KTH Computer Science
and Communication

Evaluating and improving remembered sets in the HotSpot G1 garbage collector

ANDREAS SJÖBERG
ANSJOB@KTH.SE

Master's Thesis at CSC
CSC Supervisor: Mads Dam
Examiner: Johan Håstad
Project provider: Oracle Svenska AB
Oracle Supervisor: Jesper Wilhelmsson

TRITA xxx yyyy-nn

Abstract

In this master's thesis project the remembered set implementation in Java HotSpot's implementation of G1 is evaluated. It is verified by benchmarking that using Bloom filters can reduce region scanning times when used as an intermediate data structure between card-precision bitmaps and region coarsening. It is shown that iterating the Bloom filter is made faster by combining binary trees with the Bloom filters. It is also verified that using a more narrow integer type, with an added bitmap to keep track of null entries, will decrease the memory footprint caused by remembered sets. Both modifications to the current implementation cause application throughput regressions in SPECjbb2013.

Referat

Utvärdering och förbättringar av remembered sets i Javamaskinen HotSpots skräpsamlare G1

I det här examensarbetet undersöks implementationen av remembered sets i HotSpots implementation av skräpsamlingsalgoritmen G1. Det bekräftas genom prestandamätningar att användandet av Bloomfilter kan minska tidsåtgången för regionsavsökning. Detta när Bloomfiltret används som en mellanliggande datastruktur mellan bitmappar på kortnivå och bitmappar på regionsnivå. Det bekräftas också genom prestandamätningar att iterering över Bloomfilter kan snabbas upp genom att kombinera filtret med ett binärt sökträd.

Vidare visas det att användandet av en heltalstyp med mindre räckvidd, tillsammans med införandet av en bitmapp för att registrera nullvärden, kan minska minnesanvändningen som remembered sets medför. I den standardiserade prestandamätningen SPECjbb2013 medför båda förändringarna dock prestandaförsämringar.

Preface

No master thesis project is a single person effort. I would like to take this opportunity to thank some of the people who made this work possible. First of all, I would like to thank my supervisors Jesper Wilhelmsson and Mads Dam for guiding me through the process of writing this report, carefully proofreading and keeping me on the right track. I am also grateful to my colleagues at the Oracle garbage collection team for making me feel welcome, and many interesting discussions about garbage collection topics. I would also like to extend a thank you to Oracle for sponsoring this project.

A big thank you to my girlfriend Emma, for being supportive and listening to my late night ramblings about false positive rates and iteration speeds. Last, but definitely not least, I would like to thank Marcus Larsson for being the best lab partner imaginable for the last four years.

Contents

1 Background **1**
1.1 Java . . . 1
1.2 Memory management . . . 1
1.2.1 Manual memory management . . . 2
1.2.2 Memory allocation . . . 2
1.3 Reference counting . . . 3
1.3.1 Reference counting in file systems . . . 3
1.4 Garbage collection . . . 3
1.4.1 Concurrent collectors and stop-the-world . . . 5
1.4.2 Finding the live set . . . 6
1.4.3 Reclaiming garbage . . . 7
1.5 Generational garbage collection . . . 8
1.6 Remembered sets . . . 9
1.7 Garbage first garbage collection . . . 10
1.7.1 Collection set selection . . . 10
1.7.2 G1 remembered sets . . . 11

2 Previous research **13**
2.1 Card tables . . . 13
2.1.1 Hierarchic card tables . . . 14
2.1.2 Card table hybrids . . . 15
2.2 Log based barriers . . . 15
2.3 Hash table implementations . . . 15
2.4 HotSpot G1 Write barriers . . . 16
2.5 HotSpot G1 Refinement . . . 16
2.6 The HotSpot G1 remembered set implementation . . . 17
2.6.1 Set representation . . . 17
2.6.2 Remembered set memory management . . . 18
2.6.3 Insertion operations . . . 18
2.6.4 Insertion concurrency . . . 21

3 Improvement suggestions **23**
3.1 Problem statement . . . 23

3.2 Reducing size of sparse lists 23
3.3 Using Bloom filters 24
3.3.1 Where to use Bloom filters 25
3.3.2 Only asking the relevant questions 25
3.3.3 Compact card set sizing 26
3.3.4 Managing false positives 27

4 Evaluation methodology **29**
4.1 Measurement methods 29
4.2 System setup 29
4.3 Benchmarks used 30
4.3.1 The LRU cache benchmark 30
4.3.2 SPECjbb2013 31

5 Results **33**
5.1 Compact card set iteration speed 33
5.2 The LRU cache benchmarks 35
5.2.1 Worst case performance 35
5.2.2 Template approximation performance 38
5.3 SPECjbb2013 39

6 Conclusions **41**
6.1 Recommendations 41
6.2 Suggestions for further research 42

Bibliography **43**

A Compact card set benchmarks **47**

B LRU cache benchmark **51**
B.1 BenchMarkResults.java 51
B.2 DomainItem.java 52
B.3 RequestThread.java 53
B.4 Runner.java 56
B.5 Utils.java 60

Chapter 1

Background

1.1 Java

Java is a statically typed object oriented programming language which is executed by a runtime called the Java Virtual Machine (JVM). Many such JVM implementations exist, one of which is HotSpot from Oracle [16]. Java code is compiled into byte code which is executed by the JVM. Because of this, Java code can be compiled once, and executed everywhere a JVM can run.

Portability and ease of development are two commonly argued advantages of Java. The portability comes from the fact that the JVM provides a consistent interface to the application programmer, by executing abstract bytecode. The argued ease of development has many causes, one of which is automatic garbage collection.

1.2 Memory management

Most data processed by an application are stored in memory at some point. When the size of this data is known at the time of compilation, memory can be allocated at compile time. This is what is known as static memory allocation, and can take place in the data area or stack frame depending on the context.

Dynamic memory is the opposite of static memory in the sense that its size is unknown at compile time. The dynamic memory has to be managed at runtime, and is typically allocated on the heap. Most applications allocate new data while running and will need to return dynamically allocated memory. If this was not done, eventually all the available memory would be allocated, and no new allocations could be done. This is what is known as a *memory leak*. When memory is returned, it is either simply considered available for future allocations, or returned to the operating system to be used by other applications.

1.2.1 Manual memory management

In some languages returning memory is done through explicit function calls such as calling `delete` in C++ or `free` in C. According to Jones et al. [14] this can potentially cause at least two problems:

- Memory can be returned too early, while other parts of the application still has references to it. These references are then known as *dangling pointers*, which can cause hard-to-detect and hard-to-fix program defects.
- The last pointer to a memory location can be overwritten. The memory can then never be returned. This is a resource waste that is hard to recover from, and eventually the available memory can run low [17]. It may also cause performance to suffer because the allocation algorithm will have to work harder to find available locations.

These problems are sometimes solved by making a module *own* a memory location, and be responsible for deallocating it. This means that modules have to cooperate in terms of transferring ownership, and making sure that nobody will ever use the memory location after its return. Such an approach leads to complex interfaces between software modules, and the complexity seems to be inherent in the problem. The reason that the problem is complicated is that "liveness is a *global* property, whereas the decision to call `free` on a variable is a local one" as stated by Jones et al. [14].

1.2.2 Memory allocation

Memory is usually allocated in one of two ways; from a list of available memory regions known as a free list, or from a continuous area. The free list is usually a linked list of memory regions. Each region simply contains its size and a pointer to the next available region. Memory allocation then consists of picking a region from this list, and using it. Returning memory simply inserts a region into the list.

Different schemes can be used when picking the region to use. One common strategy is to pick the smallest region that is large enough for the allocation (Best Fit). Another strategy is to pick the first encountered region that is large enough (First Fit). When inserting a memory region into the free list, the inserter may be able to decide to coalesce two regions if their memory areas constitute a single continuous area. Whether to coalesce or not is a policy decision which can depend on application and allocation strategy.

When allocating from a single continuous free area, the end pointer to this area is simply increased by the amount of memory that should be allocated. Let the pointers $free$ and top point to the beginning and end of the available area, respectively. Consider an attempt to allocate n bytes. The only operations needed are then to check that $top - free > n$, and then increment $free$ by n bytes. Obviously, the top could be decremented just as well, thus consuming the free area from the other end. This kind of allocation is known as *bump-the-pointer* for obvious reasons.

In multi threaded environments, it is very possible to get high contention for incrementing the `free` pointer. The simple approach to this is for each thread to allocate a larger chunk of memory known as a thread local allocation buffer (TLAB). This approach is used in HotSpot. Object allocations are then made from these local buffers of available memory, and the thread does not have to contend for the `free` pointer as often.

1.3 Reference counting

One simple approach to avoid dangling pointers is reference counting. Reference counting is done by associating a counter with each memory location that is pointed to. When a pointer to the location is created, the counter is increased. When a pointer to the location is deleted or overwritten, the counter is decreased. If the counter reaches 0, the memory can be reclaimed.

The argued advantage of reference counting is that the decision to collect memory is local to the thread modifying the pointer, and the low memory overhead. However, reference counting has some major disadvantages. One is that the counter must be modified atomically, which means using expensive synchronizing primitives like compare-and-swap (CAS) on each pointer store. The other, and most important reason, is that reference counting cannot collect circular structures. Consider for example an object only containing a single pointer that points to the object itself. This counter will never reach 0, which means memory leaks can occur.

1.3.1 Reference counting in file systems

Reference counting can be used successfully if a data structure is guaranteed not to contain cycles. An example is UNIX-style file systems. In a UNIX-style file system the directory structure is separate from the file data. The directory structure just contains an index into a table, where the table has information about where on the disk to find the file. This table entry is known as an *inode*. Several directories can keep references to the same inode, meaning that they refer to the same file. These references are known as *hard links*. The inode then contains a reference counter, allowing the disk blocks used by the file to be deleted when there are no references to the file anymore.

The reason that reference counting works well in this context is that most implementations do not allow the user to create hard links to a directory, but only to files. This means that there cannot be any cycles of files referring to each other.

1.4 Garbage collection

To circumvent the problems presented in Section 1.2.1, automatic dynamic memory management can be employed in the form of a garbage collector (GC).

The most intuitive definition of what is garbage in an application, would be any dynamically allocated memory that will never be accessed in the future execution of the program. However, this definition suffers from a problem. Determining whether a memory location is accessed is equivalent to the halting problem.

Theorem 1. *No program $H'(X, L)$ can tell, for any program X and memory location L, whether the label L is referenced on the execution of X.*

Proof. From H' construct H which solves the halting problem. Insert a dereference of a special location L at the end of the program, and replace all instances of `halt` instructions with dereference instructions to L. $\square$

Because of Theorem 1, a conservative approximation is used to get a useful definition of garbage. The conservative approximation is that a memory location is considered garbage if it is not *live*. A memory location l is considered live if it is reachable through some series of references $l_0 \rightarrow ... \rightarrow l$, and l_0 is contained in the *root set*. The root set consists of the memory locations immediately reachable through pointers from all currently running threads [14]. Examples of things in the root set in Java are local variables, method arguments and static variables.

In Figure 1.1 the root set is $\{A, X\}$, the live set is $\{A, X, Y\}$, and the garbage is $\{W, Z\}$. Note that both the object graph and the garbage can contain cycles of references.

