Bachelor's Thesis

Implementation of a Non-Moving, Incremental, and Generational Garbage Collector

Nicolas Trüssel

Remigius Meier
Responsible assistant

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

October 2016

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Abstract

State-of-the-art virtual machines (VM), like HotSpot or the Common Language Runtime, use moving garbage collectors and therefore seem to agree that moving garbage collection (GC) is superior to non-moving GC in terms of performance. However, the moving property causes problems for dynamic language VMs, as these VMs often provide functionality through native libraries. Since most native libraries assume memory to be non-moving, passing data between them and the VM usually relies on object pinning, temporary copies, or similar techniques to prevent dangling pointers.

This bachelor's thesis implements the proposal of a novel non-moving, incremental, and generational garbage collector and evaluates its behavior under workloads that are typical for dynamic language VMs. It investigates how fragmentation behaves, how the generational mode influences overall performance, and how its performance compares to PyPy's incminimark, a moving garbage collector with otherwise similar properties. Special attention is paid to whether using an existing GC algorithm with the implementation of the proposal has any drawbacks over using the proposed new algorithm.

The results show that the implemented garbage collector still suffers from high fragmentation, one of the main problems of non-moving GC. While the generational mode significantly improves overall performance, PyPy's incminimark is still a lot faster for the majority of the examined workloads, as the proposal omits important optimizations. Using the new algorithm provides a slight benefit over existing algorithms. The thesis concludes that the algorithm fails to achieve some of its design goals and recommends further modifications.

Zusammenfassung

Moderne Laufzeitumgebungen, wie zum Beispiel HotSpot oder die Common Language Runtime, nutzen verschiebende Garbage Collectors, da diese für die meisten Anwendungszwecke bessere Performance bieten. Laufzeitumgebungen für dynamische Sprachen stellt verschiebende Garabge Collection (GC) jedoch vor Probleme, da diese Laufzeitumgebungen häufig Funktionalitäten über native Bibliotheken bereitstellen. Da beinahe alle nativen Bibliotheken davon ausgehen, dass Objekte nicht verschoben werden, muss Speicher der mit nativen Bibliotheken geteilt wird mittels temporärer Kopien, Object Pinning oder anderen Techniken am Verschieben gehindert werden um dangling Pointers zu verhindern.

Diese Bachelorarbeit implementiert den Vorschlag eines neuen, nicht verschiebenden, inkrementellen und generationellen Garbage Collectors und evaluiert dessen Verhalten unter Verhältnissen wie sie in Laufzeitumgebungen für dynamische Sprachen vorkommen. Es werden Fragmentierung und Einfluss des generationellen Modus untersucht, sowie die Leistung mit PyPys Incminimark verglichen. Incminimark unterscheidet sich zum untersuchten Garbage Collector darin, dass er Objekte verschiebt, hat sonst aber ähnliche Eigenschaften. Ausserdem wird untersucht, ob die Implementierung des Vorschlags auch einen existierenden GC Algorithmus benutzen kann, ohne dass Nachteile entstehen.

Die Resultate zeigen, dass der implementierte Garbage Collector weiterhin unter hoher Speicherfragmentierung leidet, eines der Hauptprobleme nicht verschiebender GC. Der generationelle Modus bringt eine signifikante Verbesserung der Leistung, welche für die Mehrheit der untersuchten Anwendungsszenarien jedoch immer noch deutlich schlechter als diejenige von Incminimark ist. Dies liegt zu einem grossen Teil daran, dass die Beschreibung des neuen Garbage Collectors wichtige Optimierungen auslässt. Der Einsatz des neuen GC Algorithmus bietet leichte Vorteile gegenüber existierenden Algorithmen. Zum Abschluss zeigt die Arbeit, dass der Algorithmus einige seiner Designziele nicht erreicht und schlägt Verbesserungen vor.

Contents

1 Introduction

State-of-the art virtual machines (VM), like HotSpot for Java or the Common Language Runtime for .NET, use moving garbage collectors and therefore seem to agree that moving garbage collection (GC) offers better performance than non-moving GC for most use cases. However, for dynamic language VMs the moving property causes many issues, as these VMs often provide functionality through native libraries for performance reasons. Since most native libraries assume memory to be non-moving, passing data between them and the VM usually relies on object pinning, temporary copies, or similar techniques to prevent dangling pointers.

Recently, Mike Pall, the developer of LuaJIT [4], proposed a new design of a non-moving, incremental, and generational garbage collector in [3]. As such, it is tailored for the use in dynamic language VMs. However, it was never implemented in working code. This project's aim was to create a prototype of the proposal and to evaluate its behavior under workloads that are typical for dynamic language VMs. Therefore the prototype was integrated into PyPy [5], a VM for Python [6].

More precisely, the discussed aspects include fragmentation behavior, the influence of the generational mode on performance, and whether using an existing GC algorithm with the implementation of the proposed design has drawbacks over using the proposed new GC algorithm. Finally the GC is compared to PyPy's incminimark. However, the evaluation of interaction between native libraries and the new garbage collector lies outside the scope of this thesis.

1.1 Organization

A description of the high level design of Mike Pall's proposal and what its motivations are can be found in Chapter 2. Chapter 3 covers the implementation of the prototype, providing details where they are essential for understanding the

evaluation. Chapter 4 is dedicated to the evaluation of the prototype, results, and the discussion thereof. The thesis is concluded in Chapter 5 by summarizing the findings, recommending modifications of the design, and talking about related work.

2 Background

This Chapter provides a high level description of the implemented GC algorithm, as presented by Mike Pall's proposal.

