Avoiding Out-Of-Memory Errors in ThinGC

Pontus Ernstedt
Abstract

Avoiding Out-Of-Memory Errors in ThinGC

Pontus Ernstedt

A new garbage collector, called ThinGC, is producing out-of-memory errors when applied to certain user applications. ThinGC introduces a classification of objects as either hot or cold, where hot objects are currently in use by the application and cold objects are not. The heap is partitioned into two isolated areas, hot storage and cold storage, separating the differently classified objects. When cold objects are needed by the application again, they are first moved back into hot storage - a process called reheating. Reheating requires that space is available in hot storage to accommodate the object. If the reheating fails, ThinGC crashes reporting an out-of-memory error, as ThinGC disallows hot objects to reside in cold storage.

As it stands, ThinGC's approach in performing the reheat is flawed, leading to many failed reheats when running reheat-prone applications with limited memory space. This thesis sets out to reduce the amount of out-of-memory errors produced by reheats in ThinGC. A strategy which reserves hot storage space specifically for reheats is proposed, and implemented into ThinGC. Evaluation shows that the strategy is successful in avoiding applications crashing. Our implementation comes of short, however, as it puts the burden on the user to configure the amount of memory to reserve for the strategy to work well - a task for which there is no obvious optimal approach.
1 Introduction

The idea of manual memory management is antiquated to many. While there are notable outliers (C and C++ especially), most mainstream programming languages in use today include garbage collection of some sort, freeing programmers from the burden of managing memory themselves. In Java, for example, developers have the option of choosing between several different collectors when starting the Java HotSpot Virtual Machine (HotSpot JVM), each with different performance characteristics [6].

In a collaboration between Oracle and researchers at Uppsala University, a new garbage collector, called ThinGC, has been designed and implemented as part of HotSpot JVM [4]. ThinGC is built on top of the Z Garbage Collector (ZGC), available in OpenJDK distributions as of version 11 [8]. ThinGC introduces a classification of objects as either hot or cold – that is, objects which have recently been read or written to, and objects which have not, respectively. Using this classification, ThinGC partitions memory into two separate spaces – hot storage and cold storage. Each space is managed individually, running separate collectors concurrently with each other.

Cold objects (that is, objects that are not accessed for some time) are moved into cold storage via a process called freezing. Objects in cold storage are never directly accessed by application threads. Instead, whenever a cold object is about to be read or written, the object is first moved back into the hot storage – a process called reheating – maintaining an invariant that application threads only directly interact with objects in hot storage.

Reheating requires that free space enough to accommodate the objects is available on an unused page of the hot storage, on to which the object can be moved. It is not uncommon, however, for all pages to be in use simultaneously, especially if an application’s memory footprint gets close to the maximum allowed. If the reheating process fails to receive an unused page, ThinGC makes no further attempt at performing the reheat. Instead, the application crashes reporting an out-of-memory (OOM) error, as the object otherwise would need to be accessed directly in cold storage, thus breaking the invariant. As such, it is desirable to minimise the chance of this happening, to make ThinGC applicable in even more contexts.

1.1 Purpose and Goals

In this thesis, we set out to reduce the amount of out-of-memory errors produced by reheats in ThinGC. Specifically, our goal is to to explore the consequences of adding reserves of pages that can be used as backups whenever the reheating process fails to get unused pages by other means. If we succeed, ThinGC can be run with smaller heap sizes without crashing. Furthermore, we aim to avoid introducing any significant performance overhead
into ThinGC, especially when applied on applications that never produced OOM errors in the first place.

We test whether or not we are successful by running our modified ThinGC on selected benchmarks with heap sizes known to produce OOM errors prior to our work. We compare to both ZGC and ThinGC prior to our work to see how the different builds hold up at varying heap sizes. Furthermore, we run performance tests to see if our additions have added any noticeable overhead compared to how it was prior to our work.

1.2 Outline

We start by giving an overview of ZGC and ThinGC in §2 as well as a primer on some common garbage collection strategies. In §3 we consider possible mitigation strategies, including tweaking of switches present in ThinGC; adding a reserve of pages specifically for reheating; and making said reserve scale dynamically. We provide an implementation using the reserve-strategy, which we discuss in §4. We evaluate our implementation by comparing it to both ZGC and ThinGC as it was prior to our work in §5. We discuss work related to ThinGC in §6. Our work is concluded in §7.