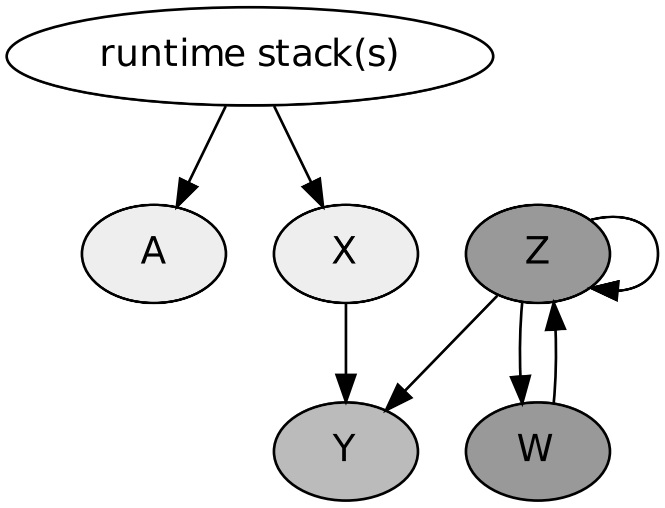

Figure 1.1. Example of a small object graph. The nodes are objects, and the edges represent references. Note that the graph is directed.

A garbage collector uses some algorithm to determine what objects are garbage and makes some or all of them available for allocation. Some interesting properties of garbage collectors are [17]

- promptness — how soon after an object has become garbage it is collected
- completeness — a complete garbage collector ensures that eventually all garbage will be collected
- memory footprint — the amount of memory needed for bookkeeping, not usable by the application
- overhead — typically measured in the percentage of CPU time spent doing garbage collection
- pause time — the length of time periods where the application threads are stopped due to garbage collection.

Garbage collectors typically need to employ trade-offs between at least these factors. Applications with different needs may need to specify a suitable GC implementation. Another problem is that of fragmentation.

One thing worth noting is that algorithms developed to do garbage collection of memory locations can be used to reclaim other resources as well. One example is the `VACUUM` command in some SQL implementations [2, 3] which reclaims unused disk space from deleted table rows in databases.

1.4.1 Concurrent collectors and stop-the-world

Garbage collection could be done while all the program threads (also known as *mutators*) are stopped. This method is known as stop-the-world (STW). STW greatly simplifies garbage collection, since no unexpected modifications can be made to objects while the garbage collector is running.

Garbage collectors that execute concurrently with the mutators are known as concurrent garbage collectors [14]. Concurrent garbage collectors have the advantage of causing shorter (if any) pauses in the running program. This is obviously an advantage to applications that need low response times. The concurrent collectors however, need to do some synchronization and deal with the fact that mutators can modify the object graph during a collection. This means that the concurrent collector typically uses more CPU time than the STW collectors. The HotSpot concurrent garbage collectors have small stop-the-world phases in the beginning and end of a garbage collection.

Which type of garbage collector to use will depend on the running application. For example web servers may require low pause times, while batch processing or scientific computations would benefit from a good high throughput STW collector.

STW garbage collectors can still use concurrency in the sense that multiple threads can be used for garbage collection [17]. This kind of concurrency is called parallelism in the context of garbage collectors. For example, a STW garbage collector that uses several threads is usually called a parallel garbage collector.

1.4.2 Finding the live set

The definition of garbage leads to a rather intuitive method for finding the live set; simply start at the root set and traverse the object graph. This can be done using a graph traversal algorithm such as Breadth First Search or Depth First Search. During the traversal, the objects are marked as live, which is why this is known as the *marking phase* of a garbage collection.

One thing to note here, is that a naive implementation of either BFS or DFS using a stack or queue could potentially use as much memory as the entire heap. This is clearly not acceptable, but can be solved using a technique called pointer reversal [4].

Tricolor marking

Traversing the live set is obviously easier when the mutators are all stopped and the graph is static. Collectors tracing the live set while the application threads are executing (concurrent collectors) need to deal with modifications that the mutator threads introduce to the object graph. The tools the collectors have at their disposal are typically read and write barriers. A barrier consists of code inserted into the runtime that calls the collector whenever a reference is read or updated.

Concurrent collectors usually deal with modifications to the object graph by designating a color to each object. This abstraction was introduced by Dijkstra et al., and is considered fundamental to the field of garbage collection [11].

- Black objects are known to be reachable from the root set, and have been processed
- Grey objects are known to be reachable from the root set, and have yet to be processed
- White objects have not yet been visited by the marking algorithm.

Initially the root set is colored grey. This is usually done in a STW phase because the runtime stacks have to be inspected.

Then the concurrent marking thread(s) scan each grey object for references. A marking thread will for each reference out of an object add the referred object into the grey set, and then mark the object as black (visited). The marking is stopped when there are no grey objects left. Then, the objects which are still white can be concluded to be garbage. To ensure that the mutators do not modify the object graph in such a way that live objects are lost, concurrent collectors maintain the weak or strong tricolor invariant. Doing so safely over-approximates the live set.

The *weak* tricolor invariant states that “all white objects pointed to by black objects are reachable from some grey object through a chain of white objects.”. The *strong* tricolor invariant states that “there are no pointers from a black object to a white object” [11].

Maintaining the weak tricolor invariant results in a category of algorithms which is called *snapshot-at-the-beginning* (SATB). Maintaining the strong invariant results in another category of algorithms known as *incremental update* [20]. These differ in how the write-barriers need to work. SATB algorithms need a write-barrier that triggers before a pointer store, while incremental update algorithms use a write barrier that is triggered after a pointer store.

1.4.3 Reclaiming garbage

When the live set has been found (or over-approximated), everything else is inferred to be garbage and should be reclaimed. Three major approaches exist to doing this, each with their own advantages and drawbacks.

Sweeping collection

One way to reclaim garbage is to iterate through the entire heap and find all the locations not visited in the marking phase. This is known as a *sweeping collection.* These locations are then added to a free list. The advantage of this approach is that to reclaim an area of memory, all that has to be done is to update some pointers.

The biggest disadvantage of sweeping collection (also known as mark-sweep GC) is that it can cause fragmentation. This happens when there are lots of slots available for allocation, but none large enough for a big continuous allocation of a large object like an array. For example in Figure 1.2, no allocation that requires ≥ 2 continuous memory cells can be made.

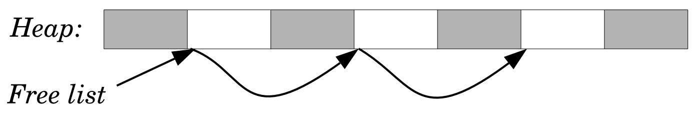

Figure 1.2. Free list after memory has been collected by mark and sweep. The gray boxes represent the live (marked) objects. Note that the free list pointers can be stored in the free memory areas, meaning little bookkeeping overhead.

Compacting collection

Compaction can intuitively be thought of as moving the live set to the beginning of the heap, and then claiming whatever is left as available for future allocations. Further allocations are then done by bumping a pointer. Figure 1.3 shows a compaction.

Compacting the live set has the advantage that is tackles the problem of fragmentation. Since all the available space is collected into one continuous area, fragmentation does not occur in the same way as for the sweeping collection. The obvious drawback is that copying objects can be slow. An object that is long-lived could be copied several times, adding to this problem.

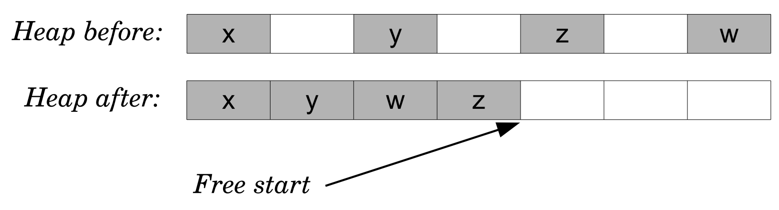

Figure 1.3. A possible way to compact the heap. Note that the order of the objects is not necessarily maintained.

Cheney style copying collection

Cheney style copying collectors does the marking and reclamation of objects in one single phase [7], making it possibly perform better than the compacting collectors. The available heap space is divided into two equally sized semi-spaces. One of the semi-spaces is known as the *from*-space and one is known as the *to*-space. Cheney style collectors are sometimes also known as semi-space copying collectors.

Cheney style collectors trace the object graph, and copy all the live-objects in the from-space into the to-space. When this is done, the roles of the two semi-spaces are swapped in preparation for the next collection.

One advantage of a Cheney style collector is that it only traverses live objects, making collection time proportional to the size of the live set and not the heap size. Another benefit can be good locality properties. This can cause performance gains due to decreased cache misses [14]. To get good memory locality, the collector can traverse the live set depth first or breadth first. Which traversing algorithm to choose depends on the application, and could be configurable. One major disadvantage of Cheney style collectors is that the object graph can only occupy half the size of the heap. Another disadvantage is the fact that long-lived objects are be copied back and forth between the two semi-spaces.

1.5 Generational garbage collection

Generational GCs rely on the generational hypothesis that object lifetimes are mostly short [14, 15]. This hypothesis leads to favoring young objects as candidates for collection. This can be achieved by using different areas called *generations* within the dynamic memory.

Let $G_1, ..., G_n$ be the generations. Then all objects in G_k are considered senior to objects in $G_1, ..., G_{k-1}$. In the common special case where $n = 2$ this can be simplified by calling the two generations the *young* and *old* generation. From now on, only generational GCs with two generations are considered, unless otherwise specified, since this is what is used in HotSpot [17].

When an allocation is attempted and the young generation is used up, it is collected. The collection traces the live objects from the root set, but does not follow references that leave the young generation. This is known as a *young collection* (YC) and is the most common kind. The root set in a YC includes any pointers into the young generation from the old generation. A remembered set is used to find these pointers into the young generation.

During a YC, the HotSpot implementations avoid promoting every single live object to the old generation [17]. This is because if a YC promoted all the live objects in the young generation, some objects that were allocated recently would be promoted without having a chance to become garbage. Avoiding this can be done by keeping a counter within the object, and incrementing it for each young collection that the object survives. After an object has survived a certain number of times, it is promoted to the old generation [17].

A young collection usually makes enough memory available in the young generation to facilitate the allocation that was attempted, according to the weak generational hypothesis. If, however, a young collection is not enough to facilitate an allocation, or the old generation is full, an old collection has to be performed. This can be done using a number of techniques, for example by compacting or doing a mark-sweep in the old generation [17].

1.6 Remembered sets

In generational garbage collection, it is assumed that objects in the young generation are likely to become garbage quickly. It follows that objects in the old generation are not very likely to refer to objects in the young generation. It is however not impossible for old objects to refer to newer ones. Consider a linked list's `append` operation for example, the `next` pointer of the last node is updated to point to the object that should be appended.

When doing a YC, to ensure no live object is collected, all objects in the old generation are assumed to be live. The collector would then in theory have to add the entire old generation to the root set in order to not collect live young objects. This could be done by scanning the old generation looking for references into the young generation. Since the old generation could be significantly larger than the young generation, this would defeat the purpose of only having to look at a small amount of memory.

A *remembered set* is used to avoid scanning the entire old generation. This set consists of areas within the old generation. An area is inserted into the set if it has objects possibly referring into the young generation. In other words, the remembered set keeps track of where to find objects in the old generation that may contain references into the young generation.

During a YC, the remembered set is inspected, and all the young objects referred to from the old generation are considered roots. When moving young objects, the remembered set is used to find all places the object is referred to and update those references. Objects can be moved either within the young generation, or by being promoted to the old generation.

A problem that can arise from using generational garbage collection is if old objects become garbage, but there is a long time before the next old collection. If an old object points into the young generation and the old object is garbage, the young object will not be collected during a YC. These objects are referred to as *floating garbage*. The effect could be compounded if the young floating garbage is eventually promoted into the old generation.

1.7 Garbage first garbage collection

Garbage first (G1) is a garbage collection algorithm that can be run in a generational mode [9]. One objective of G1 is to keep down pause times for applications with large heaps [19]. The key is that G1 should rarely (ideally never) need to do a slow, expensive collection of the entire old generation. In order to do this, the generations are made up of independent memory areas called regions. See Figure 1.4 for an example of this.

