NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications

Rodrigo Bruno
INESC-ID / Instituto Superior Técnico,
University of Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

Luís Picciochi Oliveira
Feedzai
Lisbon, Portugal
luis.oliveira@feedzai.com

Paulo Ferreira
INESC-ID / Instituto Superior Técnico,
University of Lisbon, Portugal
paulo.ferreira@inesc-id.pt

Abstract

Big Data applications suffer from unpredictable and unacceptably high pause times due to Garbage Collection (GC). This is the case in latency-sensitive applications such as on-line credit-card fraud detection, graph-based computing for analysis on social networks, etc. Such pauses compromise latency requirements of the whole application stack and result from applications' aggressive buffering/caching of data, exposing an ill-suited GC design, which assumes that most objects will die young and does not consider that applications hold large amounts of middle-lived data in memory.

To avoid such pauses, we propose NG2C, a new GC algorithm that combines pretenuring with user-defined dynamic generations. By being able to allocate objects into different generations, NG2C is able to group objects with similar lifetime profiles in the same generation. By allocating objects with similar lifetime profiles close to each other, i.e. in the same generation, we avoid object promotion (copying between generations) and heap fragmentation (which leads to heap compactions) both responsible for most of the duration of HotSpot GC pause times.

NG2C is implemented for the OpenJDK 8 HotSpot Java Virtual Machine, as an extension of the Garbage First GC. We evaluate NG2C using Cassandra, Lucene, and GraphChi with three different GCs: Garbage First (G1), Concurrent Mark Sweep (CMS), and NG2C. Results show that NG2C decreases the worst observable GC pause time by up to 94.8% for Cassandra, 85.0% for Lucene and 96.45% for GraphChi, when compared to current collectors (G1 and CMS). In addition, NG2C has no negative impact on application throughput or memory usage.

CCS Concepts • **Software and its engineering → Garbage collection**

Keywords Garbage Collection, Big Data, Latency

1. Introduction

Big Data applications are now part of the application stack present in most (if not all) large-scale systems. These applications are expected to work with high volumes of information efficiently and often run on top of platforms such as Cassandra [34], Lucene [41], GraphChi [33], etc. This is the case of latency-sensitive applications such as on-line credit-card fraud detection, graph-based computing for analysis on social networks or the web graph, etc.

To achieve good performance, developers often resort to optimization techniques to boost performance such as caching [47, 50, 56]. Caching is used to keep (in memory): i) the working set or intermediate results [55] (this is a common practice, for example, in graph processing systems such as GraphChi [33] and Spark [54]), or ii) consolidate writes in a database (for example, in-memory tables in Cassandra [34]). With caching, developers avoid costly operations such as recomputing intermediate values (in the case of GraphChi and Spark) or writing records to disk (in the case of Cassandra), among others. However, while keeping more data in memory helps reducing the latency for data requests, it puts more pressure on the Garbage Collection (GC) which results in long pauses of the application [8, 45].

GC pauses are unpredictable (from the application's perspective) and can stop the application for an unacceptably high amount of time leading to broken Service Level Agreements (SLAs) [13, 44, 48]. The issue is even worse if the application stack contains multiple managed heaps. If only one GC engages in a long GC pause, the whole stack is compromised and the SLA is broken.

By analyzing applications running on the HotSpot JVM, it is possible to conclude that the duration of GC pauses is dominated by the number and size of objects to copy in memory during a GC (this problem is also described in Gidra [18]). Such copies can be triggered by object promotion (which entails several steps/copies until the object finally arrives at the *Old* generation) or by a heap compaction (to reduce fragmentation). The problem with copying is that it is bound to the available hardware memory bandwidth. Increasing the capacity of nodes (e.g., the number of cores) will not reduce neither the number nor the duration of GC pauses.

Therefore, the widely accepted weak generational hypothesis stating that most objects die young [27, 29, 53] (which is a fundamental design rule for current HotSpot GCs) is not suited for many Big Data applications as the number of middle-long lived objects is high. This mismatch between the objects' real lifetime and the GC assumption that most objects die young has serious consequences for HotSpot applications with high memory utilization and tight response time targets (i.e., SLAs to cope with).

Avoiding object copying within the heap cannot (or is extremely hard to) be attained by tweaking the heap or GC parameters; thus, many application developers end up to (almost) reverse engineering the GC to understand how to avoid costly GC pauses in their applications [35] (with obvious software development productivity drawbacks). Another important difficulty is that a particular GC/heap configuration will only work for a specific environment (number of cores, size of memory, etc.), making a particular configuration not replicable.

ISMM'17, June 18, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5044-0/17/06...$15.00
http://dx.doi.org/10.1145/3092255.3092272

Previous works [5, 10, 11, 23, 30, 38] used pretenuring to reduce the amount of object copying, however, with limited success [28]. The main problem with these solutions is that simply pretenuring objects to a single older space (*Old* generation) leads to heap fragmentation since objects with different lifetime profiles are promoted to the same space. NG2C, opposed to previous pretenuring works, uses user-defined dynamic generations to group objects with similar lifetime profiles. This reduces object promotion and heap fragmentation. See Section 2 for more details.

In order to use NG2C, object allocation sites must be annotated to indicate in which generation the object should be allocated. To free the developer from the burden of understanding the objects' lifetime profiles, we developed a profiler tool, Object Lifetime Recorder (described in Section 3.5) that profiles the application and outputs where and how the code should be changed in order to take full advantage of NG2C (note that the application only needs to be profiled once).

NG2C is implemented for the OpenJDK 8 HotSpot Java Virtual Machine (JVM) as an extension of the next OpenJDK by-default GC, Garbage First (G1). Results are very encouraging as we are able to achieve our goal: avoid costly object copying (which occurs during object promotion and compactions). NG2C decreases the highest observable GC pauses by up to 94.8% for Cassandra, 85.0% for Lucene and 96.45% for GraphChi, when compared to current collectors: G1 and CMS (Concurrent Mark Sweep). In addition, application throughput and memory usage are not negatively affected by using NG2C.

To sum up, our main contributions are: i) a new GC algorithm, NG2C, that combines an arbitrary number of dynamic generations with pretenuring to avoid object copying and heap fragmentation, ii) an implementation of NG2C and the Object Lifetime Recorder (OLR) profiler tool on a production JVM, OpenJDK 8 HotSpot, and iii) the evaluation of the performance benefits of using NG2C on Big Data platforms (Cassandra, Lucene, GraphChi) using workloads and data sets based on real system utilization.

2. Related Work

NG2C proposes pretenuring combined with user-defined dynamic generations, where objects are pretenured into different generations, according to their lifetime profile. By allocating objects in different generations, per-lifetime profile, NG2C reduces object promotion and heap fragmentation. Since this work combines several well studied ideas, we dedicate this section to explaining how our research relates to previous work.

2.1 Generational Collectors

Segregating objects by age has been studied for a long time [28] as a way to take advantage of the weak generational hypothesis [53]. By promoting objects that survive a number of collections into older generations, the collector can concentrate on collecting younger generations more often (since these are more likely to contain more dead objects) [36]. The use of multiple generations (compared to using a single generation) has been shown to reduce application pauses [6, 26, 39, 49].

Opposed to previous works such as the Beltway framework [6] and the Mature Object Space collector [25], NG2C introduces the concept of dynamic generations. These user-defined generations are different from traditional generations in two ways: i) they can be created and destroyed at runtime, ii) survivor objects are promoted into the old generation. These generations are used to group pretenured objects with similar lifetime profiles.