2 Background

In this section we will review some background material that sets the context in which the rest of the thesis takes place. Specifically, we will review some general garbage collection terminology and strategies relevant for the thesis, after which we give an overview of ZGC. Finally, we introduce ThinGC (that is built on top of ZGC), whose design gives rise to the OOM errors that we set out to avoid.

2.1 A Garbage Collection Primer

Garbage collection (GC) [7] is the process of automatically identifying and reclaiming memory of objects that are no longer in use. Such objects are referred to as garbage. The main purpose of garbage collection is to free the programmer of the burden of manual memory management. Many GC strategies have been developed over the years as a result of shifting hardware capacities and software needs. Modern collectors often use a combination of multiple fundamental techniques.

Tracing garbage collection can be viewed as a family of collection techniques. Their common denominator is how they identify which objects should be deallocated, using the concept of reachability. Tracing collectors traverse the object graph by starting in some root-set (consisting of object references found on call stacks and in global variables), and following each reference
recursively until all objects reachable by the application has been visited. Any non-visited object is then considered garbage, as there is no way for the application to reach it, and is therefore subject to be deallocated.

*Mark-sweep* garbage collectors are aptly after the two-step procedure they take to reclaim memory. Being a tracing collector, the object graph is traversed and every reached object is marked in some way, for example by updating some bit in the object or recording them in some data structure. When all reachable objects have been marked, the collector “sweeps away” all other objects by reclaiming their memory, leaving pools of free memory available for further allocation.

A *mark-compacting* garbage collector resembles that of mark-sweep, but attempts to remedy the problem of *memory fragmentation* – pools of free memory scattered throughout the heap. After performing the mark phase, instead of sweeping unmarked objects, marked (that is, *live*) objects are moved into a contiguous area of the memory – that is, the live objects are *compacted* together on the heap. This leaves a contiguous portion of free memory available for allocation, allowing for the use of bump allocation, thus greatly reducing allocation time.

Region-based garbage collectors do not process objects individually. Instead, the memory is partitioned into *regions*, usually holding multiple objects. The applied collection strategy is then performed by collecting regions of garbage rather than individual objects. As allocation and deallocation are costly operations, performing them only on larger regions, rather than for each object individually, can speed up the overall time spent on said operations.

*Parallel* collectors attempt to speed-up collection by performing GC-related work using multiple threads. Parallel collection, as with parallelism in general, is becoming increasingly popular as most modern computers constitute of multiple cores and other support for parallelisation at a hardware level, allowing for faster processing speeds.

### 2.2 The Z Garbage Collector

The Z Garbage Collector (ZGC) is a garbage collector implemented in the Java HotSpot Virtual Machine (HotSpot JVM) and included in OpenJDK (an open-sourced toolkit for Java developers) distributions since version 11. ZGC is a region-based, mark-compacting, parallel collector that performs GC-related work concurrently with the running application. The goals of ZGC is to keep pause times introduced by collection to a minimum (ZGC promises pause times of at most 10ms, independent of heap size), while supporting heap sizes ranging from very small to very large (in the range of terabytes), all the while staying competitive in throughput with the default G1 collector.
ZGC makes use of pages to partition the heap into regions. There are three types of pages: small, medium and large. Small pages are 2MB in size, and take all objects that have a size of at most 256KB (12.5% of 2MB). Medium pages’ size is calculated at runtime. It is always a multiple of 2 megabytes, and is at most 3.125% of the maximum heap size available for the application. If the calculated size is not larger than that of small pages, medium pages are disabled throughout, meaning only small pages and large pages are in use. (For example, a 1200MB maximum heap size gives medium pages size 32MB; a 200MB maximum heap size gives medium pages size 4MB; and a 100MB maximum heap size disables medium pages.) Medium pages take all objects that occupy at most 12.5% of their size (and are too big for small pages). Large pages adapt their size to that of the object, thus only holding one object. Any object too large to fit in a small or medium page will receive a large page. Each page tracks liveness information that indicates how much space of the page is occupied by live objects.

A global sequence number is used to track time in terms of a GC cycle counter. Every page is associated with the current sequence number at the time of its creation. This is used to differentiate between allocateable pages – pages on which objects may be allocated – and relocateable pages – pages on which the objects are candidates for relocation. A page is considered allocating if its associated sequence number matches the current global sequence number, and relocatable otherwise. As such, objects are only allocated to pages created since the last GC cycle.