When a garbage collection is triggered, the young regions and *some subset* of the old generation are collected. Note that the subset can possibly be empty. The union of the young generation and the selected regions from the old generation is referred to as the *collection set* [9]. When the collection set contains some regions from the old generation, the collection is said to be *mixed*.

The collection consists of copying the live objects from the collection set into available regions, making the collection set regions available for future allocations. This is known as an *evacuation* [9].

1.7.1 Collection set selection

So, which ones of the old regions should be included in the collection set? The intuition is that the old regions with the most garbage should be favored (the *garbage first*). Selecting the collection set is done by computing for each region an estimate on the "price" of evacuating the region. The estimate takes into account among other things the fraction of the region which is live, and the amount of references into the region. The regions are then selected cheapest first until the total "budget" is spent [9]. The budget is derived from a number of factors, the most notable one being the pause time goal.

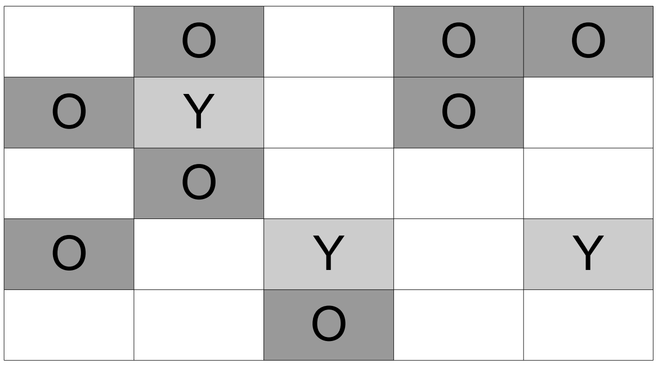

Figure 1.4. Example of a G1 heap. Each table cell represents a region. The regions marked with *O* and *Y* make up the old and young generations, respectively. The white regions represent empty, available regions.

1.7.2 G1 remembered sets

The G1 implementation in HotSpot is generational and uses remembered sets. Unlike traditional generational GCs, G1 uses one remembered set per region, instead of one per generation. This is because any region in the old generation can be collected, which means that references into it from the rest of the old generation need to be found quickly. This increase in the number of remembered sets means that the space overhead caused by them becomes more important.

Remember that the remembered set represents locations that point *into* a given region. This region is known as the *owning* region when discussing remembered sets.

One way to reduce the size of remembered sets is to adjust the granularity. In the most fine-grained mode, a remembered set could point to each field of each object that points into the region. The advantage of this approach would be that the incoming references could be found quickly.

The other extreme would be for a remembered set to simply point to regions containing objects that points into the region. During evacuation, the entire referred region would then need to be scanned for references pointing into the region to be evacuated. The current implementation in HotSpot starts out with a fine-grained model and switches over to a coarse-grained model when the remembered set becomes sufficiently saturated.

In this thesis, the usage of remembered sets in G1 is explained, and alterations to the implementation in HotSpot are evaluated.

Chapter 2

Previous research

Some early approaches to remembered sets included cooperation from the hardware or operating system. An example of a hardware supported solution is tagging memory locations as containing data or pointers and using write barriers to detect intergenerational memory writes [18, 23]. Using a bit to tag memory locations as pointers reduces the addressable memory by half. However, this has become feasible again with the transition to 64-bit architectures [8].

The virtual memory system can be used to keep track of interregional pointers [13]. After a collection, all pages are marked as protected, and subsequent writes will trigger a trap handler. The trap handler can record the page into a list of pages to be looked at later, and release the protection of the page. Any subsequent writes will then come at no overhead cost. The collector can then scan the marked pages to find inter-regional pointers.

This approach has the significant drawback of locking the precision level to the page size of the operating system, which is not very flexible. Another problem is that value writes will trigger the trap handler, and dirty the page, causing unnecessary scans of pages. This would mean for example that a write of a floating point variable would cause all pointers on the same memory page to be inserted into the remembered set. Hardware assisted approaches are not examined as part of this project.

2.1 Card tables

Card tables is a software equivalent of page protection [24]. The heap is divided into equally sized areas known as cards. Pointer stores into a card will mark the card as dirty. The card markings are kept in a continuous array known as the card table. Finding a card index is then as simple as getting the first n high bits of the target address of a pointer write. The insert into the card table is done in software, which means that only pointer-stores would cause a card to be dirty. This is because the write-barrier would obviously only be triggered on a pointer store and not a value store.

Since the card size is software defined it is more flexible to tune for performance. Tuning the card size is a difficult problem. A smaller card size means that the collector will more quickly find the interregional pointers, once looking at a card. However, small card sizes lead to large card tables, which is a problem because it uses more memory and takes longer to scan through.

One problem with card tables is that objects can span cards. When scanning a card, the collector will have to know where to start, and where to look for references. Where to start is typically stored in a card offset table.

Consider Figure 2.1. Assume each row represents two words. Then the first object starts 2 words after the card beginning, thus the card offset table for the given card has the entry 2. The card offset table will have to be maintained when objects are allocated.

The beginning of an object in HotSpot will have a mark word and a reference to the class which the object belongs to. The class typically holds the information about the object size and where to find fields that contain references.

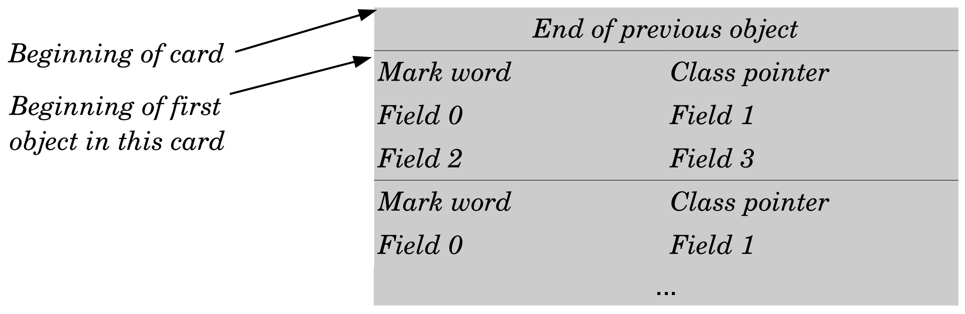

Figure 2.1. Example of how the beginning of a card could look like in memory. The lines separate objects, not cards.

2.1.1 Hierarchic card tables

With growing address spaces, the card tables can become quite large. When the table is sparse, a significant amount of time can be spent just looking at clean entries. Since the card table is scanned during a STW phase, CPU time is very precious.

It would be desirable to be able to take larger strides when scanning the card table in the common case of clean cards. Keeping track of when a large stride is possible can be done in a separate, smaller table [21]. The scanning would then consist of scanning the smaller table, allowing large parts of the actual card table to be skipped quickly. When a large stride cannot be made, the regular card table would then be inspected to find the dirty cards.

An analog to this in terms of the G1 remembered sets would be to dirty a region whenever one of the cards are dirty. All the cards in the same region could then be skipped by examining the marking for the region.

2.1.2 Card table hybrids

Card tables can be combined with other implementations of remembered sets as proposed by Hosking and Hudson [13]. Their approach keeps the highly efficient write barrier used by the card tables. At collection time, each dirty card is scanned, and any pointers into the young region are inserted into the actual remembered set. The remembered set is then considered part of the root set in the usual way. Before the mutators are restarted, the card table is cleared.

2.2 Log based barriers

Since reference write barriers are triggered often, it is important to make them cheap. Making it scalable means that the write barrier should avoid using atomic operations such as CAS or taking a lock. For this reason Detlefs et al. [10] introduce the concept of a log based write barrier.

Each thread has a local log buffer which it fills with entries, which are "locations which have been the left hand side of a pointer assignment" [10]. When the log is full, it is appended to a global list of full log buffers. Then, one or more background threads can process the global list of logs, and make the appropriate insertions into the remembered sets.

This decoupling of the remembered sets from the write barriers increases the freedom on how to implement the remembered sets. In particular, it allows for slightly slower insertion operations since they are performed in the background and not on every pointer store. The price is obviously the memory overhead caused by the log buffers. It should be noted also that before the collection can start, the remaining buffers have to be processed in a STW phase. This is because the remembered sets need to contain all the insertions before the garbage collection can start. The background threads' job is to minimize the amount of work for remembered set insertions during the STW phase.

2.3 Hash table implementations

Since the remembered sets are supposed to represent a set of memory locations, one straight-forward approach is to use a hash set. This was done successfully by Hosking et al. [12, 14]. Using linear hashing is a good method for letting the hash set grow in a smooth way without large pauses that copy and rehash all the data [14].

An obvious advantage of the hash sets is that they are very precise. Each referred object is inserted and can simply be found by iterating through the hash set. No scanning of cards or pages is necessary, though it can happen that the reference has been updated multiple times, making the lookup unnecessary. This could be circumvented by deleting from the hash set when a pointer is set to null for example, but this would possibly cause a too big overhead on the mutator.

A disadvantage of hash table based remembered sets is that they grow unbounded in size, possibly causing more memory overhead than is acceptable.

2.4 HotSpot G1 Write barriers

The HotSpot implementation has two write barriers: one that precedes the pointer store, and one that executes afterwards. The pre-write barrier is not the remembered sets' concern. It has to do with the marking algorithm, which is a snapshot-at-the-beginning (SATB) implementation. The part of the write barrier in HotSpot G1 that is interesting from a remembered set point of view is the post-write barrier.

The post-write barrier is a modified implementation of Detlefs' log based write barrier. The log of written cards is sometimes known as the dirty card queue. There are two major modifications; the use of a card table and a *hot card cache*.

After a pointer has been updated, the corresponding card is inspected in the card table. If it is already dirty, nothing needs to be done. This means that the card has been written to but not yet inspected by a background thread.

If the card is clean, it is dirtied, and inserted into the hot card cache. The hot card cache is a global data structure that caches the most recently written cards. When the hot card cache becomes full, it evicts one of the cards. This card is then appended to the log as described in Section 2.2. The purpose of delaying the log appending operation is to avoid duplications in the queue, and reduce contention when appending to the queue.

2.5 HotSpot G1 Refinement

The G1 implementation uses one or more concurrent background threads that do remembered sets insertions. The background threads look through the dirty card queue and *refine* the cards. Before a refinement, the card is marked as clean. A refinement then proceeds as follows: the background thread iterates over the objects located within the card, and if an object contains a reference that point into another region, which is not in the collection set, the card is inserted into the appropriate remembered set.

If a write is made into the card while it is being refined, the write barrier will see it as clean. The card will then be inserted into the hot card cache or queue as described above, and be scanned again later.

Since the refinement threads work concurrently with the mutators, the order of these operations is important. Consider if a refinement thread were to clean the card after processing it. Let the card have references $r_0, r_1, ..., r_n$, and the refinement thread iterate in ascending order. Say the first k references have been processed.

Then a mutator stores an interregional pointer at r_j where $j \leq k$ and dirties the card. The refinement thread then completes the iteration and cleans the card, never having looked at the new value of r_j. By having the cleaning of the card precede the refining, there is a guarantee that every pointer store will be followed by a refinement of that card.

2.6 The HotSpot G1 remembered set implementation

The current HotSpot G1 remembered set implementation uses many of the concepts described in the previous sections. The high level description is that the representation uses two different precision levels depending on the saturation; card level and region level precision.