2.2 Pretenuring and Object Demographics

Pretenuring is also a well studied technique [5, 10, 11, 23, 30, 38]. It consists on allocating objects (that are known to live for a long time) directly in older generations. By doing this, the overhead associated to object promotion is avoided. The key problem to pretenuring lies on how to estimate the lifetime profile of an object (analyzed next).

To the best of our knowledge, no pretenuring algorithm has been used to pretenure objects into multiple generations. If a collector with predefined number of generations is used (for example, two generations), pretenuring does not solve heap fragmentation, as middle-long lived objects with different lifetime profiles might be pretenured to the same heap location. To solve this problem NG2C combines pretenuring with multiple generations. By being able to pretenure into an arbitrary number (defined at runtime) of generations, NG2C avoids fragmentation. Thus, by combining these two techniques, object copying (which results from object promotion and compaction) is greatly reduced.

Pretenuring is tightly coupled with object lifetime profiling, which is used to extract object lifetime estimations, used to guide pretenuring. Extracting objects demographic information can either be performed dynamically [11, 23, 30] or statically [5, 10, 38]. Profiling information can come from stack analysis [10], connectivity graphs [21] and can also include other program's traces [38].

Our proposed profiler, (presented in Section 3.5) builds upon previous works [5, 10, 38] by resorting to stack analysis. However, opposed to previous works, it accurately estimates in which generation an object should be allocated in. In other words, our profiler answers the question of how long will the object probably live while previous profilers only tell us if the object will probably live enough to be considered old.

2.3 Region-based Garbage Collection

The hypothesis that many objects, allocated in the same scope, share the same faith, i.e., have similar lifetimes has also been leveraged by many region-based memory management algorithms [3, 7, 16, 17, 19, 20, 22, 24, 31, 42, 51]. In such algorithms, objects with similar lifetime profiles are allocated in scope-based regions, which are deallocated as a whole when objects inside these regions are no longer reachable. However, existing region-based algorithms either require sophisticated static analysis [3, 7, 16, 17, 20, 51] (which does not scale to large systems), heavily rely on manual code refactoring [19, 42] (to guarantee that objects in the same region die approximately at the same time), or support only simple programming models [37, 42, 43] (such as parallel bag of tasks).

Other region-based collectors [15] use thread-local allocation regions to allocate objects. This approach also does not support more complex models where most large data structures can be maintained by multiple threads (for example, Cassandra's in-memory tables).

NG2C can also be seen/used as a region-based collector, in which dynamic generations can be used as regions. However, opposed to typical scope-based regions, NG2C supports more complex programs such as storage platforms with minimal code changes.

2.4 Off-heap based Solutions

There are some solutions based on off-heap memory [37, 40, 42] (i.e., allocating memory for the application outside the GC-managed heap). While this is an effective approach to allocate and keep data out of the range of the GC (and therefore, reducing object copying), it has several important drawbacks: i) off-heap data needs to be serialized to be saved in off-heap memory, and deserialized before being used by the application (this obviously has performance overheads); ii) off-heap memory must be explicitly collected by the application developer (which is error prone [9, 12] and completely ignores the advantages of running inside a memory managed environment); iii) the application must always have ob-

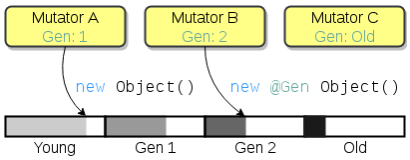


Figure 1. Allocation of Objects in Different Generations

jects identifying the data stored in off-heap (these so called header objects are stored in the managed heap therefore stressing the GC). Furthermore, as shown in Section 5, NG2C's approach outperforms off-heap memory.

To conclude our analysis of related work, to the best of our knowledge, this work is the first to: i) combine multiple generations with pretenuring, and ii) to show that it reduces object copying (coming from object promotion and compaction), that affects many HotSpot Big Data applications, thus improving applications' performance.

Although NG2C's observable benefit is the reduction of application pause times, our contribution is orthogonal to other techniques used to implement low pause time collections such as incremental (for example Immix [4] and G1 [14]) or concurrent compaction (for example C4 [52]), and to real-time collectors (for example the work in Fiji VM [46] and the Metronome [2] collector).

In fact, we envision that NG2C could be integrated with such algorithms to improve the collector's performance and reduce interference in application performance.

3. Pretenuring GC with Dynamic Generations

In this section we provide a description of the proposed solution, that combines multiple dynamic generations with pretenuring.

3.1 Heap Layout

NG2C builds upon generational collectors's [1] idea but provides an arbitrary number of dynamic generations. The concept of dynamic generation is used instead of local/private allocation region because objects are grouped by estimated lifetime/age instead of being grouped by the allocating thread.

The heap is always created with two static generations: *Young* and *Old*. By default, all objects are allocated in the *Young* generation. Upon collection (more details in Section 3.4), live objects are promoted to the *Old* generation. In other words, if no new dynamic generations are created, NG2C's heap layout is a 2-generational heap layout.

At run time, any number of dynamic generations might be created (see Section 3.2 for more details). These dynamic generations are different from the static ones (*Young* and *Old*) in two ways: i) they can be created and destroyed at runtime, ii) survivor objects are promoted into the *Old* generation.

Objects can be pretenured into any dynamic generation and also into the *Old* generation. With time, when objects become unreachable, the space previously allocated for a specific generation becomes available for other generations to use (more details in Section 3.4). In NG2C, except for the *Young*, the amount of heap space assigned to each generation is dynamic, increasing or decreasing as the amount of objects in that particular generation increases or decreases, respectively. This is possible since each generation is not implemented as a single large block of memory, but instead, as a linked list of memory regions (more details in Section 4).

3.2 Pretenuring to Multiple Generations

NG2C is designed to profit from information regarding objects' lifetime profiles (as described in Section 3.5, this information is provided by the OLR profiler). Thus, NG2C co-locates objects with similar lifetime profiles in the same generation.

Since applications might have multiple threads/mutators managing objects with different lifetime profiles (e.g., processing separate jobs), each thread must be able to allocate objects in different generations.

To efficiently support parallel allocation in multiple generations, we bind each application thread into a specific generation using the concept of current generation. The current generation indicates the generation where new objects, allocated with the `@Gen` annotation[1], will be allocated into. In practice, when a thread is created, its current generation is the *Old* generation. If the thread decides to create a new dynamic generation, this will change the thread's current generation to the new one. It is also possible to get and set the thread current generation.

More specifically, the application code can use the following calls (see Listing 1):

- `newGeneration`, creates a new dynamic generation and sets the current generation of the executing thread to the newly created generation;
- `getGeneration` and `setGeneration`, gets and sets (respectively) the current generation of the executing thread. In addition, `setGeneration` also returns the previous generation.

To allocate an object in the current generation, the `new` operator must be annotated with `@Gen`. All allocation sites with no `@Gen` will allocate objects into the *Young* generation (see Figure 1).

The code example in Listing 2 resembles a very simplified version of graph processing systems (e.g., GraphChi). It shows a method that runs several tasks in parallel threads. Each thread starts by calling `newGeneration`, to create a new dynamic generation.