ZGC uses a metaphor of colours to track the validity of pointers. Four bits of each pointer are used to store metadata. Three of these, which are called bits M0, M1 and R, together determine the colour of the pointer. The fourth bit specifies whether the object is reachable via a finalizer method. At any given time, one of the three colour bits are set, determining the colour of the pointer. While this gives way for three colours, ZGC is only interested in whether a colour is “good” or “bad”. At any time, a globally (that is, between all threads) agreed-upon colour is denoted good, and the others bad. We say that a good-coloured pointer is valid, and a bad-coloured pointer is invalid.

Much of the work carried out by ZGC is performed concurrently with application threads, or mutators, by a number of dedicated GC threads. To coordinate the GC threads’ work with that of the mutators, ZGC uses load barriers. A load barrier is a piece of code that runs prior to when an object reference is loaded from the heap to the stack and, depending on the validity of the pointer, either allows the access to continue or takes some action to update the pointer to a valid address prior to the access. The load barriers allow GC threads to move objects around in memory without informing the mutators, as any access to a moved object will be trapped – and fixed – by
a load barrier, before the access is carried out.

Having introduced load barriers and coloured pointers, the two core concepts that ZGC utilises, we now give an overview of what a collection cycle looks like in ZGC. A ZGC cycle consists of different phases interleaved with stop-the-world (STW) pauses. An overview of a ZGC cycle is given in Figure 1.

**STW Pause 1.** A ZGC cycle begins with a STW pause. During this pause, a new good colour is decided, altering between either setting bit M0 or bit M1 between cycles, and the global sequence number is incremented. Furthermore, all root references are dyed with good colour, and thus marked, and are added to a mark-stack for concurrent processing.

**Mark/Remap Phase.** The first phase is the Mark/Remap phase, which is performed concurrently with the running program. GC threads traverse the object graph, starting from the roots, and mark all reachable objects. Mutators help by marking non-marked objects, done as part of the load barrier when a bad-coloured pointer is loaded to the stack. Remapping refers to fixing pointers to objects that were relocated as part of the previous cycle (explained below). Whenever a pointer to a relocated object is found, its new address is looked up in a forwarding table that keeps track of how objects have moved from their old addresses to new. Marking an object also involves updating its associated page’s liveness information.

**STW Pause 2.** The second STW pause can be considered a synchronisation point, which is reached once the mark stacks are all empty. By now, we have updated liveness information for every relocatable page, meaning we know how fragmented each page is.

**Reference Processing Phase.** In this phase, the collector handles the various special reference types of Java such as weak references. This phase
is orthogonal to ThinGC and our work. For brevity, we will therefore not go into further detail on this phase.

**Reset Relocation Set Phase.** This phase merely resets the relocation set and the forwarding table constructed in the previous phase, both of which are explained below. This is done concurrently, although the GC threads do not interfere with the mutators here.

**Relocation Preparation Phase.** Recall that up-to-date liveness information had been obtained at the time of the second STW pause, and that no new objects could have been allocated to these pages as they are considered relocatable. In this phase, ZGC uses this information to select pages of which objects are to be relocated from. Pages with no live objects are reclaimed immediately. Other pages, with live objects, are added to a list and sorted by their *fragmentation factor* – how many bytes are considered live in relation to page size – giving an indication of how fragmented each page is. Pages that have a fragmentation factor higher than a certain fragmentation limit threshold are added to a *relocation set* (RS), and are thereby subject to having their objects relocated.

**STW Pause 3.** In the third STW pause, the good colour is changed to the R bit set. Pointers of this colour are guaranteed not to point to objects on pages in the RS. By changing the good colour, all pointers are effectively invalidated, meaning that they will be trapped by the load barrier. This ensures that any object selected for relocation will be relocated as part of the load barrier. Next, all root references are visited and relocated, were they selected, before being dyed good-coloured. Any reference to an object residing on an RS page is relocated prior to the recolouring, maintaining the invariant that pointers with the R bit set never point into a page in the RS.