2.6.1 Set representation

The hot card cache, the card table, and the dirty card queue are used when dealing with the remembered sets, but they are not actually part of the remembered set representation. These data structures are global, and not associated with a particular region. This section describes the representation of a single remembered set, which is associated with an owning region. A simplified C++ class definition is given in Figure 2.2 for illustrative purposes.

```
struct g1_rset {
  hash_map<region_id, card_list> sparse;
  hash_map<region_id, bitmap< MAX_CARD > > fine_grained;
  bitmap< MAX_REGION > coarse;
  // ... functions omitted
};
```

Figure 2.2. Simplified data structure definition for the G1 remembered sets.

There are two different levels of precision, and three different representations in the HotSpot G1 remembered sets. The two levels of precision are card level and region level precision. When enough cards in a given region are inserted into the remembered set, the entire region will be considered part of the remembered set. This is known as a *coarsening*, and is a transition from high to low precision representation.

The card level entries use two different representations, a *sparse table* and a bitmap. The bitmap is known as the *fine grained* representation. The sparse table is a hash map that maps a region to a short list of cards in that region. The fine grained data structure is a hash map that maps a region to a bitmap containing as many bits as there are cards in a region. Note that for both hash maps, the key is the originating region (i.e. where the pointer variable is stored), and not the owning region. Both hash maps are implemented as arrays of linked lists.

The region level entries (also known as the *coarse entries*) are kept in another bitmap. This bitmap has one bit for every region in the heap. The bit is simply set when the region is coarsened.

2.6.2 Remembered set memory management

The remembered sets themselves obviously need to be stored in memory. The memory used by the internal data structures has to be managed manually. They are managed in different ways:

- The coarse bitmap is simply allocated in its entirety when the remembered set is instantiated.
- The fine grained hash maps use a global free list for allocating the bitmap entries. This is possible because the bitmaps are all of the same size. If the free list is empty, a bitmap is allocated on the heap using the C++ `new` operator. When a remembered set is cleared, the fine grained entries are appended to the global free list. Appending to the global free list can be done from multiple threads. To speed this up, the bitmaps are linked together to form a doubly linked list. The freeing can thus be done using a single CAS instruction. This structure is depicted in Figure 2.3.
- A sparse table is a typical hash map consisting of an array of linked lists. The nodes in the linked lists are allocated in a continuous chunk using `new`, and are maintained in a free list that is local to the particular sparse table. When the chunk is filled, as indicated by an empty free list, the hash set is expanded to double the size. Note that this means that each node in the sparse table is a list of exactly the same size, and a single list cannot be expanded.

  Each list consists of a card array and a region identifier. The array is the list of card indexes included in the remembered set. The layout of some entries in the sparse table is illustrated in Figure 2.4.

2.6.3 Insertion operations

The insertion operations are somewhat complicated due to the complex representation. It is insertion operations that trigger shifting representations. The insertion algorithm is briefly summarized as Algorithm 1. The insertion algorithm first checks the coarse bitmap for the card's region, to see if the card to be added is already represented there. If so, nothing needs to be done, and the method returns.

Then, the fine grained hash map is queried to see if there is a card-level bitmap for the region. If it exists, the card's bit is set to 1 in the bitmap, and the method returns.

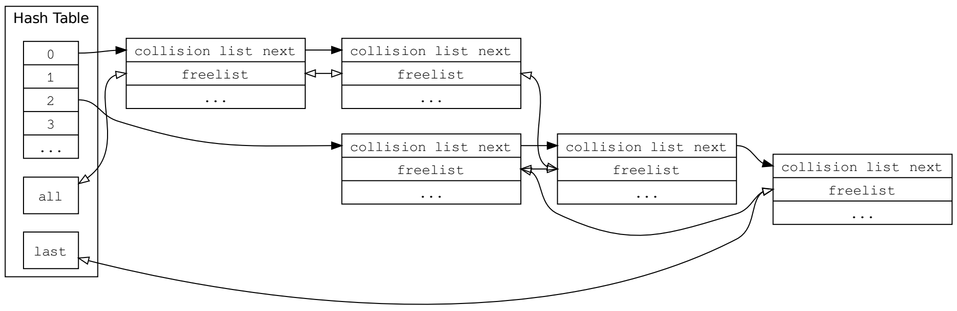

Figure 2.3. Schematic view of an example of a fine grained hash set. This figure only show how entries are linked together. The data contained in each entry are omitted.

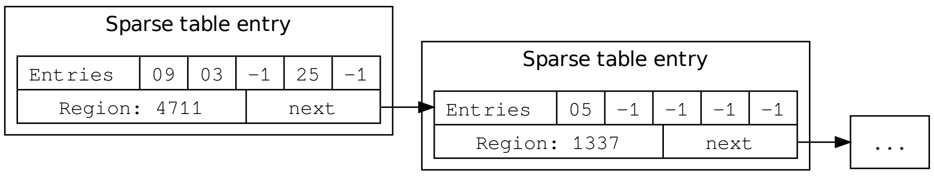

Figure 2.4. Illustration of how two entries in the sparse table could look like. The entries array can be much longer than illustrated here, but it is possible it is as short as 4 cards.

If there is no entry in the fine grained table, an attempt is made to insert the card into the sparse table. If the attempt succeeds, everything is well, and the method returns. However, when this list becomes full, the attempt fails. When this happens the list is converted to a bitmap and inserted into the fine grained hash map.

The fine grained hash map has a limit on its number of entries. If a sparse list is filled and the fine grained hash map is full, an entry from the fine grained hash map is evicted. The region represented by the evicted entry from the fine grained hash map is then coarsened. The policy for picking the entry to evict is to look at some subset of the fine grained bitmaps pick the one with the most bits set. After an entry is evicted, the filled sparse table list can be converted to a bitmap and inserted in the fine grained hash set. The conversion is done by clearing and reusing the evicted bitmap.

Algorithm 1 Remembered set insertion [1]

```
function RSInsert($r$, $c$)
    if Coarsened($r$) then
        return
    end if
    $f \leftarrow$ FindFineGrainedMap($r$)
    if $f$ = NULL then
        $m \leftarrow$ScopedMutex
        $f \leftarrow$FindFineGrainedMap($r$)
        if $f$ = NULL then
            if AddToSparseHashMap($r$, $c$) then
                return
            end if
            if $entries_{fine} = max_{fine}$ then
                $f \leftarrow$ EvictFineGrainedMap
            else
                $f \leftarrow$ AllocateFineGrainedMap
            end if
            AddToFineGrainedHashMap($f$, $r$)
            $entries_{fine} \leftarrow entries_{fine} + 1$
            $s \leftarrow$ GetSparseEntry($r$)
            for $i = 0$ to $i$ = Length($s$) $- 1$ do
                $c \leftarrow$ GetCard($s, i$)
                if $c \neq -1$ then
                    SetBit($f$, $c$)
                end if
            end for
            DeleteSparseList($r$)
        end if
    end if
    SetBit($f$, $c$)
end function
```

2.6.4 Insertion concurrency

The remembered set insertions can be made parallel, and to get acceptable throughput, synchronization operations should be kept to a minimum. Each remembered set has an associated mutex lock that is used for the operations that must be synchronized. The mutex lock is used when dealing with the sparse table and when modifying the fine grained hash set.

There are essentially two paths that are lock-free; when a card's region is already coarsened and when adding a card to a fine grained entry. When a card's region is coarsened, the bit is inspected in the bitmap, found to be set, and the method returns. This is safe to do, because there is no way to clear a coarsened bit.

Consider now a thread A about to add a card, whose region is represented in the fine grained set. First the bitmap is found, and then the bit is flipped. Suppose A is suspended after finding the bitmap, and another thread B evicts A's entry and reuses the bitmap. When A is then allowed to continue, the bit that A flips is for a card in a different region. However, because B evicted A's entry, the region containing A's card is coarsened, and thus A's insertion is represented.

The parallel insertions thus result in a small over-approximation of the remembered set, but this is an efficiency, and not a safety concern.

Chapter 3

Improvement suggestions

3.1 Problem statement

In what ways can the internal data structure used for HotSpot G1 remembered sets be modified to improve the garbage collector in terms of footprint and pause time? How is throughput affected by these modifications?

3.2 Reducing size of sparse lists

The sparse lists currently use 32 bit signed integers to represent the card indices in the short lists (see Figure 2.4). There is currently a cap on the region size at 32 MB, which means that the maximum number of cards in a single region is $\frac{32\text{MB}}{512\text{B}} = 2^{16}$. Using an unsigned 16 bit integer would be more appropriate.

However, this poses a problem. The current implementation uses -1 to represent null, or no value entries. If a 16 bit representation is to be used, each possible value is valid. This could be solved by using a separate bitmap for each sparse list, which keeps track of the null entries. This means that the cost of each entry would go from 32 to 17 bits per entry, a reduction of more than 46%.

When the sparse lists are short, this is not necessarily a save in either memory or time. The sparse lists are proportional to region size, and vary between 4 and 128 entries. To simplify the implementation, a 128 bit (16 byte) bitmap is added to each sparse list. This means that in the case where the lists are 4 entries long, the memory saved by using 16 bit integers is $4 \cdot 2$ bytes. The added bitmap is 16 bytes, which means a net growth of the sparse list of 8 bytes. In general, the footprint reduction with n entries per sparse list is $2n - 16$.

The length of the sparse lists is

$$n = 4 \cdot \frac{R}{2^{10}} \tag{3.1}$$

where R is the heap region size in bytes. The smallest heap size for which the sparse lists do not grow would be where $2n = 16$. Substituting $n = 8$ and solving

Equation 3.1 for R gives $R = 2^{11}$, or 2MB. Since G1 tries to have 2048 regions, 2MB heap regions means $2048 \cdot 2\text{MB} = 4\text{GB}$. Since the main design goal is to use G1 for large heap applications, this trade off may be suitable.

The smallest heap size for which the full footprint reduction can be utilized is trivially when the maximum region size is reached. This is when $R = 32 \cdot 2^{10}$, which translates to a heap of size 64GB.

Performance increases could be expected from better cache locality when entries are more tightly packed. Memory accesses should also be reduced since the null map can be fetched quickly and inspected in a register. This could be faster compared to doing memory reads for each entry.

Mutual exclusion is used when manipulating the sparse tables. This means there should not be a problem where different refinement threads invalidate each others cache lines by writing to distinct nearby locations (an effect known as *false sharing*). On the other hand the added overhead of a bitmap lookup per read is a possible source of performance degradation.

3.3 Using Bloom filters

A Bloom filter is a data structure that allows a set of elements to be over-approximated using a compact representation. For this reason it is interesting to consider as an implementation mechanism for remembered sets. Bloom filters support two operations, inserting and checking for membership. Insertion is illustrated in Algorithm 2, and membership checking is given in Algorithm 3.

Algorithm 2 Bloom filter insertion

```
function BloomFilterInsert($B$ , $\vec{d}$, $x$)
    for $h \in \vec{d}$ do
        $bit \leftarrow h(x) \mod length(B)$
        SetBit($B$, $bit$)
    end for
end function
```

Algorithm 3 Bloom filter member checking

```
function BloomFilterContains($B$, $\vec{d}$, $x$)
    for $h \in \vec{d}$ do
        $bit \leftarrow h(x) \mod length(B)$
        if GetBit($B$, $bit$) = False then
            return False
        end if
    end for
    return True
end function
```

The $\vec{d}$ used as a parameter for the Bloom filter is a set of hash functions. It is obvious that member checking can return false positives. For example, let $\vec{d}$ only contain a single hash function. Then any element $x \neq y$ whose hash value equals the hash value of an inserted element y would falsely be recognized as a member of the set. The probability of a false positive depends on the size of the bitmap, the number of hash functions, and the number of inserted elements.