Listing 1. NG2C API

```
1 // Methods added in class java.lang.System:
2 public static Generation newGeneration();
3 public static Generation getGeneration();
4 public static Generation setGeneration(Generation);
```

Listing 2. Job Processing Code Sample

```
1 public void runTask() {
2   Generation gen = System.newGeneration();
3   while (running) {
4     DataChunk data = new @Gen DataChunk();
5     loadData(data);
6     doComplexProcessing(data);}}
```

Listing 3. Data Buffer Code Sample

```
1 public class Buffer {
2    byte[][] buffer;
3    Generation gen;
4    public Buffer() {
5       gen = System.newGeneration();
6       buffer = new @Gen byte[N_ROWS][ROW_SIZE];}}
```

[1] Starting from Java 8, the `new` operator can be annotated. We use this new feature to place a special annotation that indicates that this object should go into the thread's current generation.

Then, while the task is not finished, all allocated objects using the `@Gen` annotation will be allocated in the new generation.

Listing 3 shows a code example that resembles a very simplified version of memory buffers in storage systems such as Cassandra; it shows how to use NG2C to allocate a large data structure (e.g., a buffer to consolidate database writes or intermediate data) while avoiding object copying. The constructor creates a new dynamic generation in which the buffer is allocated (using the `@Gen` annotation).

3.3 Memory Allocation

NG2C allows each thread to allocate objects in any generation. This is fundamentally different from current HotSpot's allocation strategy which assumes that all newly allocated objects are placed in the *Young* generation. Hence, in order to support object allocation (pretenuring) into dynamic generations and into the *Old* generation, we extend the allocation algorithm.

In NG2C, object allocation is separated in two paths: i) fast allocation path, using a Thread Local Allocation Buffer (TLAB)[2], and ii) slow allocation path (very large object allocation).

Allocations through the `slow path` are handled in one of two ways: inside a TLAB (if there is enough free space), or directly in the current Allocation Region (AR)[3] (outside a TLAB). Note that for each generation, there is one AR.

The high level algorithm is depicted in Algorithms 1 and 2. For the sake of simplicity, and without loss of generality, we keep the algorithm description to the minimum, only keeping the most important steps.

A call to `Object Allocation` starts an object allocation. If the allocation is marked with `@Gen`, the allocation takes place in the current generation which is available from the executing thread state (otherwise the object is allocated in the *Young* generation). Objects are promptly allocated from the TLAB unless there is not enough space.

Large object allocation (objects larger than a specific fraction of the TLAB size) goes directly to the current AR of the current generation (or the *Young* generation if the allocation is not annotated). If the region has enough free space to satisfy the allocation, the object is allocated. Otherwise, a new region is requested from the available regions' list within the heap. If no memory is available for a new region, a GC is triggered followed by an allocation retry. If a GC is not able to free enough memory, an error is reported to the application.

The pseudocode for allocations in TLABs is not shown because of space restrictions. Nevertheless, the code between lines 7 and 16 is already representative of how allocations inside a TLAB are conducted.

Algorithm 1 Memory Allocation - Object Allocation

```
1:  procedure OBJECT ALLOCATION
2:      size ← size of object to allocate
3:      klass ← class of object to allocate
4:      gen ← current thread generation
5:      isGen ← new operator annotated with @Gen?
6:      if isGen then
7:          tlab ← TLAB used for generation gen
8:      else
9:          tlab ← TLAB used for Young
10:     if end(tlab) − top(tlab) >= size then
11:         object ← init(klass, top(tlab))
12:         bumpTop(tlab, size)
13:         return object
14: slow path:
15:     if size >= size(tlab)/8 then
16:         returnALLOC IN REGION(klass, size)
17:     else
18:         returnALLOC IN TLAB(klass, size)
```

Algorithm 2 Memory Allocation - Allocation in Region

```
1:  procedure ALLOC IN REGION(klass, size)
2:      gen ← current thread generation
3:      isGen ← new operator annotated with @Gen?
4:      if isGen then
5:          region ← gen alloc region
6:      else
7:          region ← Young alloc region
8:      if end(region) − top(region) >= size then
9:          object ← init(klass, top(region))
10:         bumpTop(region, size)
11:         return object
12:     if isGen then
13:         region ← new gen alloc region
14:     else
15:         region ← new Young alloc region
16:     if region not null then
17:         object ← init(klass, top(region))
18:         bumpTop(region, size)
19:         return object
20:     else
21:         trigger GC and retry allocation
```

3.4 Memory Collection

In NG2C, three types of collections can take place (see Figure 2):

- minor collection: triggered when the *Young* generation has no space left for allocating new objects. Collects the *Young* generation. Objects that survived a number of collections (more details in Section 4) are promoted to the *Old* generation;
- mixed collection: triggered when the *Young* generation has no space left for allocating new objects and the total heap usage is above a configurable threshold. Collects the *Young* generation plus other memory regions from multiple generations whose amount of live data is low (more details in Section 4). Survivor objects from any of the collected memory regions are copied to the *Old* generation. Please note that, in a mixed collection, although all the regions belonging to the *Young* generation are collected, regions belonging to other generations are only collected if the percentage of live data is below a configurable threshold (the percentage of live data per region is gathered during a concurrent marking cycle, described next);
- full collection: triggered when the heap is nearly full. Collects the whole heap. In a full collection, all regions belonging to all generations are collected. All survivor objects are copied to the *Old* generation.

Note that when all regions that compose a dynamic generation are collected, the generation is discarded. If future allocations that target a specific dynamic generation that was previously discarded, the target generation is re-created before the first allocation is actually performed.

[2] A TLAB is a Thread Local Allocation Buffer, i.e., a private buffer where the thread can allocate memory without having to synchronize with other threads.

[3] An Allocation Region is used to satisfy allocation requests for large objects and also for allocating TLABs. Whenever an AR is full, a new one is selected form the list of available regions.

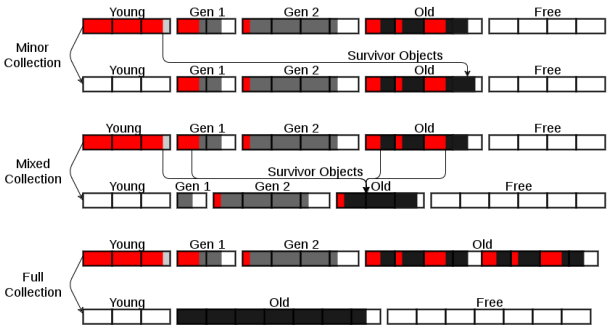


Figure 2. Types of collections (red represents unreachable data)

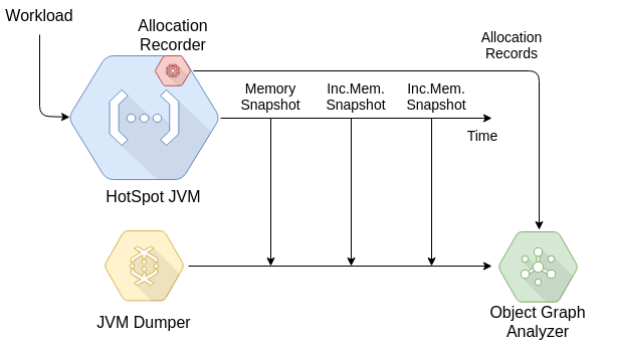


Figure 3. Object Lifetime Recorder Profiler Architecture

Concurrent marking cycles are triggered when the heap usage exceeds a configurable threshold. During a marking cycle, the GC traverses the heap and marks live objects. As the name indicates, most of this process is done concurrently with the application. When the marking phase ends, the GC frees all regions containing only unreachable (i.e., unmarked) objects. For the regions that still contain reachable content, the GC saves some statistics (used for example in mixed collections) on how much memory can be reclaimed if a particular region is collected.