**Relocation Phase.** The final phase tackles concurrent relocation. GC threads start relocating objects on pages in RS, page by page, reclaiming the pages whenever they empty. A forwarding table is used to map relocated objects’ old addresses to their new addresses. As the good colour was updated, any mutator accessing an object that is to be relocated will be trapped by the load barrier, upon which the mutator will help by relocating the object before validating the pointer and allowing the access. If the object has already been relocated, the load barrier will look up the object’s new address in the forwarding table and validate the pointer before allowing access.

Once all objects of all pages in RS have been relocated, and the pages in RS free’d, the relocation phase, and also the ZGC cycle, is finished. There might still be pointers to objects which have been relocated – these will be
fixed either by the load barrier upon mutator access or via remapping in the following cycle, before the forwarding table is reset.

2.3 Introducing ThinGC

ThinGC [4], for Thermal Insulation GC, is a garbage collector implemented as an extension to ZGC. ThinGC classifies objects as either hot or cold – that is, objects which have recently been read or written to, and objects which have not, respectively. ThinGC uses this classification to “offload” infrequently used objects (cold objects) to a separate part of the heap, called cold storage.

All objects are considered hot upon allocation, and are, as such, allocated in hot storage – the common work-storage of the application. When a mutator accesses an object, the object is considered hot. Marking an object as hot is done in the load barrier, similar to the handling of liveness. For this, ThinGC extends the metadata stored in pointers of ZGC with an extra bit, altering the colour scheme slightly. ThinGC extends the load barrier to move cold objects to hot storage on mutator accesses – a process called reheat. This behaviour maintains an invariant that mutators never directly accesses cold storage.

If a hot object has not been accessed for a period of time, it becomes cold and is thus subject to be moved to cold storage – done via a process called freezing. Freezing occurs as part of relocation. ThinGC selects pages for relocation following the logic of ZGC, but may optionally assign less weight in terms of bytes to cold objects when calculating live bytes. The number of live bytes on a page is used to select pages for relocation (densely populated pages are less likely to be selected for relocation as this work will recover less memory from garbage objects). When objects are relocated from a page, hot objects are relocated to a free page in hot storage, whereas cold objects are relocated to the cold storage. Thus, cold objects on pages not in the relocation set will remain in hot storage.

Objects in cold storage may hold references to objects in hot storage. If such a hot object would be relocated, the reference in the cold object would need to be updated to reflect this. However, as ThinGC avoids mutation in cold storage, all references from objects in cold storage to objects in hot storage are replaced with indexes into a remembered set – a mapping of object references and their up-to-date address – that resides in the hot storage. When an object in the hot storage moves, then the remembered set is updated, but the indexes into the remembered set from the cold storage can stay the same. Entries in the remembered set are roots when performing garbage collection in the hot storage.

To populate the remembered set, ThinGC adds another phase – the freeze-
patching phase – into the ZGC cycle, which comes after the relocation phase. Here, any reference to an object in hot storage from an object that was frozen is added into the remembered set. ThinGC refrains from adding any extra stop-the-world pauses. Figure 2 gives an overview of a collection cycle with the added phase.

By isolating cold storage and preventing direct access to objects therein, ThinGC allows the cold storage to be managed separately from that of hot storage. In particular, ThinGC enables the use of separate collectors for hot and cold storage. To manage objects in the cold storage, ThinGC introduces a separate collector – the Cold Storage Garbage Collector (CSGC). CSGC runs concurrently with ZGC, on a dedicated thread. Besides reclaiming memory in cold storage, CSGC also manages the remembered set, keeping indices up to date and reclaiming outdated entries. Notably, running garbage collection in the cold storage is optional, meaning it is possible to run ThinGC with a continuously growing cold storage, for example if backed on a large multi-terabyte medium like an SSD drive.

3 Avoiding Out-Of-Memory Errors

In this section we identify the source of the OOM errors produced in ThinGC. Specifically, we investigate why reheats sometimes fail, as this then leads to the OOM errors we are seeing. We discuss some mitigation strategies, both aimed at reducing how often reheats occur overall, and aimed at reducing the risk of a reheat failing, so as to avoid the OOM error otherwise produced.