2.1 Mark-and-Sweep Garbage Collection

Mike Pall's proposal is a refinement of the tri-color algorithm described in [2], and thus is a mark-and-sweep garbage collector. Mark-and-sweep garbage collectors work in two steps: In a first step, the mark phase, all reachable (live) objects are marked. An object is considered to be live when it can be reached through references from other live objects or when it belongs to a given set of root objects. The process of marking all objects that are live is also called tracing. In a second step, the sweep phase, the heap memory is traversed, and all objects that are not marked are freed. Marked objects become unmarked, preparing for the next mark phase.

Most GC algorithms use colors to indicate different marking states of objects. In a two-color algorithm, unmarked objects are colored white and marked objects are colored black. For a collection step, a two-color algorithm just atomically executes a mark and a sweep phase. The tri-color algorithm adds gray colored objects and slightly changes the meaning of the colors: White colored objects are unmarked objects, black objects are marked objects with no references to any white object, and gray objects are marked objects that may have references to white objects. In the tri-color algorithm a black object may never contain a reference to a white object.

The additional color allows to split the collection step into iterative steps (incremental collection) as opposed to halting the main program (mutator) to execute a full collection like the two-color algorithm does. To maintain the invariant that no black object points to a white one, the mutator has to call a write barrier on all objects whose references it modifies. The collector then decides whether it

has to change the color of the object in question (in case a reference to a white object is stored in a black object).

2.2 Quad-Color Algorithm

(Summary of [3])

The proposed design tries to achieve the following design goals:

- Collection steps should be incremental.
- Latency needs to be very low.
- To ease the interaction with native libraries objects must be non-moving.
- Fragmentation has to be tightly controlled.
- Collections should be able to use a generational mode.

Generational GC will be explained later in this chapter.

Mike Pall introduces a new GC algorithm that uses four colors, making it a quad-color algorithm. White objects are unmarked and clean. Clean means the object contains no references that were updated since it was last examined by the collector. Similarly, black objects are marked and clean. Light gray objects are unmarked objects that contain updated references, and dark gray objects are marked objects that contain updated references. Additionally the proposal demands that color information is stored as a mark bit and a gray bit. Table 2.1 shows how the state of the mentioned bits is related to the actual color of the object.

		Gray Bit	
		0	1
Mark Bit	0	white	light gray
	1	black	dark gray

Table 2.1: Color information encoding

The distinction between light gray and dark gray makes the properties *reachable* and *clean* orthogonal. Reachable objects are marked (dark gray or black), unreachable objects are unmarked (white or light gray). Objects that contain updated references since they were last examined are dirty (light gray or dark gray), while objects that were not modified are clean (white or black).

Whenever an object gets colored dark gray it is pushed to a gray stack, which stores all objects that have to be traced before a sweep phase may start. Dark gray objects have to be traced again because they are both live and contain references that were not yet examined, hence might point to live objects that are not marked yet. The quad-color algorithm maintains two invariants:

1. An object is on the gray stack if and only if it is colored dark gray
2. No black object points to a white or light gray object.

Like the tri-color algorithm, the quad-color algorithm supports incremental collections, as this is a design goal. Thus it also requires a write barrier, which the mutator has to call on all objects whose references it modifies. The write barrier first checks whether the object it is called on is colored light or dark gray. If that is the case it exits right away, as these objects are already considered dirty. This is called the fast path of the write barrier. If the check fails (the object is colored white or black), the write barrier marks the object as dirty by turning white objects light gray and black objects dark gray. In case the object is now colored dark gray, it is pushed to the gray stack. This is called the slow path of the write barrier. Figure 2.1 illustrates, how the object color is modified by all the actions that happen during a garbage collection cycle.

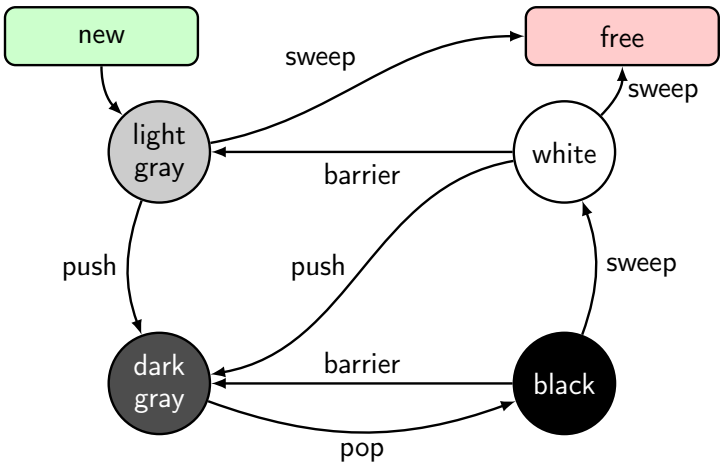

Figure 2.1: State diagram of the quad color algorithm. Adapted from [3].

Newly allocated objects are colored light gray, as they are considered to be unreachable after creation. During the mark phase, traced objects are colored dark gray and pushed to the gray stack, unless they are already marked (dark gray or black). This makes the garbage collector consider these objects reachable. Objects then are popped from the gray stack, colored black, and all objects that the

popped object references are traced. Once the gray stack is empty, the sweep phase is allowed to start. The sweep phase frees objects that are unmarked (white or light gray) and resets the mark bit for all other objects, making them white colored. All objects that survive a collection step are colored white.