3.5 Object Lifetime Recorder

To enable developers to take full advantage of NG2C, we developed the Object Lifetime Recorder (OLR) profiler, a HotSpot JVM profiler that records object allocation sites and lifetimes. Using OLR, no developer's knowledge is required to change the code in order to take advantage of NG2C.

OLR is composed by three components (see Figure 3). First, the Allocation Recorder (implemented as a Java Agent[4]) is used to: i) notify the JVM Dumper (second component, described next) when a collection finishes, and ii) record allocation sites. The second component, the JVM Dumper creates incremental heap dumps[5] (regarding previous heap dumps, taken upon previous collections) whenever a collection finishes (the JVM Dumper is notified by the Allocation Recorder when a heap dump should be taken). Compared to other heap dump tools, for example, with the `jmap` tool, incremental dumps are smaller in size (as they contain only modified memory positions), thus leading to shorter application stop times for creating the heap dump.

The third component, the Object Graph Analyzer is used to process the allocation sites and heap dumps generated during the application profiling. Objects' metadata (allocation timestamp, collection timestamp, and allocation site) is loaded into memory and an object graph is built. Then, the graph is processed in order to extract and present which objects should belong in the same generation, and where these objects are allocated.

In practice, even an inexperienced developer can change the source code of an application to take advantage of NG2C. The developer only needs to run the application using OLR's Allocation Recorder and run the JVM Dumper. When the application finishes (or after running for some time under the normal/target workload), the developer launches the Object Graph Analyzer that outputs where and how the code should be modified. With this information, even an inexperienced programmer can change the code locations suggested by the OLR in minutes.

We measured a performance cost (throughput) of up to 4 times when running the profiler on the systems we use in the evaluation. However, note that the profiler only has to run once and that the code changes proposed by the profiler lead to significant performance improvements in production settings (as observed in Section 5). In other words, despite the fact that the profiler analyzes the application with reduced throughput, the captured allocation and collection patterns hold true in a production setting.

4. Implementation

NG2C is implemented on top of the Garbage First (G1) GC [14]. G1 is the most recent and advanced GC algorithm available for the OpenJDK HotSpot JVM 8. In addition, G1 is the new default GC in the HotSpot JVM. NG2C builds upon G1, by adding approximately 2000 LOC. NG2C is integrated with G1 in the way that applications that do not use the `@Gen` annotation will run using the G1 collector (i.e., the code introduced by NG2C is never activated). This has the great benefit that all the effort put into developing G1 ensures that NG2C works with the same performance to G1 for all applications. For the rest of this section, we describe how we modified G1 for supporting pretenuring into multiple generations.

By using G1's as our code base, we inherit many techniques that are already well implemented and tested. In other words, we are using all the GC techniques already implemented in G1 (such as heap region management, remembered sets management, safepoints, write barriers, and concurrent marking) to support NG2C's implementation.

G1 uses a heap divided in equally sized memory regions. It contains two generations, the *Young* and the *Old*. The first is divided into three spaces [53] (*Eden*, and two survivor spaces, *To*, and *From*). NG2C maintains both these generations with the exact same structure and semantic.

Additional dynamic generations are created by allocating regions from the free memory regions list (also available in G1). The existing code in G1 looks at NG2C's dynamic generations as part of G1's *Old* generation. This means that we reuse G1's write barrier and remembered set for inter-generational pointers.

NG2C's inherits G1's collector algorithms without almost any change. Minor, mixed, and full collections work in the exact same way in both NG2Cand G1. The only modification is that, in NG2C, the collector can promote objects from dynamic generations into the *Old* generation, while in G1, the collector either only promotes from *Young* to *Old* or compacts regions belonging to the *Old* generation.

Most of the code introduced by NG2C, lies in the object allocation path. In the next sections we describe how the new alloca-

[4] A Java Agent is a small pluggable component that can be attached to the JVM, being able to analyze its state.

[5] A heap dump is a memory snapshot (taken while the application is running) of the Java heap (where all the application objects reside). We create incremental dumps using CRIU, a Checkpoint/Restore tool for Linux processes.

tion algorithm works and how the byte code interpreter and Just-In-Time compiler are adapted to work with it.

4.1 Parallel Memory Allocation

Contention in memory allocation is a well-known problem [17, 28]; memory allocation must be synchronized between threads so that each memory block is used by a single thread. In G1 this is achieved by using Thread Local Allocation Buffers (TLABs) and Allocation Regions (ARs). Therefore, whenever a thread needs to allocate some memory, it allocates directly from its TLAB. If the TLAB is full, the thread must allocate memory from the current AR. This allocation, however, will only occur after the thread acquires a lock on that AR. If the AR does not have enough available space, a new AR is allocated directly from the list of free regions (this step requires even further locking to ensure that no other thread is allocating another region).

In NG2C, we extend the use of both TLABs and ARs to multiple generations (the complete algorithm is presented in Section 3.3). Since each thread can now allocate memory in multiple generations, multiple TLABs are necessary to avoid costly memory allocations. The TLAB to use for each allocation is decided at runtime, based on the use of `@Gen` annotations (see Section 4.3 for more details). Additionally to TLABs, NG2C also extends the use of ARs to multiple generations. Therefore, whenever a TLAB used for a particular generation is full, an allocation request is issued directly to the AR of the specific generation.

By using multiple TLABs and ARs (one for each generation), allocations are more efficient as fewer synchronization barriers exist compared to not using (TLABs and ARs). This, however, introduces a problem: as any thread can allocate memory in any generation, each thread must have a TLAB in each generation (even if that thread never allocates memory in that particular generation). As the number of generations grow, more and more memory is wasted for allocating TLABs that are never actually used.

To solve the aforementioned problem, NG2C never actually allocates any memory for TLABs when we create a new generation. Memory for each TLAB is effectively allocated only upon the first allocation request. This way, threads will have TLABs (with allocated memory) only for the generations that are being used (and not for all the existing generations).

4.2 `@Gen` Annotations

For allocating memory in generations other than the *Young* generation, we considered several options: i) simply calling the JVM to switch the generation to use for allocation; ii) add a new `new` operator with an extra argument (target generation); iii) annotate the `new` operator.

We opted for the last option for the following reasons. The first was immediately ruled out because it is very difficult to control which objects go into non-young generations; e.g., naïve `String` manipulation can easily result in many allocations that would potentially go into a non-young generation. The second option (creating a new allocation operator) would force us to extend the Java language, and the compiler.

A clear advantage of using annotations is its simplicity; however, it has one disadvantage: we must call the JVM whenever we need to change the current target generation. However, in practice and according to our experience, this almost never imposes a relevant overhead because: i) a thread handling a particular task will most probably only need one generation (worker threads tend to use one generation at a time), and ii) large object allocation and copying is much more expensive than calling the JVM to change the target generation (therefore it pays off to allocate a large object in the correct generation). In both cases, the cost of calling the JVM is absorbed and the overhead becomes negligible (see Section 5 where we show that NG2C does not decrease the application throughput). Also note that getting and setting the current generation does not require any locking as it only changes a field in the current thread's internal data structure.

4.3 Code Interpreter and JIT

The OpenJDK HotSpot uses a combination of code interpretation and Just-in-Time (JIT) compilation to achieve close to native performance. Therefore, whenever a method is executed for the first time, it is interpreted. If the same method is executed for a specific number of times, it is then JIT compiled. This way, the JVM compiles (a costly operation) only the methods where there is benefit (since executing compiled code is much faster than interpreting it).

In order to comply with such techniques in NG2C, we modify both the interpreter and the JIT compiler to add the notion of generations. To be more precise, we had to detect if the allocation is annotated with `@Gen` and, if so, which generation is being targeted (choose the correct TLAB).