3.1 Why OOM Errors Occur in ThinGC

Moving an object into hot storage, both as part of relocation (moving it from hot storage to hot storage) and reheating (moving it from cold storage to hot storage) requires that space is allocated on an allocating page of hot storage. Prior to our work, the steps taken to reheat or relocate an object were similar. An allocating page is held by the collector, on which objects-to-be-moved are allocated space regardless of the objects prior storage location.
If the allocating page is to full, a new page is requested, replacing the old allocating page, before the object can be moved successfully. This process will fail if the request for a new page is not granted, which can happen when the application is already using all its allotted memory, meaning no new pages can be created.

There is good reason to not share the same behaviour for relocation and reheating of an object, however. Should the process fail when performing a relocation, the collector can simply abort the relocation and leaving the object (and the rest of the objects on the page) in place. While this affects the collector’s ability to compact the heap, the program can still run, and the collector may hopefully defragment the heap and free garbage objects during the next compaction phase. Upon reheating an object, however, the object can not be left in place if the process fails, as this breaks the invariant that mutators never access objects residing in cold storage. Instead, the program is forced to exit, reporting OOM.

As an application’s required heap space approaches the maximum allowed heap space, the risk increases that all available pages are in use. Therefore, the reheating process fails more frequently as the maximum heap size is lowered. Furthermore, the collection strategy of ZGC, shared with any tracing collector, can cause an inflation on how much memory is used, due to the latency of identifying and collecting pages for reuse.

To illustrate how big this inflation can be and give an idea of the heap size-requirements of ThinGC in relation to that of only using ZGC, we find that for a particular benchmark, ThinGC starts to consistently see failed runs below 700 MB heap, whereas ZGC succeeds consistently even at 100 MB heap (the exact details of this evaluation is given in §5).

Besides the impact of the program’s required heap-space compared to the maximum heap size allowed, different programs are differently prone to reheating objects. When observing how objects are reheated in ThinGC across various benchmarks in DaCapo (release 9.12-MR1) [2], Nyblom found that in some of the benchmarks objects are never or very rarely reheated, whereas in others, many objects are reheated, often multiple times [3]. This confirms that reheating tendencies varies between different applications. In programs that never reheat, the OOM errors produced by failing reheats cannot occur.

By adding extra measures to guarantee successful reheats more often, it might be possible to reduce the amount of OOM errors that are produced when running reheat-prone programs with small (relative to program requirements) heap sizes. We will proceed by discussing some strategies in doing this, as well as other measures that could possibly remedy the problem of OOM errors produced by reheats.
3.2 Avoiding OOM Errors by Freezing Less Objects

There are measures one can take if running into OOM errors due to reheat already in ThinGC. The most obvious one, but perhaps also the most effective, is to run the program with a larger maximum heap size, using the `-XX:MaxHeapSize` switch (present in HotSpot JVM). If increased enough, this solution is likely to avoid the OOM errors, as more pages are available. However, this may not be favourable or even possible, for example if the application is already allowed the maximum size physically available. If this is the case, we need to deploy some other strategy to resolve the problem.

There are two values in ThinGC – `MinColdAge` and `HotWindows` – that can be set via command line switches, which could be tweaked to possibly remedy some of the problem. `MinColdAge` dictates how long ago an object must have been created to be considered hot, counted in ZGC cycles. `HotWindows` dictates how long an object is to be considered hot upon mutator access, also counted in ZGC cycles. While these values could be set to reduce how many objects are frozen (say, by giving `MinColdAge` some very large value), and thereby also subject to be reheated, it is not the intended use of the switches, and is likely to remove the benefits of using ThinGC to begin with.

To allow densely populated pages with many cold objects to be selected for relocation (a prerequisite for freezing the cold objects), ThinGC allows assigning a weight to control the extent to which live cold bytes contribute to the live bytes on a page. The weight can be controlled by a command-line switch called `ColdAsDead` which takes a value between 0 and 1, inclusive, as the factor of which cold bytes count as dead bytes when calculating the liveness information for the page they reside on. A value of 0 means that cold objects are counted equal to hot objects and a value of 1 means that cold objects are not counted at all. As objects only are frozen when their pages are selected for relocation, setting `ColdAsDead` to a lower value can result in less objects freezing. This then effectively means the program will likely see less re-heats – reducing the risk of an OOM error to occur.

If a user is running their program with a `ColdAsDead`-value larger than 0 (its default value), decreasing it could remedy the problem, as pages with many cold objects are then less likely to be selected into the relocation set. However, the selected `ColdAsDead`-value might have been beneficial for the application at hand (say, in regards to execution speed), making this solution suboptimal.