To prevent that the sweep phase has to iterate the whole heap, the proposal suggests, that the mark bit for each object is stored in a separate metadata area. As the sweep routine only has to look at the mark bit to determine whether an object is live, it no longer has to iterate the whole heap, but only the metadata area.

The gray bit on the other hand is stored in the object itself. This is mostly for caching reasons, as the write barrier always has to read and set the gray bit, which would pollute caches if the gray bit was stored in a metadata area. Storing the gray bit in the object itself leads to less cache pollution, as the write barrier is followed by a store to the object on which the write barrier is called, which would load the object into the cache anyway.

The proposal also describes a generational mode of the collector. Generational GC distinguishes between different generations of objects (usually two or more). Newly allocated objects belong to the first generation. When an object survives a collection step it is promoted to the next higher generation. Moving garbage collectors usually have different memory areas for each generation and copy the surviving objects to the area of the next generation after a collection step. As for many programs most objects die when they still belong to the first generation, this copying step significantly reduces fragmentation. Additionally, the garbage collector has the possibility to limit a collection step to a single generation (called minor collection), as opposed to collecting memory from all generations (called major collection). Minor collections usually only collect memory in the first generation and are a lot faster than major collections because they only operate on a subset of the heap.

The generational mode of the quad-color algorithm also distinguishes between major and minor collections. However, there is no dedicated memory area for young objects as the non-moving property prevents this. Thus, for both major and minor collections the whole heap has to be examined. The two collection modes have different sweep routines: Major collections use the sweep routine that the previous paragraphs already describe, while the sweep routine for minor collections frees unmarked objects but does not unmark marked objects. Thus, all surviving objects remain colored black. Section 4.5 will analyze whether the quad-color algorithm can benefit from the generational mode.

3 Implementation

In this chapter we discuss the implementation of the prototype. Where the implementation uses standard techniques, its description is kept short.

3.1 General

The prototype closely follows the specification in [3]. However the sequential store buffer and the gray queue were omitted, as they are optional optimizations. Further details about these features can be found in Mike Pall's proposal. Where the prototype deviates from the specification, the deviations are justified. For the rest of this thesis, the prototype will be referred to as QCGC.

QCGC is implemented as a C library and offers the following API:

- `qcgc_initialize` initializes QCGC.
- `qcgc_destroy` destroys QCGC.
- `qcgc_allocate` allocates a new object of a given size. Allocating a new object may trigger an incremental mark step or a sweep step.
- `qcgc_push_root` pushes an object to the shadow stack. The purpose of shadow stack will be explained below.
- `qcgc_pop_root` pops a given amount of objects from the shadow stack.
- `qcgc_write` is the write barrier.
- `qcgc_collect` executes a full garbage collection step.
- `qcgc_register_weakref` registers a weak reference. A weak reference initially points to an object and is set to `NULL` once the referenced object is collected.

The user has to provide the tracing method `qcgc_trace_cb` which takes an object `o` and a function `f` as its arguments and calls `f` on all objects `o` references.

3.2 Design

3.2.1 Roots

QCGC stores the root objects in a shadow stack, to which the mutator has to push all objects it considers to be roots. If an object is no longer considered to be a root, the mutator has to pop it from the shadow stack again. Additionally, QCGC supports so called prebuilt objects, which are immortal objects that are allocated outside of the garbage collector. Prebuilt objects are root objects and may contain references to other objects (prebuilt or allocated through QCGC). After creation, a prebuilt object can only point to other prebuilt objects, thus it is not necessary to trace them. When a reference in a prebuilt object gets updated, the mutator has to call the write barrier on it, which causes QCGC to register the prebuilt object as it may now contain references to normal objects. In mark steps all registered prebuilt objects will be traced to prevent QCGC from falsely identifying a live object that is only reachable through a reference from a prebuilt object as garbage.

3.2.2 Arenas

Like other garbage collectors, QCGC manages its memory in arenas, contiguous blocks of memory. An arena is organized in cells of 16 bytes each and naturally aligned. Memory is allocated in blocks, spanning one or more cells. The start of each arena is reserved for metadata, containing both the mark bitmap and the block bitmap. The former stores all mark bits for objects that are allocated from the arena, the latter indicates whether a cell is allocated or not. As we have to store two bits for each cell, the metadata overhead is $\frac{1}{64}$ of the total arena size. However, not all cells can be used to allocate objects, as some cells are used to store the metadata. The parts of the mark bitmap and the block bitmap that store the mark bits and block bits for these cells are used for arena management. The arena size is configurable in powers of two from 64KB to 1MB. The lower bound makes sure that at least 16 bytes of each arena are available for arena management, whereas the upper bound guarantees that the index of each cell fits into a 16 bit integer.

3.2.3 Allocation

QCGC internally uses two allocation modes, bump allocation and fit allocation. It switches between them, trying to find a good trade-off between memory usage, fragmentation and allocation speed. Section 4.4 explains when the allocation mode is switched. Both bump and fit allocator are implemented using standard techniques. The fit allocator uses segregated free lists with a best-fit allocation policy. If the appropriate free list for a request has no entries, higher size classes are searched, using a first fit policy and splitting the block. In case there is no block large enough to satisfy the request, more memory is requested from the operating system.

Unlike the implementation description in [3], QCGC always updates its free lists and does not search free blocks lazily. As a sweep step coalesces blocks, all free lists are rebuilt. This trade-off makes allocation faster and simplifies the implementation.

3.2.4 Collection

At the beginning of a mark phase, QCGC pushes all root objects the gray stack, which is then processed iteratively. The top object is popped, marked black and traced, causing all unmarked objects it references to be pushed to the gray stack.