Selecting the correct TLAB to allocate is done as follows. For each thread, NG2C keeps a pointer to the current generation TLAB. This pointer is only updated when the thread calls `newGeneration` or `setGeneration`. Then, if the current allocation site is annotated with `@Gen`, the current generation TLAB is used.

Detecting if the current allocation is annotated with `@Gen` is done differently before (interpretative mode) and after JIT compilation. Before JIT, we use a map of byte code index to annotation, that is stored along the method metadata (this map is prepared during class loading). Using this map, it is possible to know in constant time if a particular byte code index is annotated with `@Gen` or not. Upon JIT compilation, the decision of whether to go for the *Young* generation or not is hardcoded into the compiled code. This frees the compiled code (after JIT) from accessing the annotation map.

5. Evaluation

We now evaluate the performance of NG2C while comparing it with G1, CMS. We also have results for C4. Although not being an OpenJDK collector, C4 comes from a similar JVM, Zing[6]. Since we only have one license available, we could not run all experiments with it.

We use three relevant platforms that are used in large-scale environments: i) Apache Cassandra 2.1.8 [34], a large-scale Key-Value store, ii) Apache Lucene 6.1.0 [41], a high performance text search engine, and iii) GraphChi 0.2.2 [33], a large-scale graph computation engine. A complete description of each workload, including how the source code was changed (with the help of OLR profiler), is presented in Section 5.2.

For evaluating NG2C, we are mostly concerned on showing that, compared with other collectors, NG2C: i) does reduce application pause times; ii) does not have a negative effect with throughput nor for memory utilization; iii) greatly reduces object copying; iv) does not increase the remembered set management work.

5.1 Evaluation Setup

We evaluate NG2C in three different environments (Table 1 provides a summary of the evaluation environments). First, we use Feedzai's[7] internal benchmark environment. This environment mirrors a real-world deployment and uses a Cassandra cluster to store data. For Feedzai, it is very important to keep Cassandra's GC pauses as short as possible to guarantee that client SLAs are not

[6] Zing is a JVM developed by Azul Systems (www.azul.com).

[7] Feedzai (www.feedzai.com) is a world leader data science company that detects fraud in omnichannel commerce. The company uses near real-time machine learning to analyze big data to identify fraudulent payment transactions and minimize risk in the financial industry.

Platform	Workload	CPU	RAM	OS	Heap Size	*Young* Gen Size	LOC Changed
Cassandra	Feedzai	Intel Xeon E5-2680	64 GB	CentOS 6.7	30 GB	4 GB	22
Cassandra	WI,RW,RI	Intel Xeon E5505	16 GB	Linux 3.13	12 GB	2 GB	22
Lucene	RW	AMD Opteron 6168	128 GB	Linux 3.16	120 GB	2 GB	8
GraphChi	PR,CC	AMD Opteron 6168	128 GB	Linux 3.16	120 GB	6 GB	9

Table 1. Evaluation Environment Summary

broken by long query lantencies. The Cassandra cluster is composed by 5 nodes.

Second, we use a separate node to evaluate NG2C with Cassandra under three different synthetic workloads with varying number of read and write operations (more details in Section 5.2.1): Write-Intensive (WI), Write-Read (WR) and Read- Intensive (RI).

Given the size of the data sets used for Lucene (Wikipedia dump) and GraphChi (Twitter graph dump), we use another separate node to evaluate NG2C. On top of Lucene we perform client searches while continuously updating the index (read and write transactions). For GraphChi, we use two workloads, PageRank and Connected Components. More details in Sections 5.2.2 and 5.2.3 (for Lucene and GraphChi workloads, respectively).

Each experiment runs in complete isolation for at least 5 times (i.e., until the results obtained become stable). Feedzai's workload runs for 6 hours, while all other workloads run for 30 minutes each. When running each experiment, we never consider the first minute of execution (in Feedzai's benchmarks we disregard the first hour of execution to allow other external systems to converge). This ensures minimal interference from JVM loading, JIT compilation, etc.

We always use fixed heap and *Young* generation sizes (see Table 1). We found that these sizes are enough to hold the working set in memory and to avoid premature massive promotion of objects to older generations (in the case of CMS and G1). Table 1 also reports the number of lines changed after using the OLR profiler.

5.2 Workload Description

We use this section to provide a more complete description of the workloads used to evaluate NG2C.

5.2.1 Cassandra

We use Cassandra under 4 different workloads: i) Feedzai's workload (consisting of 500 read queries and 25000 write queries per second, for the whole Cassandra cluster); ii) write intensive workload (2500 read queries and 7500 write queries per second); iii) read-write workload (5000 read queries and 5000 write queries per second); iv) read intensive workload (7500 read queries and 2500 write queries per second).

Note that Feedzai's workload is based on representative data from real deployments of their product (i.e., fraud detection). All workloads besides Feedzai's are synthetic but mirror real-world settings (e.g., we use the YCSB benchmark tool).[8] When running Cassandra in Feedzai's cluster, we setup the JVM with 30GB of heap and we fix the *Young* generation to 4GB. Note that we tested several heap sizes and found these ones to be particularly good for short GC pause times.

To use NG2C we profiled Cassandra using the OLR profiler. The code was mainly modified to allocate all objects belonging to a particular `Memtable`[9] in a separate dynamic generation. Thus, whenever a new `Memtable` is created or flushed, we create a new dynamic generation. Each `Memtable` contains a B-Tree (self-balancing tree data structure) with millions of objects. These objects contain references to buffers with real data. To take advantage of NG2C, we allocate all objects and buffers belonging to a particular `Memtable` in the dynamic generation created for that specific `Memtable`.

In total, we changed a total of 22 code locations: i) 11 code locations where we annotate the `new` operator, and ii) 11 code locations where we create, or change generation.

5.2.2 Lucene

We use Lucene to build an in-memory text index using a Wikipedia dump from 2012.[10] The dump has 31GB and is divided in 33M documents. Each document is loaded into Lucene and can be searched.

The workload is composed by 20000 writes (document updates) and 5000 reads (document searches) per second; note that this is a write intensive workload which represents a worst case scenario for GC pauses. For reads (document queries), we loop through the 500 top words in the dump ; this also represents a worst case scenario for GC pauses.

When running Lucene, we use all available cores (48 cores), the heap size is limited to 120GB with a 2GB *Young* generation size. Again, we tested with different heap sizes and we found out that this value is beneficial for short GC pauses.

To reduce Lucene's GC pauses we profiled it with the OLR profiler. The code was mainly modified to allocate documents' data (of the Wikipedia dump) in the *Old* generation. Objects created to hold the indexes of documents will live throughout the application lifetime; therefore, if we do not use NG2C such objects would be copied within the heap (thus leading to long GC pauses). With NG2C, most objects holding the index (including objects such as `Term`, `RAMFile` and `byte` buffers) are allocated outside the *Young* generation. To accomplish it, we changed 8 code locations in Lucene, all of which to annotate the `new` operator.