In general, while there might be good reasons to tune these values, their intended use is not for avoiding OOM errors. By tweaking the switches provided in ThinGC solely for the purpose of getting rid of errors, one may likely remove some of the benefits of using ThinGC in the first place, making this approach undesirable.
3.3 Avoiding OOM Errors by Reserving Pages for Reheats

A relatively straightforward way of avoiding failing reheats due to OOM is to introduce a reserve of pages that can be used as a “backup” when the reheating process fails. At first, this might seem contradicting, as running out of pages was the reason reheats failed. However, by reserving some pages for reheats, the rest of the pages (left for general use) can be managed more thoroughly (for example by triggering more frequent GC cycles when those pages become full). Furthermore, we already noted how failed relocations are not vital in the sense that it forces the application to crash, meaning trading off failed relocations for successful reheats can be considered worthwhile.

Assuming the reserve is big enough – that is, enough pages are reserved – to cover all reheats following the point where no more pages are available, until the end of the ZGC cycle, all OOM errors otherwise raised are successfully avoided. On the downside, a page reserved specifically for reheating is a page that cannot be used for other tasks, such as relocation or general memory allocation. Thus, placing too many pages in a reheat reserve can impact how successful the GC is able to perform compaction, for example. Furthermore, it can force more frequent collection cycles, as less memory is available to the application.

It is by no means an easy task to find a reserve capacity that is big enough to cover any otherwise failing reheats but not too big to have a negative impact on other GC tasks. What is clear is that there is no one-size-fits-all solution, given that the reheating tendencies, and also the required heap space, varies between applications. To remedy this, an easy solution is to give the user control over the reserve size, for example via command-line switches. Another approach would be to have the reserve alter its size dynamically at runtime, following some observed trend of objects’ behaviour. We discuss the latter approach in §3.3.1.

As a first step in dealing with the problem of OOM errors due to reheat, we propose a design using two reserves – one with small pages and one with medium pages, to account for any object size (recall that large objects are never frozen and thus never reheated). Each reserve holds up to a user-specified capacity of pages of their respective types, the reserves themselves possibly of different capacities. The reserves are replenished at the start of each GC cycle. These pages are then used to reheat objects onto as needed, whenever the reheat process fails to get a new page by normal means. If there are unused pages left once the relocation phase ends, they will be saved for possible use during forthcoming cycles.

We describe an implementation of this design in §4. It is difficult to foresee the ramifications of this design, given what we already discussed about possibly affecting other aspects of GC negatively when reserving pages for a
sole purpose. Ultimately, our goal is to successfully avoid OOM errors, but we note that we risk doing this as a trade-off on some other properties. In §5 we evaluate our design studying how successful it is in avoiding OOM errors, and also study if it introduces significant performance overhead compared to that of ThinGC prior to our work.

### 3.3.1 Scaling Reserve Capacities Dynamically

While carefully chosen reserve capacities might prove to be successful in remedying the problem, the burden of choosing such capacities lies on the user, and might only be found by a trial and error procedure. A very welcomed addition to the reserve-strategy would be to have the reserves adapt their capacities dynamically during program runtime. By scaling up the reserve capacities when the programs sees many reheats, for example, and scaling them down when reheats are less prominent (allowing for the pages to be used for other purposes) we could successfully avoid failed reheats without claiming away too much memory from the rest of the application.

The existence of an ideal scaling strategy, derived from some well-studied pattern, is not known. Plots from Nyblom [5], for example, shows varying patterns, or lack there-of, between programs’ freezing/reheating tendencies. A perhaps best settle would be if some heuristic could be found. Finding such a heuristic, dictating how the reserve capacities should be scaled, however, is not trivial. Ideally, the heuristic should work well for many different types of programs, and their varying freezing/reheating patterns. We shall discuss two candidate strategies below, that were considered during our work.

**Counting used-up reserve pages.** A candidate for a heuristic-based approach in scaling the reserve capacities involved counting how many pages of the reserve are used up during every cycle. The hypothesis was that the amount of used-up pages would gradually increase or decrease per cycle. Empirical studies showed that this is not the case – some programs showed tendencies of not using the reserve at all to using most or all of it in between cycles, in a seemingly unpredictable manner. Furthermore, the tendencies were inconsistent between runs. As we failed to find empirical data strengthening the hypothesis, we conclude that this approach might not be suitable, unless justified by more sophisticated studies.