Let ϕ represent the proportion of not yet set bits in a Bloom filter of size N, when d hash functions have been used for n insertions. Then ϕ can be approximated by the following formula:

$$\phi \approx (1 - d/N)^n \tag{3.2}$$

[6]. A false positive means that for each of the d hash values computed, the bit is found to be set. The probability of this is

$$P = (1 - \phi)^d \tag{3.3}$$

[6]. These equations can be used when sizing Bloom filters and deciding on number of hash functions.

3.3.1 Where to use Bloom filters

Realizing that Bloom filters have some desirable properties does not directly lead to where and how to use them in the context of G1 remembered sets. What is implemented as a part of this master thesis project is to insert another level of precision in between the fine grained bitmaps and the coarse representation. At this level, it is possible to create a data structure that uses Bloom filters which is significantly smaller than a bitmap, but allows faster scanning than the complete scanning used for coarsened regions.

It is particularly useful if the application behaviour leads to a situation where fine grained bitmaps are often evicted with few bits set. This happens when a region has a medium sized number of pointers from many of the other regions. Medium sized in this context means enough to overflow the sparse lists, but not enough to set a high percentage of the bits in the fine grained map.

3.3.2 Only asking the relevant questions

At evacuation time, the remembered set has to be iterated. The naive approach to iterating over a Bloom filter is to execute Algorithm 3 for each possible input. In the context of remembered sets that would mean to query the Bloom filter for each card in a from-region. This may be a possible solution, but it is desirable to limit the number of queries to the Bloom filter, especially if few hits are expected.

The implemented solution is to keep track of what prefixes of card numbers have been inserted, and only for the prefixes used, query the Bloom filter for each postfix. The contained prefixes are organized in a binary tree, making it quick to iterate over, and especially quick to skip large areas of a given region. The combined data structure using a Bloom filter and a binary tree for prefixes has been dubbed a “compact card set”, to emphasise the primary concern of being very memory efficient.

The prefix trees can be stored in a very compact way, as illustrated in Figure 3.1. Note the numbering of the nodes. By numbering the nodes in this way, the left child of a node n is always $2n + 1$, and the right child is $2n + 2$. Having this invariant means that each node can be represented with only two bits. The two bits in a node would represent whether or not there exists at least one leaf node in the left or right branch, respectively. The tree in Figure 3.1 would be represented using $2 \cdot 7 = 14$ bits. In general, with depth d, the binary tree would use $2^{d+1} - 2$ bits.

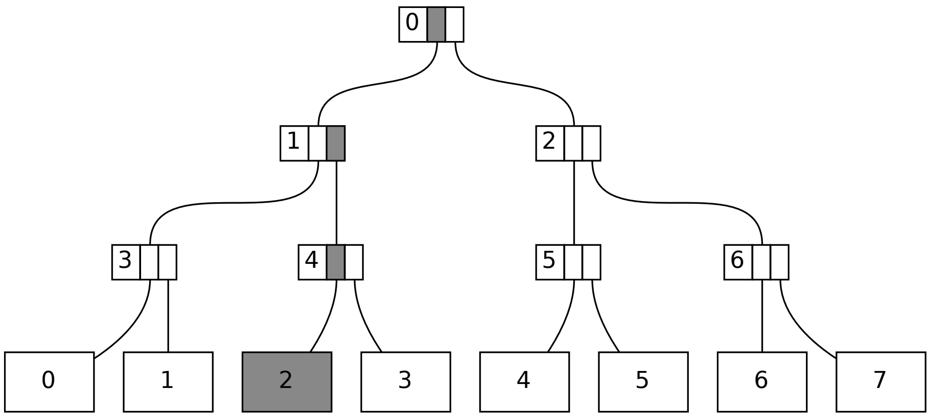

Figure 3.1. Conceptual representation of a prefix binary tree of depth 3. The empty cells are those that are actually stored in memory. The example has the value 2 inserted, and the marked cells correspond to set bits.

3.3.3 Compact card set sizing

The compact card set data structure needs to find a middle ground between the memory usage of the fine grained bitmaps and the coarse bitmap. It also needs to reduce the amount of cards scanned significantly. If the Bloom filter is sized too small, it will be little more than a costly indirection to a coarse region.

One simple aim is to be significantly smaller than the bitmap, and to need to check around half the cards. The bitmap could be as small as 2048 bits in the smallest case, since 1MB = 2048 cards $\cdot$ 512 bytes/card. So the combined size of the Bloom filter and the prefix tree needs to fall below this limit. Some additional bytes are also needed for pointers related to storing each compact card set in a hash map.

The implemented solution, in fact uses 64 bytes, accounted for in Table 3.1. The prefix tree is 16 bytes, which is 128 bits. Recall the formula for binary tree size of Section 3.3.2. With a depth of 6, the binary tree needs 126 bits, which leaves two bits of waste. In other words, each from-region is split into $2^6 = 64$ equally sized areas identified by the 6 high bits of the card indices.

The Bloom filter uses 32 bytes (256 bits). Picking a power of two has the advantage of making the modulo operations in Algorithms 2 and 3 a bitwise logical `AND`, which is a low cost operation.

Table 3.1. Memory sizes chosen for the compact card set.

Item	Bytes used
Prefix tree	16
Bloom filter	32
Pointers	16

3.3.4 Managing false positives

As stated above, Bloom filters have false positives. The simple aim of attempting to be at least twice as fast as linearly scanning a region would allow for a false positive rate of up to 50%.

With the sizing developed above, $N = 256$ and $P = \frac{1}{2}$. The remaining variables are n, the number of insertions to support and d, the number of hash functions. Substituting Equation 3.2 into Equation 3.3 gives

$$P = \left(1 - \left(1 - \frac{d}{256}\right)^n\right)^d \tag{3.4}$$

.

Figure 3.2 shows the analytical expected false positive rate and measured performance of the compact card set. As can be seen from the graph, a hit rate of about 50% is at around 150 insertions. This is used as a decision point for when to use the compact card set, and when to immediately coarsen a region. The region size used for the measurements is 16 MB.

From Equation 3.4, and more clearly from Figure 3.2 it can be seen that 2 hash functions initially has more false positives than using 3, but does not grow quite as quickly. This can be understood intuitively by realising that many insertions using more hash functions sooner lead to a high percentage of the bits being set, which leads to more false positives. On the other hand, when few bits are set, having a higher number of hash functions in Algorithm 3 reduces the risk of collisions.

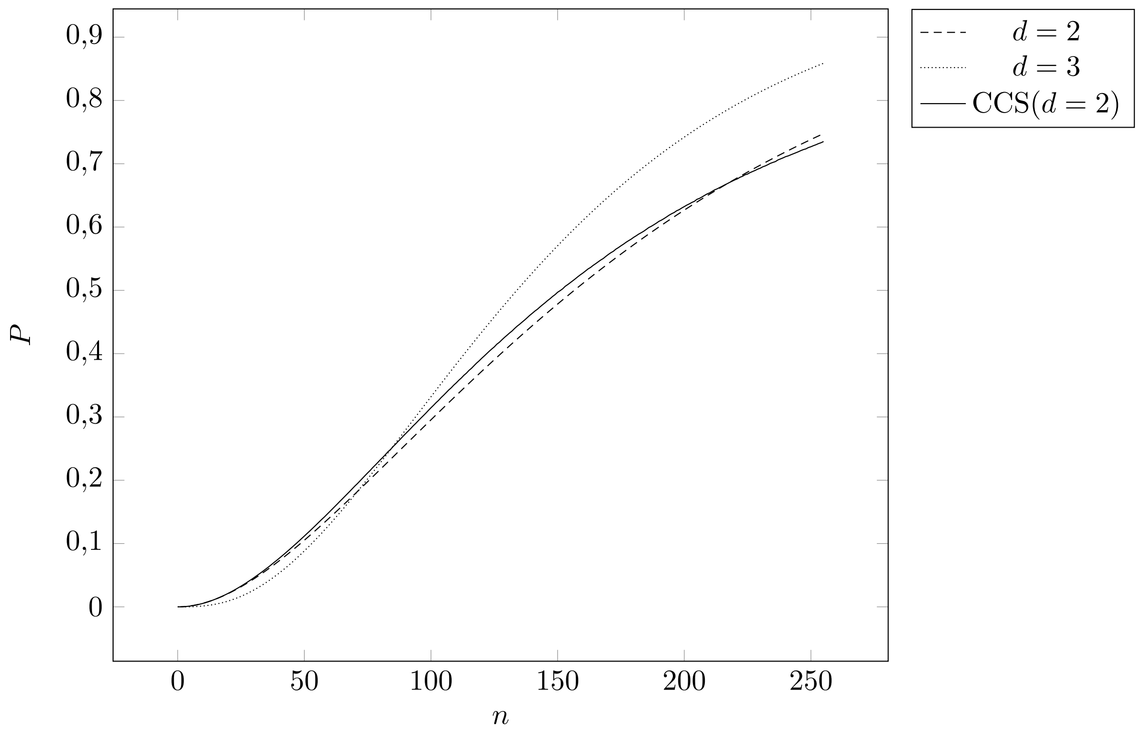

Figure 3.2. False positive analytical predictions and measurements using 256 bits Bloom filter for n insertions. The number of hash functions is denoted by d. CCS is the implementation using the prefix trees.

Chapter 4

Evaluation methodology

Evaluating a garbage collector is a difficult problem. This is partly because the garbage collector is never executed in isolation from the rest of the runtime. Another fact that makes it hard to give a conclusive judgement on the quality of a GC is the fact that GCs involve trade offs such as time versus memory usage. Depending on the application one or the other could be more important. Since G1 is aimed at large heaps and low pause time goals, the memory footprint and pause times are measured. A baseline is established using the current implementation with added instrumentation.

4.1 Measurement methods

To do memory usage measurements, the Java Flight Recorder infrastructure available in HotSpot is used. The Flight Recorder infrastructure allows recording of events. Events can have data associated with them, and timing information recorded. An event can either be instantaneous or have a duration. The Java Mission Control application can then be used to analyze a recording. The infrastructure also allows events to easily be turned on or off or filtered, so as to only record the relevant events.

The compact card sets in Section 3.3.1 are benchmarked separately. Code for these benchmarks can be found in Appendix A.

4.2 System setup

All benchmarks are executed on the same system, as specified in Table 4.1. This is done in order to eliminate all external factors affecting performance such as compiler, operating system version, system library and memory configuration.

Table 4.1. Benchmark server specifications.

CPU Model	Intel(R) Xeon(R) CPU X5670
Number of Cores	12
Number of HW Threads	24
RAM	94,6 GB
Operating System	Oracle Linux Server release 5.8

4.3 Benchmarks used

Two relatively simple benchmarks are used for showing that modifications have the desired effects. In addition to these two very similar benchmarks, the standard SPECjbb2013 benchmark is run to detect any throughput regressions.

The DaCapo benchmark suite [5] could also considered as a candidate for measurements, but none of the benchmarks in the suite have working sets large enough. When trying to measure the performance of interregional pointer updates and especially mixed collections, large working sets (at least 1500MB) are necessary.

4.3.1 The LRU cache benchmark

This benchmark exercises a sort of worst case performance for G1. A `LinkedHashMap` is used to implement a least recently used (LRU) cache. It works by randomly generating an integer key and checking if the cache has the object. If the cache is found to not have the object, it is created and inserted, otherwise it is fetched. The space of possible keys is twice as big as the size of the cache. This means that once the cache is filled, about half the requests are `insert` and half are `get` calls. A modified version of this benchmark adds a medium sized byte array to each object. This modification is done to model a simplified version of having each object in the cache represent a page in a templating system.

A simple templating system could generate some large array (like a `String` of HTML code) based on a database object. In order to save on processing and database access, the generated string could be stored in a LRU cache together with the object it represents. The modified version does not exercise the worst case performance for G1 in the same way as the original version, because there are much fewer pointers in each region. Most of the data in the heap consist of the arrays, but there are still lots of interregional pointers, making the remembered set footprint, update times and scan times interesting.