5.2.3 GraphChi

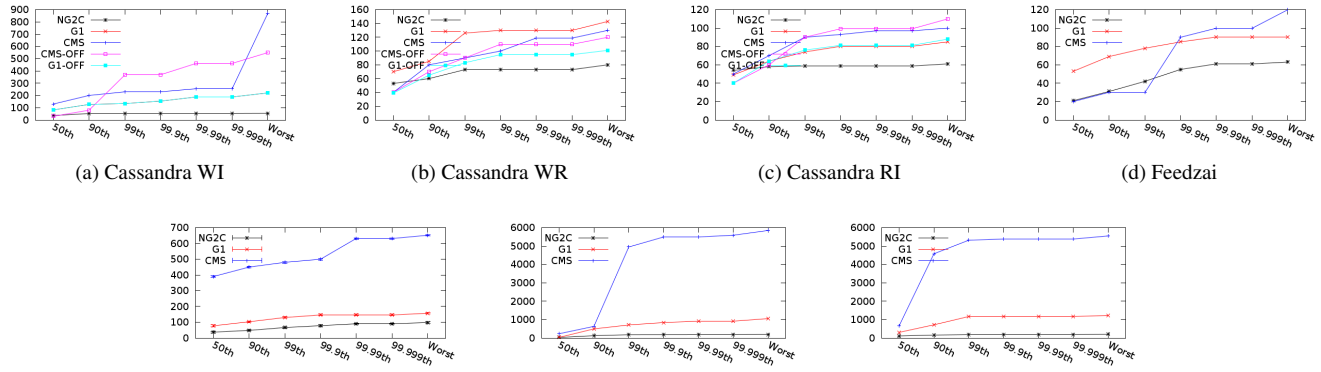
When compared to the previous systems (Cassandra and Lucene), GraphChi is a more throughput oriented system (and not latency oriented). However, we use GraphChi for two reasons: i) we want to demonstrate that NG2C does not decrease throughput even in a throughput oriented system; ii) with NG2C, systems such as GraphChi can now be used for applications providing latency oriented services, besides performing throughput oriented graph computations.

In our evaluation, we use two well-known algorithms: i) page rank, and ii) connected components. Both algorithms are feed with a 2010 twitter graph [32] consisting of 42 millions vertexes and 1.5 billions edges. These vertexes (and the corresponding edges) are loaded in batches into memory; similarly to Cassandra' `Memtables`, GraphChi calculates a memory budget to determine the number of edges to load into memory before the next batch. This represents an iterative process; in each iteration a new batch of vertexes is loaded and processed.

[8] The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source benchmarking tool often used to compare NoSQL database systems.

[9] A `Memtable` table buffers recent writes in memory. When a `Memtable` is full, a flush is scheduled and a new `Memtable` is created. The capacity of each `Memtable` is proportional to the JVM heap size.

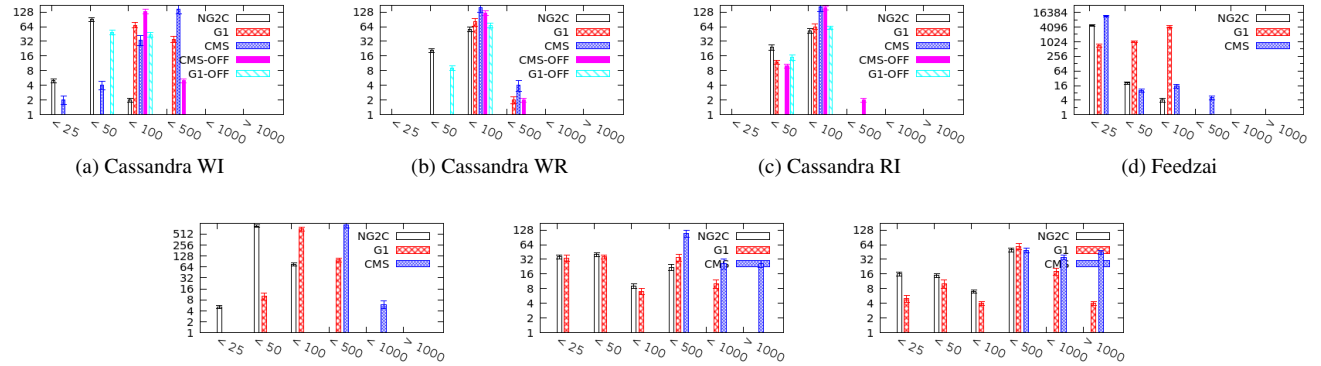
[10] Wikipedia dumps are available at dumps.wikimedia.org



(a) Cassandra WI (b) Cassandra WR (c) Cassandra RI (d) Feedzai

(e) Lucene (f) GraphChi CC (g) GraphChi PR

Figure 4. Pause Time Percentiles (ms)



(a) Cassandra WI (b) Cassandra WR (c) Cassandra RI (d) Feedzai

(e) Lucene (f) GraphChi CC (g) GraphChi PR

Figure 5. Number of Application Pauses Per Duration Interval (ms)

When running GraphChi, we use all available cores (48 cores), the heap is limited to 120GB, and the *Young* generation is limited to 6GB (we measured with different sizes and we found that this provides the shortest GC pause times in the current environment and workload).

To take advantage of NG2C, we profiled GraphChi with the OLR profiler. The code was mainly modified to allocate objects representing graph vertexes (`ChiVertex`), edges (`Edge`), and internal pointers (`ChiPointer`) in multiple dynamic generations (one per batch). We modified a total of 9 code locations, in which we annotate the `new` operator.

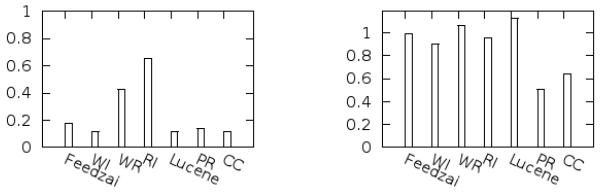
5.3 GC Pause Times

Figure 4 presents the GC pause times for each GC (CMS, G1, and NG2C) and for each percentile, for all the workloads. We do not show pause times for C4 because it is a concurrent collector and therefore, the application should never be paused. In practice, using C4, we got pauses of only up to 15 milliseconds for Cassandra.

In Feedzai's workload, GC pauses are shorter when compared to the other Cassandra workloads. This is mainly because the hardware used in Feedzai achieves better performance compared to the one used for running the other Cassandra workloads. Still regarding Feedzai's workload, CMS shows shorter GC pauses for lower percentiles but shows the worst results in higher percentiles (25% worse than G1 and 47% worse than NG2C). G1 shows more stable GC pause times (when compared to CMS) as it does not lead to long pauses in higher percentiles; NG2C shows GC pause times very similar to CMS in lower percentiles, and it shows shorter GC pause times for higher percentiles as well.

The other Cassandra workloads (WI, WR, and RI) differ only in the percentage of read and writes. From the GC perspective, more writes means that more objects are kept in memory (which results in more object copies and therefore longer GC pauses). This obviously applies to Cassandra because it buffers writes in memory. This is clearly observable by comparing the GC pauses across the three workloads (WI, WR, and RI) for CMS and G1. RI workload shows shorter GC pauses than WR and WI, while WR shows shorter pauses than WI but longer than RI. According to our results, CMS is more sensitive to writes (than the other two collectors) as it has a steep increase in the GC pause time as we move towards write intensive workloads. G1 has a more moderate increase in GC pause time in more intensive workloads.