Sweeping is non-incremental and sweeps all arenas. It can be performed as soon as the gray stack is empty. Consecutive free blocks are immediately coalesced and each free block is added to the corresponding free list. The sweep routine only has to access the arena's bitmaps to turn black objects white (changes the mark bitmap) and free unmarked objects (changes both bitmaps).

3.2.5 Huge Blocks

Blocks that are larger than a configurable threshold, which has to be smaller than the arena size, are allocated separately. Their metadata is stored in a hash table as their quantity is expected to be rather small. Marking huge blocks is no different to marking normal objects, but instead of modifying the mark bitmap, the metadata entry in the hash table has to be updated, which is more expensive. Sweeping iterates the hash table entries, resets the mark bits for marked objects, and frees unmarked objects.

3.3 Optimizations

When all functionalities of the prototype were implemented, it was integrated into PyPy. As this primarily is a technical step, details about the integration step are omitted.

Analyzing the QCGC performance when executing a GC benchmark from PyPy's set of benchmarks with the Linux profiling tool `perf` and `chachegrind` from the `valgrind` tool suite, several performance issues in the QCGC implementation were detected and mitigated. The following optimizations were implemented:

- Make the shadow stack fixed size
- Minimize the code size and complexity of the shadow stack push and pop methods.
- Minimize the frequency of metadata manipulation in the bump allocator
- Vectorize arena sweeping code using the bitmap tricks described in [3]
- Enable inlining of function calls by GCC
- Tune allocator switching heuristics
- No zero initialization of newly allocated memory

These optimizations made QCGC four times faster on the selected benchmark, without increasing overall memory usage.

4 Evaluation

After describing the algorithm and its implementation, we now evaluate the behavior and performance of the prototype. Special attention is paid to how fragmentation behaves, how minor collections influence overall performance, and whether the fourth color helps performance. Finally QCGC is compared to incminimark, PyPy's default garbage collector.

4.1 Benchmark Method

To benchmark QCGC, we compiled a version of the PyPy interpreter that uses QCGC as the garbage collector. Various Python programs from PyPy's benchmark collection were then executed using this interpreter. QCGC logs several events and internal values, such as memory usage, fragmentation, and memory utilization to a log file, from which we extracted the presented results. For all tests, the arena size is configured to 1MB, incremental marking happens when 1MB of additional memory is allocated and every tenth mark step is replaced by a full collection that marks all remaining live objects and executes a sweep phase. Allocations larger than 16KB are treated as huge blocks and do not count towards total memory usage, as they are not managed by QCGC but by malloc. These parameters are not optimized yet and chosen arbitrarily. Running the empty program in the PyPy interpreter using QCGC requires 12MB of memory in total.

PyPy is compiled with `-O2` and `-no-allworkingmodules` as QCGC does not support threads. The used C compiler is GCC 6.2.1 and all benchmarks were executed on an iMac 27" 2011 with an Intel Core i7-2600 CPU and 16GB of RAM, running Arch Linux.

4.1.1 Selected Benchmarks

The following Python programs were selected for the evaluation:

- `gcbench.py`: Dedicated benchmark for garbage collectors creating large binary trees.
- `nbody.py`: N-body simulation for N = 5.
- `nqueens.py`: Solves the n queens problem for n = 10.
- `spectral-norm.py`: Calculates the spectral norm of a 130×130 matrix.
- `table.py`: Creates a 1000×1000 HTML table as a list of strings.

Unless stated otherwise, all benchmarks are executed ten times and without restarting the interpreter between the executions.

4.2 General Performance

In this test we measure how long QCGC spends for GC tasks (marking and sweeping), execution time, memory usage, memory utilization, fragmentation, and GC pause time. The results serve as baseline values for the other evaluation steps. We measure fragmentation as $1 - \frac{\text{Largest Block of Free Memory}}{\text{Total Free Memory}}$ and utilization as $\frac{\text{Used Memory}}{\text{Total Memory}}$. GC pauses are the time frames of the program execution when the normal program (mutator) is halted for GC tasks like marking or sweeping.

As the results in Table 4.1 show, all average GC pause times are quite low, achieving the design goal of low latency. However, it seems that the pause time increases with higher memory usage, hence long pause times might become an issue for programs with a high memory usage. Section 4.3 takes a closer look at how the GC pause time behaves with increasing memory usage.

The extremely high amount of time spent for marking in `table.py` can be explained by the design of the benchmark. This benchmark creates a large list of strings that is constantly appended to. As a list is stored as a single object, changes to it trigger the write barrier on the whole list object, causing it to be traced over and over again. A possible solution would be a more fine grained write barrier that does not mark the whole object as dirty, but only parts of it. Evaluating this solution lies beyond the scope of this thesis.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Memory Usage (MB)	Max.	63	12	14	12	185
GC Pause (ms)	Avg.	1.76	0.21	0.23	0.19	35.52
	Max.	26.57	2.32	1.98	2.36	149.94
Time per run (s)	Total	72.34	0.99	60.04	0.99	36.83
	Mark	30.97	0.09	8.57	0.13	34.70
	Sweep	7.59	0.04	3.77	0.05	0.87
Fragmentation (%)	Min.	97.85	90.54	90.37	92.55	95.02
	Avg.	99.99	91.93	96.09	96.88	99.90
	Max.	99.99	98.05	98.70	98.74	100.00
Memory utilization before sweep	Min.	16.30	77.70	68.21	73.90	11.82
	Avg.	32.43	79.88	78.80	76.02	43.13
	Max.	100.00	100.00	100.00	100.00	100.00

Table 4.1: Baseline measurement results

Average memory utilization is reasonable for programs with low memory usage. As the memory usage increases, utilization gets worse. QCGC seems to be optimal concerning maximal memory utilization, as 100% is reached for all tests. A more detailed analysis of the memory utilization is provided in Section 4.4 together with more details about the fragmentation, which is quite high.

