20collection set candidates early. <https://tschatzl.github.io/2021/02/26/early-prune.html>. Accessed: 2026-07-23.
 - [26] Antony L Hosking and Richard L Hudson. Remembered sets can also play cards. *OOPSLAGC OOP93*. Available for anonymous FTP from *cs. utexas. edu in/pub/-garbage/GC93*, 1993.
 - [27] Thomas Schatzl. Card table card size shenanigans. <https://tschatzl.github.io/2022/02/15/card-table-card-size.html>. Accessed: 2026-07-23.
 - [28] OpenJDK. Early reclamation of large objects in g1. <https://bugs.openjdk.org/browse/JDK-8027959>. Accessed: 2026-08-11.
 - [29] OpenJDK. Rebuild remembered sets during the concurrent cycle. <https://bugs.openjdk.org/browse/JDK-8180415>. Accessed: 2026-08-30.
 - [30] OpenJDK. Use single mark bitmap in g1. <https://bugs.openjdk.org/browse/JDK-8210708>. Accessed: 2026-09-01.

-
- [31] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.
 - [32] Thomas Schatzl. New write barriers for g1. <https://tschatzl.github.io/2025/02/21/new-write-barriers.html>. Accessed: 2026-07-23.
 - [33] OpenJDK. Jep 522: G1 gc: Improve throughput by reducing synchronization. <https://openjdk.org/jeps/522>. Accessed: 2026-09-01.
 - [34] Wenyu Zhao and Stephen M Blackburn. Deconstructing the garbage-first collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 15–29, 2020.
 - [35] OpenJDK. G1 gc: Reduce need for full gcs. <https://openjdk.org/jeps/156>. Accessed: 2026-08-30.
 - [36] OpenJDK. Jep 423: Region pinning for g1. <https://bugs.openjdk.org/browse/JDK-8276094>. Accessed: 2026-08-30.
 - [37] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
 - [38] Lynda Bounif and Djamel Eddine Zegour. Avl and red-black tree as a single balanced tree. *International Journal of Applied Computer Science & Mathematics*, 10(2):72–77, 2016.
 - [39] Thomas Schatzl. Heap regions x-large. <https://tschatzl.github.io/2021/11/15/heap-regions-x-large.html>. Accessed: 2026-07-07.
 - [40] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson, 4th edition, 2013.
 - [41] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. Rethinking Java performance analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*,

Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025. ACM, 2025.

- [42] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.