Regarding NG2C, it produces a different behavior as it shows shorter GC pauses for lower percentiles in WI, and longer pauses for WR in higher percentiles. One factor contributes for this difference (between NG2C, and G1 and CMS): we profiled (using OLR profiler) Cassandra under the WI workload. This means that the read path is not as optimized as the write path. Therefore, in write intensive workloads, NG2C is more optimized than in read intensive workloads. This is also observable by measuring the difference



(a) Object Copy Norm. to G1 (b) Rem. Set Update Norm. to G1

Figure 6. NG2C Object Copy and Remembered Set Update

	Max Mem Usage			Throughput		
	CMS	G1	C4	CMS/OFF	G1/OFF	C4
Feedzai	.92	1.00	-	-	-	-
WI	.96	1.01	1.73	1.07/1.08	.99/1.01	.70
WR	.80	1.00	2.04	.76/.90	.93/0.73	.67
RI	.73	.98	1.94	.86/1.18	.90/0.65	.71
Lucene	.39	.98	-	.59	.87	-
PR	1.44	1.04	-	.80	.96	-
CC	1.43	1.17	-	1.03	.96	-

Table 2. Max Memory Usage and Throughput norm. to NG2C (i.e., NG2C value is 1 for all entries)

between the GC pause times in higher percentiles; as we move towards write intensive workloads, the difference between NG2C and other GCs increases.

We also have results for Cassandra with the off-heap memory enabled for CMS and G1 (Cassandra uses off-heap memory to store values while the keys remain in the managed heap). Using off-heap reduces GC pause times by up to 50% in the WI workload (versus 93.8% using NG2C), around 20% in the WR workload (versus 39% using NG2C), and shows no improvement for the RI workload (versus 61% using NG2C). In sum, using NG2C is more effective to reduce GC pause times than using off-heap memory mainly because Cassandra needs to keep header objects in the memory managed heap to describe the contents stored in off-heap. In the case of Cassandra (key-value store), keys are stored in the managed heap and therefore contribute for long application pauses. NG2C is able to move all key-value pairs into a specific dynamic generation (thus avoiding pause times).

The remaining workloads (Lucene, PR, and CC) are all write intensive. CMS shows very high GC pause times compared to the other two GCs. G1 shows a more moderate increase in GC pause times, when compared to CMS, but is still worse than NG2C. In sum, NG2C clearly improves the worst observable GC pause times by: 85% (CMS) and 38% (G1) in Lucene, 97% (CMS) and 84% (G1) in PR, and 97% (CMS) and 82% (G1) for CC.

Figure 5 presents the average and standard deviation for the number of pauses in different duration intervals. Results show that: i) NG2C does not increase the number of pauses, and ii) it moves pauses to smaller duration intervals. CMS presents the worst results by having the most amount of pauses in longer pause intervals.

5.4 Object Copy and Remembered Set Update

We now look into how much time is spent: i) copying objects within the heap, and ii) updating remembered set entries, upon a collection. Note that the remembered set updates is an important metric since pretenuring can lead to high number of remembered set updates because of the potential increase in the number of references coming from older to younger spaces [28]. We only show results for G1 and NG2C, given that CMS and C4 do not provide such logging information. However, both metrics are similar for different generational collectors because they mostly depend on: i) the mutator allocation speed (dictates how fast minor collections are triggered and how many objects are promoted), and ii) the available hardware memory bandwidth. Both these factors are kept constant across GCs (G1 and NG2C).

Figure 6 presents results for total object copying time and remembered set update time during each workload. All results are normalized to G1. Results show that NG2C reduces objects copying between 30.6% and 89.2%. Note that, in G1, we can not differentiate between object promotion and object compaction since the collector collects both young and old regions at the same time (during mixed collections).

NG2C also has a positive impact for the remembered set update work. This means that, in NG2C, there is not an increase in the number of inter-generational references pointing to the *Young* generation. This is possible because objects referenced by pretenured objects are most likely to be pretenured as well. NG2C even reduces the amount of remembered set update work for most workloads since it reduces the amount of premature promotion in G1 (objects with short lifetimes that were allocated right before a minor collection and were prematurely promoted). This also means that NG2C puts less pressure on the write barrier (compared to G1).

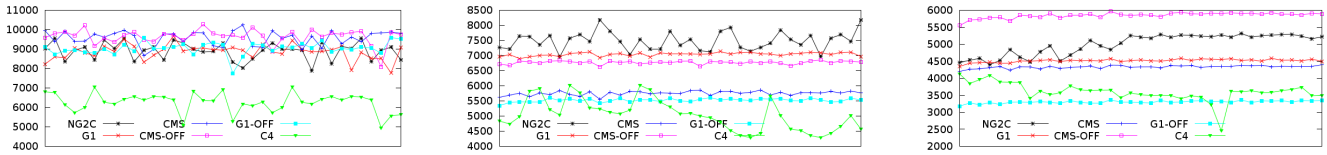
5.5 Memory Usage

In this section, we look into the max memory usage to understand how NG2C relates to other collectors regarding heap requirements (see Table 2). Regarding the workloads' max heap size: Feedzai workload has 30GB, while the other Cassandra workloads (WI, WR, and RI) have 12GB; each Lucene and GraphChi's workload (PR and CC) have 120GB.

From Table 2 we can conclude that, regarding Cassandra workloads (i.e., Feedzai, WI, WR, and RI) all collectors (excluding C4) have a very similar max memory usage. CMS has a slightly smaller heap (compared to G1 and NG2C) while NG2C has a slightly larger heap (compared to G1 and CMS). This slight increase comes from the fact that dynamic generations are only collected upon a mixed collection, which is only triggered when the heap usage is above a configurable threshold. This can lead to a slight delay in the collection of some objects that are already unreachable. C4 has a considerably higher memory usage since it reserves approximately 75% (12GB) of the system's memory, when the JVM is launched. We were unable to extract the actual memory usage during execution. We do not show the results for C4 with other workloads because we only have one license (for one physical node).

Lucene max memory utilization is lower for CMS when compared to G1 and NG2C. These larger heap sizes in G1 and NG2C comes from humongous allocations. Using this technique, very large objects are directly allocated in the *Old* generation. It has the clear drawback of delaying the collection of such very large objects. Since CMS does not have such technique (i.e., all objects are allocated in the `Eden`), CMS tries to collect these large objects upon each minor collection, leading to faster collection of such objects, thus achieving lower heap usage. Comparing G1 with NG2C, the heap usage is similar.

Regarding GraphChi, it shows a different memory behavior when compared to Cassandra and Lucene, as it allocates mostly small objects. Most of these small objects (mostly data objects representing vertexes and edges) are used in a single iteration, which is long enough for them to be promoted into the *Old* generation (in the case of CMS and G1). Since we set the maximum heap size to 120GB, the heap fills up until a concurrent marking cycle is triggered. In CMS, the concurrent marking cycle is triggered a bit later compared to G1 and NG2C (thus leading to an increase in the max



(a) Cassandra WI (b) Cassandra WR (c) Cassandra RI

Figure 7. Cassandra Throughput (transactions/second) - 10 minutes sample

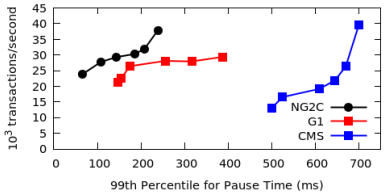


Figure 8. Throughput vs Pause Time

heap usage). Regarding G1 and NG2C, both present similar max heap values.

5.6 Application Throughput

We now discuss the throughput obtained for each GC and workload (except for Feedzai). We do not show the throughput for Feedzai's workload because the benchmark environment (where the Cassandra cluster is used) dynamically adjusts the number of transactions per second according to external factors; e.g., the credit-card transaction generator produces different transactions through time, some result on more Cassandra transactions than others, thus making it infeasible to reproduce the same workload multiple times. The throughput for all remaining workloads is presented in Table 2. Throughput for Cassandra using off-heap is shown for WI, WR, and RI workloads. All results are normalized to NG2C.