4.3 GC Pause

This section further investigates in how the total memory usage is related to GC pause time. To rule out differences caused by the nature of the executed benchmark, we modified `gcbench.py` to use different amounts of total memory and measured both GC pause time and how long the sweep phases are. Table 4.2 shows the results.

Memory Usage (MB)	Max.	19	31	52	188
GC Pause (ms)	Avg.	0.31	0.36	0.46	0.88
	Max.	4.72	8.01	20.88	100.79
Time per sweep phase (ms)	Max.	3.07	3.91	6.31	26.79

Table 4.2: Measurement results for `gcbench.py` using different amounts of total memory

We observe, that both the GC pause and the time per sweep phase increase with higher memory usage. While the pause time for marking depend on the parameters of the GC, the pause time for sweeping cannot be tuned by parameters. As sweeping is non-incremental by design and performed per arena, the time per sweep step will inevitably increase for programs with large memory usage. Thus, the design goal of low latency is not satisfiable for all workloads.

4.4 Fragmentation and Memory Utilization

4.4.1 Fragmentation

This test takes a closer look at the high fragmentation values measured in Section 4.2 and examines how often the fit allocator is used, whether using it more often would decrease the fragmentation, and what overhead the fit allocator has compared to the bump allocator in terms of execution time. When calculating the fragmentation, we do not count arenas that are completely free to the amount of free cells, as they could be returned to the operating system.

QCGC currently uses the bump allocator whenever there is a block of 256 or more cells available (at least 4KB). Since the average object size for all benchmarks is about five to six cells, this guarantees, that the bump allocator is used about 40 to 50 times. However, we did not evaluate whether the block size of at least 256 cells is optimal. We do not take fragmentation into consideration for allocator switching because the fit allocator has too little influence on fragmentation, as this section will show later.

The current switching strategy causes the size of the largest free block to decrease when the bump allocator is used, unless there are many large free blocks and the bump allocator can not use them faster as sweeping creates new large blocks. As the fit allocator always uses the smallest free block that satisfies the requested allocation size, it should reduce fragmentation (as the amount of free memory decreases while the size of the largest free block usually stays the same).

In a first step we thus check how often the bump allocator and how often the fit allocator is used. The results in Table 4.3 suggest, that higher usage of the fit allocator has no positive impact on fragmentation, as for both `gcbench.py` and `table.py`, which use the fit allocator often, all fragmentation values are high as well.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Allocator (% of allocations)	Bump	0.51	98.04	95.03	99.99	29.41
	Fit	99.49	1.96	4.97	0.01	70.59
Fragmentation (%)	Min.	97.85	90.54	90.37	92.55	95.02
	Avg.	99.99	91.93	96.09	96.88	99.90
	Max.	99.99	98.05	98.70	98.74	100.00
Time per run (s)	Total	72.34	1.07	60.04	0.99	36.83

Table 4.3: Used allocator, fragmentation and execution time

However, when executing all tests only using the fit allocator (see Table 4.4), the fragmentation behavior of the benchmarks that use the fit allocator often significantly improve. This can be explained by the bump allocator behavior: Normally, the fit allocator is only used when no large blocks of memory are available. Using the fit allocator often implies that no large blocks are available. When only the fit allocator is used, large blocks remain untouched as long as there are small blocks available, resulting in a higher size of the largest free block, thereby reducing fragmentation. For tests that rarely use the fit allocator, disabling the bump allocator does not improve the average fragmentation values.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Fragmentation (%)	Min.	84.91	86.89	85.46	84.22	79.22
	Avg.	97.06	93.91	96.21	95.81	91.86
	Max.	99.93	96.50	98.50	97.53	99.99
Execution time (s)	Avg.	65.12	1.25	71.33	1.34	45.10

Table 4.4: Fragmentation and execution time when only using fit allocation

As one can observe, the execution time actually increases for all tests except `gcbench.py`, so the lower fragmentation comes at the cost of slower execution time. The reason why `gcbench.py` is actually faster lies in both allocator design and high fit allocator usage: The allocator is optimized to make bump allocations fast, causing additional overhead when bump allocation fails. As

for `gcbench.py` bump allocations are extremely rare in the previous test, the additional overhead is paid for nearly each allocation. When only fit allocation is used, this additional overhead can be omitted, making the execution faster.

This analysis of fragmentation clearly showed that QCGC fails to achieve the design goal of tightly controlling fragmentation. Even always using the fit allocator fails to reduce the average fragmentation values below 90%.

4.4.2 Memory Utilization

As the average memory utilization for `gcbench.py` was quite low, we take a closer look at it in this test. Figure 4.1 shows the behavior of the reserved and allocated memory over time for a single execution. Reserved memory is calculated by multiplying the arena size with the number of arenas that contain live objects.