There are several reasons these benchmarks are troublesome for G1. First, the generational hypothesis does not hold. The objects that are evicted from the cache are the *least* recently used, which means that objects becoming garbage typically have been promoted to the old generation. Second, the linked hash map modifies the map when an object is accessed. This means lots of pointer updates that span memory regions in a pseudo-random way. A large set of objects with random pointers between them is troublesome for G1, since the pointers into a given region will be spread out across most of the heap.

4.3.2 SPECjbb2013

SPECjbb2013 is a standardized benchmark used to measure performance of Java systems. It is used both as a hardware benchmark and JVM implementation benchmark.

For this thesis, the benchmark is executed in the composite mode. The composite mode is the simplest way to execute the benchmark, running all parts in a single JVM instance.

SPECjbb2013 computes two performance metrics known as max-jOPS and critical-jOPS [22]. The max-jOPS is a measurement of peak throughput and the critical-jOPS is a "response time constraint metric". The critical benchmark is of greater interest to G1 modifications, since one major objective of the G1 garbage collector is to put a soft limit on pause times. It should be noted that short pause times and critical throughput are the same thing. For example, a JVM implementation that pauses for 10 ms 50 times per second will spend 50% of the time in a stop the world phase, and thus probably have very low throughput but also low pause times. As discussed previously, this is a typical trade off for GC implementations.

Chapter 5

Results

This chapter shows the results from the benchmarks presented in Section 4.3. Three implementations are evaluated, and each has been given a designated code. The codes are explained in Table 5.1.

Table 5.1. Explanation of algorithm codes used for presentation.

Code	Algorithm description
BL	Baseline — the current implementation in HotSpot with added instrumentation
CSL	Compact Sparse Lists — the implementation using unsigned 16 bit card indices as described in Section 3.2
CCS	Compact Card Set — the implementation using the compact card set as described in Section 3.3

5.1 Compact card set iteration speed

Figure 5.1 is a demonstration of the effect of using the binary tree together with a Bloom filter as described in Section 3.3.2 compared to a plain Bloom filter. The Bloom filters in the two implementations are of the same size. Note that the x-axis is a log-scale. The reason that the curve drops into negative values after a certain amount of inserts is that it becomes likely that most "sub-regions" maintained by the binary tree has at least one card added. In other words, the binary tree becomes saturated. Then the added work of iterating over the binary tree costs some time, but does not save any work.

Note that the time saving is eliminated after approximately 100 insertions. This is close to the number of insertions where the Bloom filter hit rate starts to approach 50% (as seen in Figure 3.2). This means that performance in both the iteration speed and hit rate starts to degrade around the same number of inserts. This means that the Bloom filter and binary tree become saturated at a similar pace.

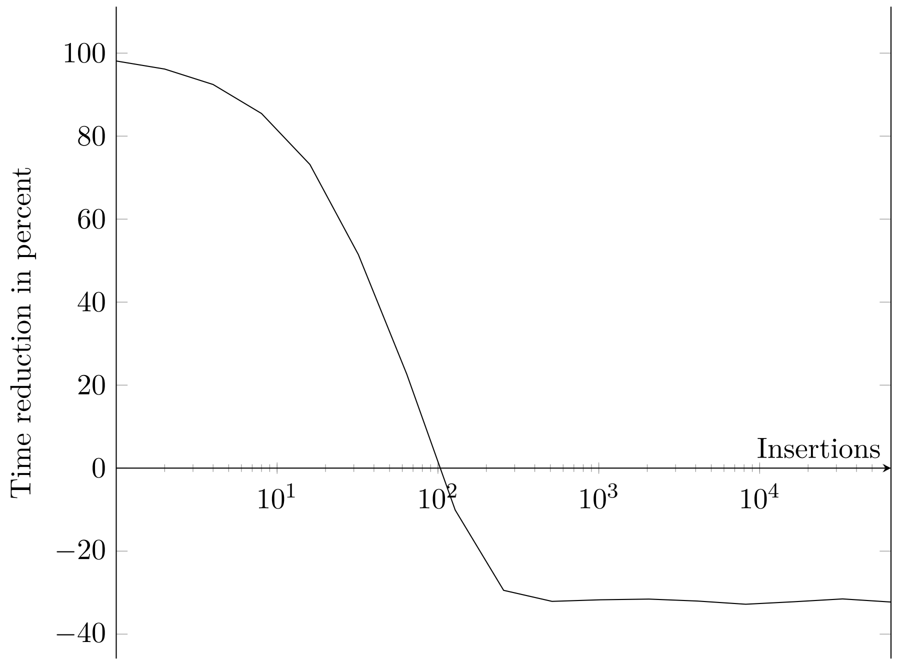

Figure 5.1. Time saved by using the compact card set as opposed to a plain Bloom filter of the same size.

5.2 The LRU cache benchmarks

The following two sections present and discuss relevant results from executing the LRU cache benchmark. The results are not presented in an entirely consistent way between the two benchmarks. This is in the interest of brevity and to not overwhelm the reader with irrelevant figures and graphs.

5.2.1 Worst case performance

The runtime parameters and benchmark properties are listed in Table 5.2. Figure 5.2 and Table 5.3 show the characteristics of the implementations in the worst case benchmark.

Table 5.2. Some important LRU cache benchmark parameters. Additional parameters for logging, JFR recording et cetera are omitted.

Total heap size	4GB
Young generation size	256MB
`duration`	1800
`templating`	`false`
`workingset`	5000000
`threads`	4

As can be seen in Figure 5.2, the pause times are significantly reduced by using the compact card sets. This is because, as expected, the compact card sets can intercept coarsening events that would otherwise cause entire regions to be scanned. The variation in the GC pauses is simply that the longer pauses are mixed GCs, while the young generation pauses actually do meet the pause time goal. Each mixed GC frees up a set of regions which is then used for the young generation allocations. This back-and-forth between the two causes the spikes in pause times seen in Figure 5.2.

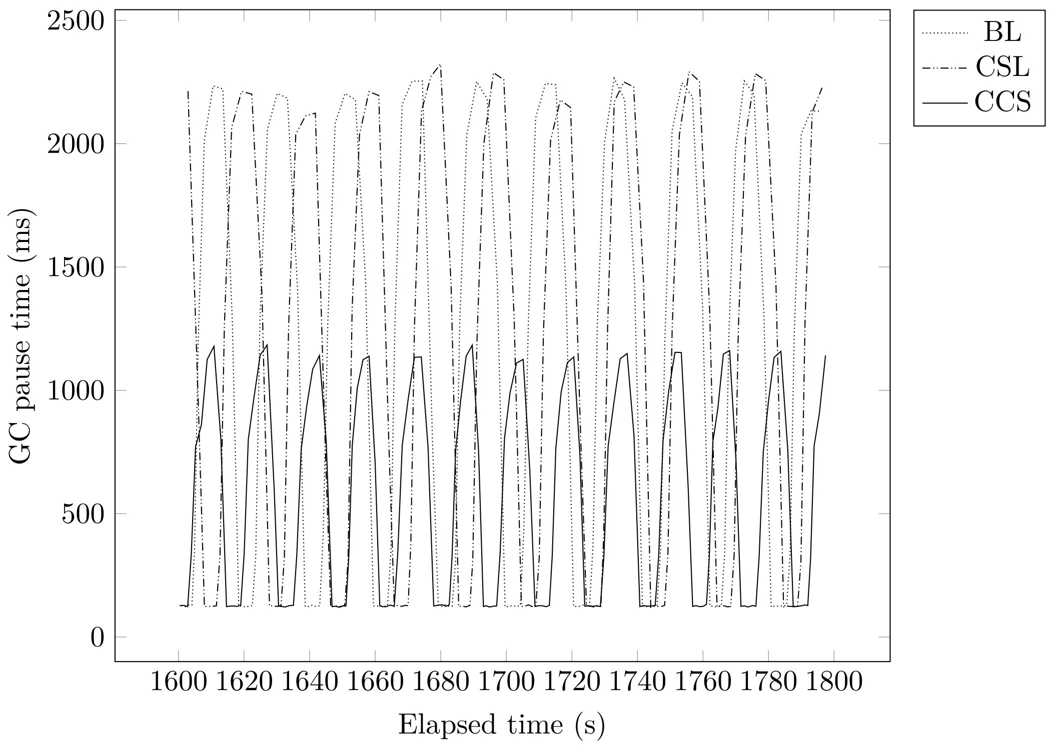

Figure 5.2. GC pause times. Each point on the lines represents a GC pause. The timeline has been cut in order to increase legibility in the graph.

Table 5.3 shows the remembered set footprint. As expected, the compact card set algorithm (CCS) uses some additional memory compared to the baseline implementation for the Bloom filters and prefix trees. The reason that the switch to use unsigned 16 bit integers is not reducing the footprint significantly in this benchmark is that almost all of the remembered set footprint is used by the fine grained bitmaps. This is not surprising, since the benchmark causes lots of small objects to contain references into pseudo-random places in the heap, which means that each region has a nontrivial amount of references from each other region. This situation means that the sparse lists quickly overflow, causing the fine grained maps to be used extensively.

Algorithm	Average footprint (bytes)
BL	794.1MB ± 5.38 %
CSL	759.7MB ± 5.46 %
CCS	845.7MB ± 4.58 %

Table 5.3. Memory footprint used for remembered sets.

Figure 5.3 shows that the remembered set scan times are significantly reduced by using the compact card sets, compared to both the BL and CSL implementations. This is precisely the purpose of the CCS implementation. Another thing worth noting from Figure 5.3 is that the scan times are not affeceted by using shorter sparse lists. This is expected, because this benchmark makes very little use of the sparse lists, and mainly uses the fine grained bitmaps.

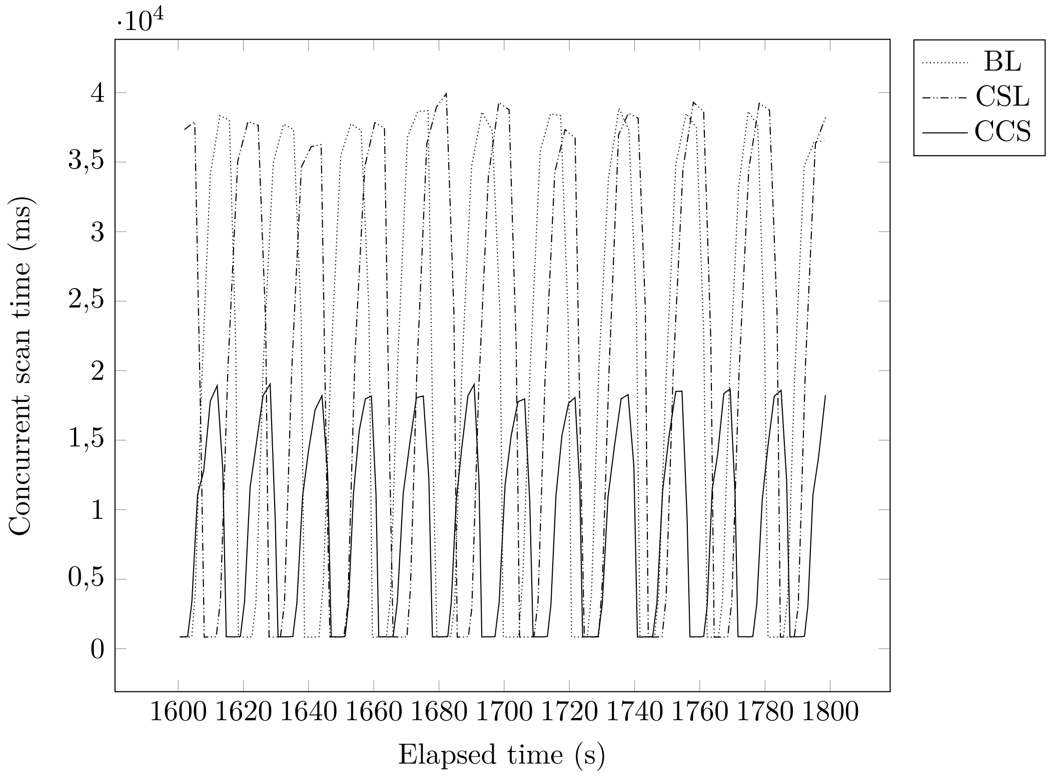

Figure 5.3. Remembered set scan times in the LRU cache benchmark. Note that the time is the sum of total time spent by multiple threads working concurrently. This explains why the scan time can exceed the total pause time.