From Table 2, we conclude that NG2C outperforms CMS, G1, and C4 (we could only obtain results for Cassandra workloads using C4 because we only have one license) for most workloads. Figure 7 shows the throughput evolution for Cassandra workloads. NG2C is the solution with overall best throughput across the three workloads. Only the CMS collector using off-heap outperforms NG2C in the read intensive workload (by approximately 18%).

For all previous experiments, we use latency oriented GC configurations, i.e., the configurations we found to enable shorter GC pause times in higher percentiles. This, however, has the drawback of potentially decreasing the throughput. Among the used workloads, the most explicit example of this throughput decrease is Lucene running with CMS, in which a throughput oriented GC configuration, i.e., the configuration we found to enable higher throughput, could increase the throughput by up to 3x (when compared to the throughput achieved with a latency oriented configuration).

To better understand the trade-off between throughput and latency, we ran the Lucene workload with 6 different *Young* generation sizes. We found that this parameter alone allows one to achieve good latency (if the size is reduced) or good throughput (if the size is increased). Other GC parameters did not have a relevant effect and therefore we keep them fixed. We start with the configuration used in the previous sections (2 GB). Then, we keep increasing the size of the *Young* generation by 2 GB.

Figure 8 shows a plot with the relation between throughput and GC pause time, for each GC, in which each point on each line represents a different *Young* generation size. CMS shows always longer GC pauses independently of the GC configuration. It also shows a steep increase in the throughput, with a small increase in the GC pause time; this shows how easy it is to dramatically reduce throughput when CMS is configured for latency. On the other hand, G1 shows much shorter GC pauses than CMS at the cost of some reduced throughput. Note that moving from latency oriented to throughput oriented configurations has a small impact on throughput, but has a larger negative impact on GC pause time. Finally, NG2C provides the shortest GC pause times with a very small throughput impact. In the most throughput oriented configuration (point on the top of the curve), NG2C is only 5% worse than CMS and the GC pause time is 66% better. In conclusion: i) CMS can be difficult to configure for short GC pause time (while keeping an acceptable throughput); ii) G1 leads to shorter pauses but can damage throughput; iii) NG2C keeps up with the best throughput achieved by CMS, while also reducing the GC pause times by 66% and 39% w.r.t. CMS and G1, respectively.

6. Conclusions and Future Work

This paper presents the design and implementation of NG2C,[11] a new HotSpot GC algorithm that avoids copying objects within the heap by aggregating objects with similar lifetime profiles in separate generations. NG2C is built on top of G1, by modifying the way it allocates objects and manages generations. The experimental evaluation shows that it is possible to reduce the object copying done by current collectors (G1 and CMS) by up to 89.2%, resulting in shorter GC pause times. We are able to reduce the worst observable GC pause times in Cassandra by 94.8%, 85% for Lucene, and 96.45% for GraphChi. We also show that despite increasing the complexity of the JVM allocation algorithm, NG2C does not penalize application throughput, the heap usage, and the remembered set update work, when compared to current GC implementations.

We envision that the NG2C could be integrated in other JVMs and collectors. Even concurrent collectors such as C4 could take advantage of the ideas described in this work to reduce the amount of object copying within the heap and therefore reduce the application interference, possibly increasing the throughput. We are currently working on integrating the object lifetime estimation directly into the JVM in order to allow online pretenuring into multiple generations.

7. Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and through the FCT scholarship SFRH/BD/103745/2014.

[11] Both NG2C and the OLR profiler can be downloaded from `github.com/rodrigo-bruno/ng2c`

References

[1] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Softw. Pract. Exper.*, 19(2):171–183, Feb. 1989. ISSN 0038-0644. doi: 10.1002/spe.4380190206.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 81–92, New York, NY, USA, 2003. ACM. ISBN 1-58113-647-1. doi: 10.1145/780732.780744.

[3] W. S. Beebee Jr and M. Rinard. An implementation of scoped memory for Real-Time Java. In *International Workshop on Embedded Software*, pages 289–305. Springer, 2001.

[4] S. M. Blackburn and K. S. McKinley. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375586.

[5] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 342–352, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504307.

[6] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting Around Garbage Collection Gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 153–164, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512548.

[7] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership Types for Safe Region-based Memory Management in Real-time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 324–337, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781168.

[8] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A Bloat-aware Design for Big Data Applications. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 119–130, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2100-6. doi: 10.1145/2464157.2466485.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1454-1. doi: 10.1145/2338965.2336769.

[10] P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 162–173, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277718.

[11] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754181.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[13] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2): 74–80, Feb. 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794.

[14] D. Detlefs, C. Flood, S. Heller, and T. 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. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029879.

[15] D. Doligez and X. Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158611.

[16] D. Gay and A. Aiken. Language Support for Regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378815.

[17] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 82–93, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X.

[18] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451142.

[19] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, D. Murray, S. Hand, and M. Isard. Broom: Sweeping out Garbage Collection from Big Data Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 2–2, Berkeley, CA, USA, 2015. USENIX Association.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563.

[21] S. Z. Guyer and K. S. McKinley. Finding Your Cronies: Static Analysis for Dynamic Object Colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 237–250, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1028996.

[22] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512547.

[23] T. L. Harris. Dynamic Adaptive Pre-tenuring. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 127–136, New York, NY, USA, 2000. ACM. ISBN 1-58113-263-8. doi: 10.1145/362422.362476. URL `http://doi.acm.org/10.1145/362422.362476`.

[24] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with Safe Manual Memory-management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029883.

[25] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *Memory Management*, pages 388–403. Springer, 1992.

[26] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting the World: One Car at a Time. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 162–175, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.264353.

[27] R. Jones and C. Ryder. Garbage collection should be lifetime aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 182–196, 2006.

[28] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[29] R. E. Jones and C. Ryder. A Study of Java Object Demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375652.

[30] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic Object Sampling for Pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 152–162, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029892.

[31] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring Code Safety Without Runtime Checks for Real-time Control Systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 288–297, New York, NY, USA, 2002. ACM. ISBN 1-58113-575-0. doi: 10.1145/581630.581678.

[32] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751.

[33] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[34] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922.

[35] P. Lengauer and H. Mössenböck. The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 111–122, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2733-6. doi: 10.1145/2568088.2568091.

[36] H. Lieberman and C. Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983. ISSN 0001-0782. doi: 10.1145/358141.358147.

[37] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.*, 9(12):936–947, Aug. 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994513.

[38] S. Marion, R. Jones, and C. Ryder. Decrypting the Java Gene Pool. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 67–78, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: 10.1145/1296907.1296918.

[39] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375637.

[40] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814313.

[41] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., Greenwich, CT, USA, 2010. ISBN 1933988177, 9781933988177.

[42] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 675–690, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694345.

[43] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 349–365, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

[44] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043560.

[45] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218.

[46] F. Pizlo, L. Ziarek, and J. Vitek. Real Time Java on Resource-constrained Platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-732-5. doi: 10.1145/1620405.1620421.

[47] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[48] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[49] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In *European Conference on Object-Oriented Programming*, pages 235–252. Springer, 1995.

[50] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: Increased Performance for In-memory Hadoop Jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, Aug. 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367513.

[51] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and Efficient Hybrid Memory Management for Java. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 81–92, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754185.

[52] G. Tene, B. Iyengar, and M. Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993491.

[53] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: 10.1145/800020.808261.

[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[55] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[56] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge & Data Engineering*, 27(7):1920–1948, 2015. ISSN 1041-4347. doi: doi.ieeecomputersociety.org/10.1109/TKDE.2015.2427795.