`gcbench.py` first stretches memory by creating a large binary tree, then creates a smaller, long lived tree, and finally builds many short lived trees of different heights. Figure 4.1 illustrates this behavior quite well. We observe that after stretching the memory the amount of reserved memory only briefly decreases before it is reserved again. Re-reserving this memory can be explained by the allocator behavior: QCGC reuses free arenas when no large blocks are available to be able to use the bump allocator more often, resulting in higher allocation speed as long as free arenas are available but decreases memory utilization. The fact that the amount of reserved memory never decreases implies that each arena contains some live objects. Again, having live objects in all arenas can be explained by the allocator behavior: When large blocks are available these are reused for the bump allocator, preventing an arena to become completely empty and trading memory utilization for allocation speed.

The observed behavior suggests, that QCGC will almost always keep live objects in an arena once it was requested from the OS due to allocator design. To test this, `gcbench.py` was modified to skip the stretching step. The findings in Figure 4.2 confirm that the allocator design causes the low utilization. The stretching step requires a lot of arenas and the allocator design prevents any of the arenas to become entirely empty, causing low overall utilization. Without the stretching step the average of all measured utilization values for a single execution is 90.73%, as opposed to 38.35% for a single execution with the stretching step. However, having more memory available does not seem to speed up the execution time, questioning whether the aggressive usage of the bump allocator

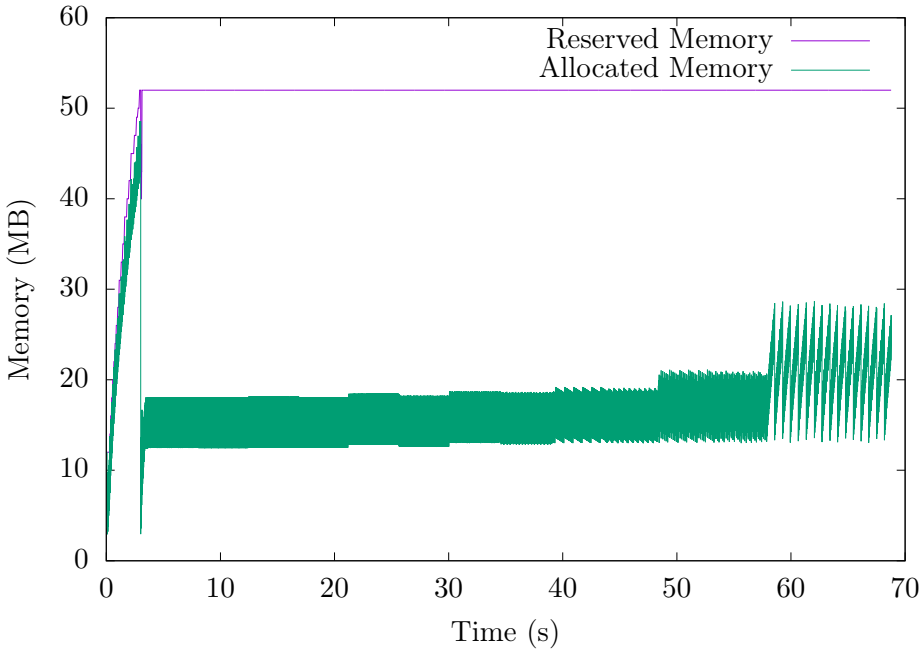

Figure 4.1: Behavior of reserved and allocated memory over time for a single execution of `gcbench.py`

is a reasonable design choice.

4.5 Minor Collections

For this test, we implemented the generational mode of the quad-color algorithm as described in Chapter 2. Unlike a moving GC, QCGC can not just trace a nursery or some other small area for its minor collections. Instead, all arenas have to be marked and sweeped like for normal collections. Like the non-generational version, collections are triggered once 10MB of additional memory are allocated. Every fifth collection is a major collection, the other collections are minor ones. Further tests would have to show whether these values could be further optimized, as they have been chosen arbitrarily.

We compare the generational mode to the results from Section 4.2. The results in Table 4.5 show that memory usage slightly increased. This increase is caused by the modified sweeping (compare Chapter 2): As the mark flag is not cleared, objects that were once marked remain marked until a major collection is carried out even tough some objects might have become unreachable in the meanwhile. As only every fifth collection is a major collection, memory usage temporarily

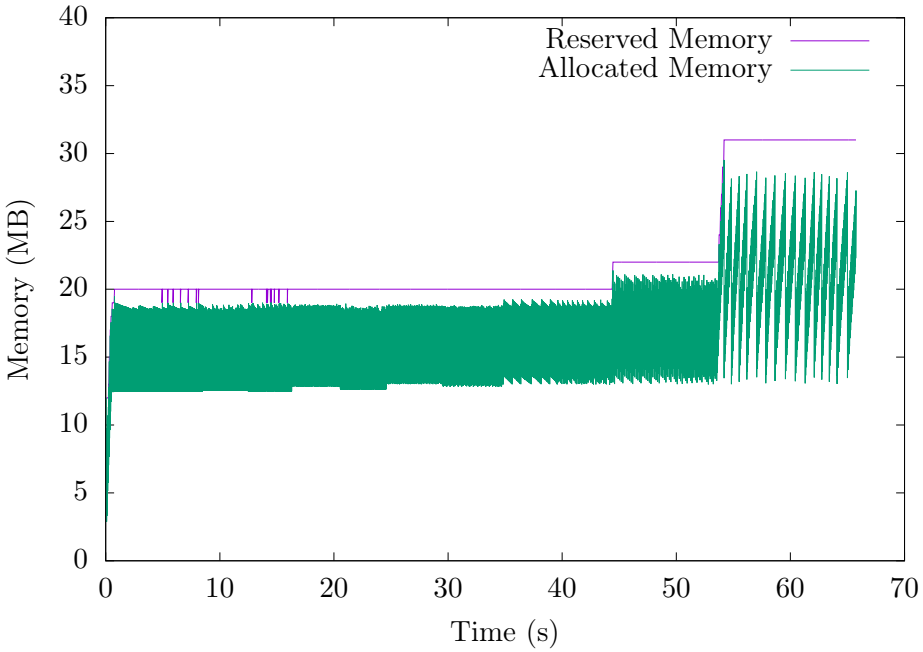

Figure 4.2: Behavior of reserved and allocated memory over time for a single execution of `gcbench.py` without stretching the memory

increases. The higher memory usage could be traded for higher GC overhead by triggering a major collection step more often.