Table 5.4 shows a crude estimate of the number of request that would time out in the application. A weakness with this measurement is the fact that the VM is generating requests itself. This means requests cannot be generated in a STW phase, which is not a realistic situation.

Table 5.4. LRU cache benchmark timeout statistics.

Algorithm	Total timeouts	Maximum response time (ms)
BL	2192	2325
CSL	2179	2379
CCS	2786	1210

The CCS algorithm has more requests that fail to complete on time. However, the maximum response time is almost reduced by half compared to the BL and CSL algorithms. Lower response times are expected, given the lowered GC pause times. However, the reason that the total number of timeouts is increased is not as evident. It most likely has to do with the fact that the CCS algorithm causes more, but shorter, garbage collection pauses than in the BL algorithm.

5.2.2 Template approximation performance

These tables show the measurements on the LRU cache benchmarks where each object also has a reference to their own `String` generated using a mix of statically computed text and some dynamically computed text depending on the object key. The source code for this can be found in Appendix B.3. The runtime configuration, and benchmark parameters are listed in Table 5.5.

This benchmark exposes the footprint reduction caused by having smaller sparse table entries. Recall from Section 3.2 that the smallest heap size to get the full benefit of the smaller integers is 64 GB.

Table 5.5. Some important LRU templates benchmark parameters. Additional parameters for logging et cetera are omitted.

Total heap size	64GB
Young generation size	256MB
`duration`	1800
`templating`	`true`
`workingset`	5000000
`threads`	4

Table 5.6. Average mixed pause times for different algorithm implementations.

Algorithm	Mixed pause time (ms)
BL	$263 \pm 27\%$
CSL	$258 \pm 27\%$
CCS	$287 \pm 27\%$

In this scenario, the algorithms perform significantly better in terms of meeting the pause time goal. As can be seen in Table 5.6, the differences are relatively small, with a small regression for CCS. In this benchmark, the sparse lists are exercised heavily. Because the CCS algorithm has a more complicated code path for insertions, reaching the sparse list insertion path takes slightly longer compared to the other two.

Table 5.7 shows that the remembered set footprint is significantly reduced by switching to unsigned 16 bit integers (CSL). This is the desired effect, and the heap size is chosen to be large enough to achieve this effect. The reason that the memory reduction is not quite 46%, as is claimed in Section 3.2, is that the sparse lists are

Table 5.7. Memory footprint for remembered sets.

Algorithm	Average footprint (bytes)
BL	926.0MB ± 3.59 %
CSL	732.6MB ± 3.87 %
CCS	931.6MB ± 3.35 %

organised in hash sets. The hash sets have some footprint for the bucket arrays, and the pointers between list nodes.

5.3 SPECjbb2013

Table 5.9 lists the runtime parameters used when running the benchmark. The SPECjbb2013 results listed in Table 5.8 are established by running the benchmark 6 times and picking the best result for each algorithm. Both implementations exhibit throughput regressions. The regressions are both in terms of critical and maximal throughput.

Table 5.8. SPECjbb2013 performance results. The standard throughput measurement as defined by SPEC is known as jOPS [22].

Algorithm	max jOPS	critical jOPS
BL	19086	9381
CSL	18291	9331
CCS	18887	9158

In the case of the CSL algorithm, the performance regression probably means that the computational overhead of the bitmaps added to the sparse lists outweigh the cache locality benefints of compacting the sparse array. The sparse lists are the maximum length with this heap size, so the cache locality benefits should be maximal. A reasonable explanation could also be that the benchmark allocation behaviour does not lead to heavy usage of the sparse lists.

For the CCS algorithm, the throughput regressions are most likely explained by removing the possibility for a lock-free path to the fine grained bitmaps. Recall from Section 2.6.4 that only the representation which is last before coarsening works in a lock free way. When the compact card sets are not needed, they cause a performance regression for the applications that need only the fine grained or sparse list representations.

Table 5.9. SPECjbb2013 runtime parameters. Note that setting the initial and maximum heap and young gen sizes effectively locks them to the specified value.

Runtime parameter	Value	Explanation
-Xloggc	log.gc	Log garbage collection information to a file
-XX:+UseG1GC		Use the G1 garbage collector
-Xmx	64G	Maximum heap size
-Xms	64G	Initial heap size
-XX:NewSize	10G	Initial young generation size
-XX:MaxNewSize	10G	Maximum young generation size

Chapter 6

Conclusions

Two ways of improving the G1 remembered sets implementation are explored in this thesis. The compact card set implementation, using Bloom filters as an intermediate data structure between the fine grained hash tables and the coarse bitmap, is shown to be able to reduce pause times in situations where the current implementation fails to meet pause time goals. The other implementation, using 16 bit integers and a bitmap instead of 32 bit integers, is shown to be able to reduce memory footprint in applications with large heaps where the sparse lists are heavily used.

It is not part of this project to evaluate whether the two approaches together could provide some benefit compared to the current implementation. One reduction in memory footprint and one reduction in pause times, which causes added memory footprint, could in a sense "cancel out" the memory difference. This would then provide the lowered pause times and no added memory footprint. However, since provoking these effects require some special and distinct benchmarks to be made, it is not clear that there are applications where both effects occur. In fact, it is not very likely that there would be a large amount of sparse entries and a large amount of fine grained evictions in the same application. It could be possible to get the benefits of both these improvements simultaneously if the application exhibits both behaviours at the same time. This could happen if a JVM is running as an application host for many different applications sharing a big server.

6.1 Recommendations

It is not recommended to add the current implementation of compact card sets to the HotSpot code base. The reason for this is that the added footprint and reduced computational throughput is most likely not acceptable for most applications. It seems though, as if the approach of having a more smooth precision versus footprint trade off is promising in terms of lowering pause times.

Since G1 is aimed at applications with large heaps, the switch to using 16 bit integers in the sparse lists could be worth integrating into the code base. For heaps larger than 4GB the remembered set footprint can be significantly reduced.

6.2 Suggestions for further research

The sizes picked for the prefix trees and Bloom filters in the compact card set implementation are rather arbitrary. An interesting study would be to try dynamically sizing the data structures to fit the application's allocation behaviour, or to derive the sizes during system startup from the heap size.

In this thesis project, modifications to the internal structure of the remembered sets have been evaluated. It would be interesting to look at reducing the total number of remembered sets as well. Regions that are bound to be collected at the same time could in principle share a remembered set. One obvious such set of regions is the young generation.

Another interesting idea is to try to reduce the number of empty remembered sets. These include remembered sets for not yet used regions. Another category of remembered sets that fall into this category is remembered sets which belong to a continued humongous region. It is possible to allocate very large objects (humongous is the term used in HotSpot) that needs to span several regions. Then the remembered set for every region except the first and last will necessarily be completely empty, because in Java there cannot be references to object fields. Eliminating these empty remembered sets completely, or reducing their memory footprint could be an optimization worth evaluating.

Bibliography

[1] Other regions table :: add_reference. http://hg.openjdk.java.net/jdk9/hs-gc/hotspot/file/1fd17e0a0791/src/share/vm/gc_implementation/g1/heapRegionRemSet.cpp. Accessed 2014-04-29.

[2] Postgresql 9.3.3 documentation: Routine vacuuming. http://www.postgresql.org/docs/9.3/static/routine-vacuuming.html. Accessed 2014-02-25.

[3] Sqlite query language: Vacuum. http://www.sqlite.org/lang_vacuum.html. Accessed 2014-02-25.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java Second Edition*. Cambridge University Press, 2004. ISBN 0-511-04286-8.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[8] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments - VEE '05*, page 46, New York, New York, USA, 2005. ACM Press.

[9] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on*

Memory Management, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[10] David Detlefs, Ross Knippel, William D. Clinger, and Matthias Jacob. Concurrent remembered set refinement in generational garbage collection. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, 2002.

[11] Edsger W. Dijkstra, Leslie Lamport, a. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[12] AL Hosking, JEB Moss, and D Stefanovic. A comparative performance evaluation of write barrier implementation. *ACM Sigplan Notices*, pages 92–109, 1992.

[13] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *IN OOPSLA'93 WORKSHOP ON GARBAGE COLLECTION AND MEMORY MANAGEMENT*, 1993.

[14] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook - The Art of Automatic Memory Management.* CRC Press, 2012. ISBN 978-1-4200-8279-1.

[15] H Lieberman and C Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 1983.

[16] Sun Microsystems. The java hotspot virtual machine. Technical report, Sun Microsystems, Inc, 2001.

[17] Sun Microsystems. Memory Management in the Java HotSpot ™ Virtual Machine. Technical Report April, 2006.

[18] David A Moon. Garbage collection in a large LISP system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*, pages 235–246, New York, New York, USA, 1984. ACM Press.

[19] Oracle. *Java SE Documentation - Garbage-First Collector*, 7th edition, 2014.

[20] PP Pirinen. Barrier techniques for incremental tracing. *ACM SIGPLAN Notices*, pages 20–25, 1998.

[21] PG Sobalvarro. *A lifetime-based garbage collector for Lisp systems on general-purpose computers.* Bachelor thesis, Massachusetts Institute of Technology, 1988.

[22] Standard Performance Evaluation Corporation (SPEC). SPECjbb2013 Design Document. `http://www.spec.org/jbb2013/docs/designdocument.pdf`. Accessed 2014-05-19.

[23] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR. *ACM SIGARCH Computer Architecture News*, 12(3):188–197, June 1984.

[24] Paul R Wilson and Thomas G Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.