While fragmentation and utilization did not change significantly, we observe a much lower marking overhead for all benchmarks except `table.py`. As mentioned above, the huge marking overhead for this program is caused by the inefficient write barrier. The lower marking overhead for the other programs comes from the minor collections: Since marked objects remain marked after a minor sweeping step, the incremental mark steps that follow the minor collection only have to mark and trace newly allocated objects and objects that contain modified references (dark gray colored). All other objects contain no updated references and are already marked. Thus they do not have to be traced in mark steps that follow a minor collection.

The sweep phase got slightly more expensive, which can be explained by both a less optimized sweeping algorithm for minor collections and the overhead of additional branches for the differentiation between minor and major mode. Both of these issues could be mitigated, but doing so lies outside the scope of this thesis. Altogether, the generational mode improves execution time with slightly increased memory usage, despite having to sweep all arenas.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Memory Usage (MB)	Max.	67	12	15	12	186
GC Pause (ms)	Avg.	0.67	0.12	0.13	0.11	34.01
	Max.	24.36	2.23	3.22	3.30	144.96
Time per run (s)	Total	49.44	0.91	56.45	0.86	35.69
	Mark	6.10	0.02	2.03	0.02	33.43
	Sweep	8.64	0.05	5.37	0.07	0.95
Fragmentation (%)	Min.	95.59	90.68	91.00	92.55	95.02
	Avg.	99.99	93.01	97.43	97.93	99.90
	Max.	99.99	98.12	99.31	99.13	100.00
Memory utilization before sweep	Min.	16.27	76.73	67.75	73.10	10.92
	Avg.	32.45	82.14	76.32	77.99	43.67
	Max.	100.00	100.00	100.00	100.00	100.00

Table 4.5: Measurement results when using the generational mode

4.6 Impacts of the Fourth Color

In this test we replace the quad-color algorithm by a variant of the tri-color algorithm and examine what influences on the behavior of QCGC this has. While studying the quad-color algorithm, we observed that the white color could be omitted, if black objects would be turned light gray after sweeping instead of white. The white color is superfluous because both marking and sweeping treat light gray and white objects the same. The write barrier turns white objects into light gray ones, while it does nothing for light gray ones. But this has no influence on the other operations as they do not distinguish between light gray and white.

We thus implemented a version of QCGC that turns objects light gray after sweeping, now following the state diagram from Figure 4.3. To prevent resetting the gray flag for each object when sweeping, we switch the meaning of the gray flag after each collection. Resetting the gray flags in the objects itself would nullify the advantage of the separate mark and block bitmap, as the sweep step would have to access each object instead of only the dedicated metadata area.

Again, we compare the behavior of the PyPy interpreter using the simplified version of QCGC (see Figure 4.6) to the results from Section 4.2. For all tests,

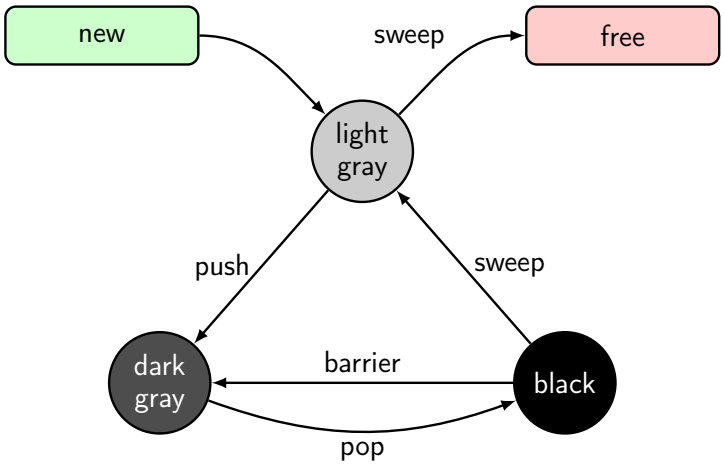

Figure 4.3: Modified state diagram

fragmentation, memory utilization and memory usage are nearly identical to the values observed without simplification. For `gcbench.py` the execution time became much worse, probably caused by the more complex check and initialization for the gray flag. For all other tests, the total execution time stayed more or less the same.

The results show, that while QCGC could use a tri-color algorithm, doing so does not improve the overall behavior of QCGC. Furthermore, only the state diagram is simplified, the actual implementation becomes more complex as checking whether an object is gray is no longer a simple flag lookup but has to check the meaning of the flag as well. So while the white color of the quad-color schema in fact is superfluous, it reduces overall complexity which makes the implementation simpler and even decreases the execution time for some programs.

The generational mode can also be implemented in this tri-color version of QCGC. Both major and minor collections work similar to their quad-color counterparts: Unmarked (here only light gray) objects are freed, major collections clear the mark bit (turning black object light gray), and minor collections do not clear the mark bit (leaving black objects black). As after a minor collection only black objects remain, the meaning of the gray bit must not be inverted, as this would turn these objects dark gray, which would require that they are pushed to the gray stack and nullify the advantage of minor collections. After a major collection the meaning of the gray flag is inverted, as all surviving objects are temporarily colored white and thus have to be colored light gray again.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Memory Usage (MB)	Max.	65	12	14	12	185
GC Pause (ms)	Avg.	1.93	0.19	0.13	0.19	35.22
	Max.	31.02	2.36	3.22	2.27	141.53
Time per run (s)	Total	81.58	0.92	62.68	0.99	36.55
	Mark	34.26	0.08	9.04	0.13	34.41
	Sweep	8.12	0.03	3.63	0.05	0.85
Fragmentation (%)	Min.	97.18	90.47	90.67	92.55	95.02
	Avg.	99.99	92.70	95.90	96.85	99.89
	Max.	99.99	98.14	98.70	98.72	100.00
Memory utilization before sweep	Min.	16.70	76.69	68.95	73.78	12.99
	Avg.	31.54	79.37	76.61	75.93	43.16
	Max.	100.00	100.00	100.00	100.00	100.00

Table 4.6: Measurement results for generational mode

4.7 Comparison to Incminimark

In a last test, we compare the execution time for each benchmark between QCGC using the generational mode and incminimark, PyPy's default GC. Incminimark is a moving, incremental, and generational GC. New objects are allocated in a dedicated, fixed size area for first generation objects (nursery), from which live objects are copied out to a area for second generation objects in a minor collection once the nursery is full.

		gcbench.py	nbody.py	nqueens.py	spectral-norm.py	table.py
Time per run (s)	Incminimark	19.20	0.71	37.58	0.71	1.10
	QCGC	49.44	0.91	56.45	0.86	35.69

Table 4.7: Execution time comparison

For all tests, using incminimark is between 1.21 and 32 times faster than using QCGC. The higher execution times for QCGC are probably both caused by

the inefficiencies that arise from memory fragmentation and missing important optimizations. For example, incminimark supports card marking, which heavily reduces the marking and overhead for large objects. Furthermore, incminimark benefits from years of parameter tuning, whereas QCGC's parameters are not tuned yet.

5 Conclusion

The evaluation showed that the quad-color algorithm itself is not able to solve one of the major issues that arises from non-moving GC, namely fragmentation. Moreover, the specification fails to address the problem of the coarse granularity of the write barrier, as the `table.py` benchmark showed (compare Section 4.2).

The GC pause time is low as long as the total memory usage does not become too large, but for programs with high memory usage, the non-incremental design of the sweep phase causes high latency for sweeping. To be able to compete with incminimark, QCGC needs an incremental sweeping mode. A possible implementation could allocate objects only from new arenas or arenas that already have been sweeped.

The generational mode is able to decrease the GC overhead significantly for most workloads, even though the quad-color algorithm is non-moving and thus cannot benefit from reduced sweeping overhead. Thus, we recommend to activate the generational mode by default.

Compared to PyPy's incminimark, the current version of QCGC cannot keep up in terms of performance. However, there is still room for improvements. On the one hand, the codegen for PyPy could exploit the non-moving property of QCGC, on the other hand, the evaluation showed that QCGC could benefit from a different allocator design, and a more fine grained write barrier. QCGC could further benefit from tuning its parameters for performance, as all parameters are not tuned yet. We think that implementing these changes could make QCGC nearly as fast as incminimark.

5.1 Related Work

The Boehm-Demers-Weiser conservative garbage collector [1] is a non-moving, incremental, and generational garbage collector. However, it is conservative,

while QCGC is precise. While precise garbage collectors exactly know to which objects a given object points, conservative garbage collectors have to *conservatively* guess, whether a given word of memory is a pointer to an object or whether it is a value. They thus might suffer from memory leaks if a program stores a value that looks like a pointer. Additionally, deciding whether a word of memory looks like a pointer causes overhead compared to precise GC. However, conservative garbage collectors require no help from the compiler or the VM to trace an object and thus can be used with compilers and VMs that were not designed to be used with a garbage collector.

In [7], Ueno, Ohori, and Otomo present a non-moving and generational garbage collector that is designed for functional languages. Its allocator splits the heap into sub-heaps which are used for different allocation sizes, which causes it to be immune against fragmentation: As each block can only be used for objects of the same size, the actual size of a block cannot get smaller. Future iterations of QCGC could try to adopt this allocator design.

Bibliography

[1] Hans-J. Boehm and Alan J. Demers. *A garbage collector for C and C++*. URL: http://www.hboehm.info/gc/ (visited on 10/02/2016).

[2] Edsger W. Dijkstra et al. "On-the-fly Garbage Collection: An Exercise in Cooperation". In: *Commun. ACM* 21.11 (Nov. 1978), pp. 966–975. ISSN: 0001-0782. DOI: 10.1145/359642.359655. URL: http://doi.acm.org/10.1145/359642.359655.

[3] Mike Pall. *New Garbage Collector*. URL: http://wiki.luajit.org/New-Garbage-Collector (visited on 09/10/2016).

[4] Mike Pall. *The LuaJIT Project*. URL: http://luajit.org/ (visited on 10/02/2016).

[5] *PyPy*. URL: http://pypy.org/ (visited on 10/02/2016).

[6] *Python*. URL: https://www.python.org/ (visited on 10/03/2016).

[7] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. "An Efficient Non-moving Garbage Collector for Functional Languages". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 196–208. ISSN: 0362-1340. DOI: 10.1145/2034574.2034802. URL: http://doi.acm.org/10.1145/2034574.2034802.