Appendix A

Compact card set benchmarks

Note that the compact card set implementation is not included.

```
#include <cstdio>
#include <iostream>
#include <cstdlib>
#include <thread>
#include <vector>
#include <sys/time.h>
#include <sys/resource.h>

#include "compactCardMap.hpp"

template<typename Container>
void bench_insertions(const unsigned N,
  const unsigned n_threads,
  const unsigned insertions,
  const unsigned n_cards) {
  Container map;
  Container * mapptr = &map;
  CompactCardMap::set_cards_per_region(n_cards);


  for (unsigned i = 0; i < N; ++i) {
    std::vector<std::thread>  threads;
    map.clear();
    for (unsigned t = 0; t < n_threads; ++t) {
      threads.push_back(std::thread([=]() {
        for (unsigned c = 0; c < insertions; ++c) {
          size_t card = random() % n_cards;
          mapptr->insert(card);
        }
      }));
    }

    for (auto & t : threads) {
      t.join();
    }
  }
};


volatile size_t global_var = 3047;
template<typename T>
void eat(T val) {
  global_var ^= val;
};

size_t time_between(struct rusage & t0, struct rusage & t1) {
  long long sec_diff = t1.ru_utime.tv_sec - t0.ru_utime.tv_sec;
  long long usec_diff = t1.ru_utime.tv_usec - t0.ru_utime.tv_usec;
  long long result = sec_diff * 1000000 + usec_diff;
  return (size_t)result;
};

template <typename Container>
double bench_iterations(const unsigned inserts,
    const unsigned iterations,
    const unsigned n_cards) {
  CompactCardMap::set_cards_per_region(n_cards);
  srandom(4711);
  size_t total_microseconds = 0;
  for (unsigned i = 0; i < iterations; ++i) {
    Container bag;
    for (unsigned j = 0; j < inserts; ++j) {
      size_t val = random() % n_cards;
      bag.insert(val);
    }
    struct rusage ru_before, ru_after;
    getrusage(RUSAGE_SELF, &ru_before);
    for (auto it = bag.begin(); !it.done(); ++it) {
      eat(*it);
    }
    getrusage(RUSAGE_SELF, &ru_after);
    total_microseconds += time_between(ru_before, ru_after);
  }
  double average_time = double(total_microseconds) / iterations;
  printf("Used %lu microseconds for %u iterations (avg %lf microseconds/iteration)\n", total_microseconds, iterations,
    average_time);
  return average_time;
}

int main() {
  unsigned insertions;
  std::cin >> insertions;
  printf("========== Benchmarks for %u insertions ==========\n",
     insertions);
  double avg_compact = bench_iterations<CompactCardMap>(
    insertions, 4096 /* iterations*/, 1 << 16 /* cards*/);
  double avg_bloom = bench_iterations<BloomFilter>(
    insertions, 4096 /* iterations*/, 1 << 16 /* cards*/);
  double savings_in_percent =
    100 * (avg_bloom - avg_compact) / avg_bloom;
  printf("Compact is ~%lf %% faster\n", savings_in_percent);
  printf("%lf\t%lf\n", avg_bloom, avg_compact);
  return 0;
}
```

Appendix B

LRU cache benchmark

B.1 BenchMarkResults.java

```
package ansjob.benchmarks.lrubench.domain;

import ansjob.benchmarks.lrubench.domain.Utils.Pair;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.LongSummaryStatistics;

class BenchMarkResults {

  long totalIterations;
  long totalCacheMisses;
  LinkedHashMap<Integer, Pair<DomainItem, String>> cache;
  List<Long> missedDeadlines = new ArrayList<>();
  LongSummaryStatistics deadLineStats;

}
```

B.2 DomainItem.java

```
package ansjob.benchmarks.lrubench.domain;
import static ansjob.benchmarks.lrubench.domain.Utils.
    randomString;

import java.util.Random;

class DomainItem {

  public int id;
  public int likes;
  public String name;
  public String[] imageUrls;

  DomainItem(int id) {
    this.id = id;
    Random r = new Random(id);
    likes = r.nextInt(5000000);
    int maxNameLength = 32;
    int maxImages = 8;
    int maxUrlLength = 32;
    name = randomString(r, r.nextInt(maxNameLength));
    imageUrls = new String[r.nextInt(maxImages)];
    for (int i = 0; i < imageUrls.length; ++i) {
      imageUrls[i] = randomString(r,
          r.nextInt(maxUrlLength));
    }
  }

  @Override
  public int hashCode() {
    return id;
  }

  @Override
  public boolean equals(Object o) {
    DomainItem x = (DomainItem) o;
    return x.id == id;
  }

}
```

B.3 RequestThread.java

```
package ansjob.benchmarks.lrubench.domain;

import java.util.*;
import static ansjob.benchmarks.lrubench.domain.Utils.*;

class RequestThread extends Thread {

  private final long endMillis;
  private long iterations;
  private long cacheMisses;
  private final int maxId;
  private final Random rand;
  private final boolean templatingMode;
  private final long goalMillisPerRequest = 300;
  private final ArrayList<Long> missedDeadlines;
  private long trashVariable;

  private final LinkedHashMap<
      Integer,
      Pair<DomainItem, String>> cache;

  public RequestThread(Random r,
      boolean templatingMode,
      int maxId,
      LinkedHashMap<Integer,
          Pair<DomainItem, String>> cache,
      long endMillis) {
    this.missedDeadlines = new ArrayList<>();
    this.cache = cache;
    this.templatingMode = templatingMode;
    this.endMillis = endMillis;
    this.maxId = maxId;
    this.rand = r;
  }

  @Override
  public void run() {
    while (true) {
      long startTime = System.currentTimeMillis();
      if (startTime > endMillis) {
        return;
      }
      iterations++;
      Pair<DomainItem, String> entry = null;
      int nextId = rand.nextInt(maxId);
      synchronized (cache) {
        if (cache.containsKey(nextId)) {
          entry = cache.get(nextId);
        }
      }
      if (entry == null) {
        cacheMisses++;
        DomainItem item = new DomainItem(nextId);
        String textRepresentation;
        if (templatingMode) {
          textRepresentation = getHtml(item);
        } else {
          textRepresentation = null;
        }

        entry = new Pair<>(item, textRepresentation);
        synchronized (cache) {
          cache.put(nextId, entry);
        }
      }
      long endTime = System.currentTimeMillis();
      long requestTime = endTime - startTime;
      if (requestTime > goalMillisPerRequest) {
        missedDeadlines.add(requestTime);
      }
      consume(entry);
    }
  }

  public long getIterations() {
    return iterations;
  }

  public long getCacheMisses() {
    return cacheMisses;
  }

  private void consume(Pair<DomainItem, String> cachedEntry)
      {
    trashVariable += cachedEntry.first.id;
    }

    private static final String header =
        randomString(new Random(0), 1000);
    private static final String footer =
        randomString(new Random(-1), 200);

    private static final String[] intermediateHtmlChunks = {
      randomString(new Random(1), 512),
      randomString(new Random(2), 512),
      randomString(new Random(3), 100),
      randomString(new Random(4), 50)
    };

    private String getHtml(DomainItem item) {
      StringBuilder sb = new StringBuilder(8 * 1024);
      sb.append(header);
      sb.append(item.name);
      sb.append(intermediateHtmlChunks[0]);
      sb.append(item.likes);
      sb.append(intermediateHtmlChunks[1]);
      if (item.imageUrls != null) {
        for (String url : item.imageUrls) {
          sb.append(intermediateHtmlChunks[2]);
          sb.append(url);
          sb.append(intermediateHtmlChunks[3]);
        }
      }
      sb.append(footer);
      return sb.toString();

    }

    public List<Long> getMissedDeadlines() {
      return missedDeadlines;
    }

    public long getTrashValue() {
      return trashVariable;
    }

}
```

B.4 Runner.java

```
package ansjob.benchmarks.lrubench.domain;

import ansjob.benchmarks.lrubench.domain.Utils.Pair;
import java.util.ArrayList;
import java.util.Date;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map.Entry;
import java.util.Random;
import org.apache.commons.cli.*;

public class Runner {

  public static void main(String[] args) throws
      InterruptedException {
    CommandLine cli = null;
    Options options = getOptions();
    try {
      CommandLineParser cliParser = new GnuParser();
      cli = cliParser.parse(options, args);
    } catch (ParseException ex) {
      printHelp(options);
      System.exit(-1);
    }

    if (cli.hasOption("help")) {
      printHelp(options);
      return;
    }

    System.out.printf(
        "Running␣ansjob␣LRU␣Templating␣Benchmark\n");

    int requestThreads = Integer.parseInt(
        cli.getOptionValue("threads", "4"));
    int cacheSize = Integer.parseInt(
        cli.getOptionValue("workingset", "50000000"));
    int timeLimit = Integer.parseInt(
        cli.getOptionValue("duration", "1800"));
    boolean useTemplates = Boolean.parseBoolean(
        cli.getOptionValue("templating", "false"));
```

```java
    Runner runner = new Runner(requestThreads,
        useTemplates, cacheSize, timeLimit);
    runner.printParameters();
    runner.runBenchmark();
    BenchMarkResults results = runner.getResults();
    printResults(results);
  }

  private static Options getOptions() {
    Options options = new Options();

    options.addOption("help", false,
        "prints help message");
    options.addOption("threads", true,
        "number of request-generating threads");
    options.addOption("workingset", true,
        "number of elements in the LRU cache");
    options.addOption("duration", true,
        "the time to execute the parallel threads"
            + " before stopping (in seconds)");
    options.addOption("templating", true,
        "wether to use or not to use the"
            + " templating mode (true/false). "
            + "Default value is false");

    return options;
  }

  private static void printHelp(Options options) {
    HelpFormatter printer = new HelpFormatter();
    printer.printHelp("lrubench", options);
  }

  private LinkedHashMap<Integer, Pair<DomainItem, String>> cache;
  private List<RequestThread> requestThreads = new ArrayList<>();
  private final long nThreads;
  private final long endMillis;
  private final int cacheSize;
  private final boolean useTemplates;

  private Runner(int requestThreads,
      boolean useTemplates,
      int cacheSize,
      int timeLimit) {
    this.useTemplates = useTemplates;
    cache = new LinkedHashMap<Integer,
        Pair<DomainItem, String>>
      (cacheSize, (float) 0.75, true /* access order */){

      boolean saturationReached = false;

      @Override
      protected boolean removeEldestEntry(
        Entry<Integer, Pair<DomainItem, String>> entry) {
        if (!saturationReached &&
            size() >= cacheSize) {
          System.out.printf(
              "[%s]␣Saturation␣reached!\n",
              new Date());
          saturationReached = true;
        }
        return size() > cacheSize;
      }
    };
    this.cacheSize = cacheSize;
    endMillis = System.currentTimeMillis() + timeLimit *
       1000;
    nThreads = requestThreads;
  }

  private void runBenchmark() throws InterruptedException {
    int maxId = this.cacheSize * 2;
    for (int i = 0; i < nThreads; ++i) {
      RequestThread t = new RequestThread(
          new Random(i), useTemplates, maxId,
          cache, endMillis);
      requestThreads.add(t);
      t.start();
    }

    for (Thread t : requestThreads) {
      t.join();
    }
  }

  private BenchMarkResults getResults() {
        BenchMarkResults res = new BenchMarkResults();
        for (RequestThread t : requestThreads) {
            res.totalCacheMisses += t.getCacheMisses();
            res.totalIterations += t.getIterations();
            res.cache = cache;
            res.missedDeadlines.addAll(t.getMissedDeadlines());
        }
        res.deadLineStats = res.missedDeadlines.
                stream().mapToLong((x) -> x).
                summaryStatistics();
        return res;
    }

    private static void printResults(BenchMarkResults results) {
        System.out.printf(
                "Benchmark completed, "
                        + "cache size at termination: %d\n"
                , results.cache.size());
        System.out.printf(
                "\tTotal iterations: %d\n",
                results.totalIterations);
        System.out.printf(
                "\tTotal cache misses: %d\n",
                results.totalCacheMisses);
        System.out.printf(
                "\tDeadline missed stats(ms): %s\n",
                results.deadLineStats);
    }

    private void printParameters() {
        System.out.println("Parameters:");
        System.out.println("==========");
        System.out.println("\tCache size: "
                + this.cacheSize);
        System.out.println("\tUsing templates: "
                + this.useTemplates);
        System.out.println("\tnThreads: "
                + this.nThreads);
    }

}
```

B.5 Utils.java

```java
package ansjob.benchmarks.lrubench.domain;

import java.util.Random;

public class Utils {

  public static class Pair<T, U> {

    public T first;
    public U second;

    public Pair(T first, U second) {
      this.first = first;
      this.second = second;
    }
  }

  public static String randomString(Random r, int length) {
    StringBuilder sb = new StringBuilder(length);
    for (int i = 0; i < length; ++i) {
      sb.append((char) r.nextInt(
          Character.MAX_CODE_POINT));
    }
    return sb.toString();
  }

}
```