**Counting frozen objects.** Another candidate is that of scaling the reserve capacities based how many objects there are currently residing in cold storage. By observing how many objects are currently in cold storage, one could potentially foresee possible spikes in the amount of reheats, and scale up the reserve capacities accordingly. This is particularly true for the op-
posite case – if none or very few objects are residing in cold storage, we
know that only few reheats can occur, and we can scale down the reserve
capacities to only match the potential need. The biggest caveat with this
strategy is that many objects will likely reside in cold storage without neces-
sarily being reheated any time soon (in fact, this is key for ThinGC to be
effective), making it difficult to scale the capacities accordingly.

It might be the case that the small and the medium reserve could benefit
from different strategies. In a large subset of the benchmarks found in
the DaCapo-9.12-MR1 batch [2], we find that less than 0.001% of objects
are of medium size [5]. As such, we could, for example, expect a more
manageable amount of medium objects residing in cold storage at any given
point, advocating the use of the strategy of counting frozen objects to scale
the medium reserve capacity.

As of our work, we have been unsuccessful in finding a strategy for scaling
the reserve capacities refined enough to justify an implementation, and it is
therefore left as possible future development.

4 Implementation

We implement our design on-top of the ThinGC code base[1] which is im-
plemented on-top of OpenJDK. Our additions are mostlyanchored in the
ZGC directory.

4.1 Object Allocation

The object allocator module in ThinGC (and ZGC) is responsible for allo-
cations of objects onto pages of the hot storage. This includes allocation
of objects due to both relocations and reheats. Previously, the process of
allocating space for an object in hot storage was shared between both re-
location and reheat, at least when carried out by mutators (relocations are
performed both by GC threads and mutators, whereas reheats are performed
exclusively by mutators). We already discussed that this might not be ideal,
seeing as the consequences of a failed reheat is more severe than that of a
failed relocation. To remedy this, we keep the old process intact, but add
a back-up procedure which is only used for reheats, that uses a reserve of
pages to which the reheat can be carried out, hopefully saving the program
from an OOM error.

Before explaining how the reserves are introduced, we give an overview of
the steps taken by the object allocator when performing a reheat or relo-
cation. For mutators, the object allocator holds a shared small page on a

[1] The ID of the commit we started on is 6530c5f21dd1053efc429136d8165c10abcfc20c.
per-CPU basis meaning threads on different CPU’s have access to different pages. It also holds one medium-sized page that is shared for all mutators independent of what CPU they are running on. (There are no large pages as they are not subject to neither relocation nor reheat.) It is to these pages that objects-to-be-moved end up.

Multiple threads could be allocating objects as part of relocation and reheating to the same page concurrently. To allow for this, the actual allocation of an object onto a page is made using atomic operations. If the page that a mutator is allocating to cannot fit an object, the mutator will request a new page. The request is sent to the page allocator, that returns an unused allocating page to the mutator, assuming there are such pages left. The mutator proceeds by allocating its object to the new page, before attempting to replace the old shared page with this new page. For this to work concurrently, a compare-and-swap (CAS) operation is used. It might be that another mutator has already replaced the shared page, whereby the CAS will fail. The mutator will then attempt the allocation to that page instead, releasing its requested page back to the page allocator upon success. This procedure is repeated either until the object allocation succeeds, or when the page allocator cannot grant a new page upon request.

If the mutator fails to allocate an object, and subsequently fails to obtain a new page, the relocation or reheat performed is unsuccessful. If a reheat is performed, this is where the reserve comes in handy. We extend the object allocator to also hold a list of small pages and a list of medium pages – the reserves. When a mutator fails to get a new page whilst performing a reheat, it will attempt to allocate the object to the first page of the reserve of the respective size. Should the object not fit on the first page, the page is dropped from the reserve and the next page is used, assuming there are pages left. We note that dropping the first page whenever an object does not fit might end up wasting a few bytes of memory (specifically 12.5% of the page size) that could fit a smaller object reheated later on during the cycle. We justify this, however, with the decreased complexity of using the reserve (as mutators need not search through multiple pages for a page that fits the object) and adheres to how pages are dropped in other parts of the object allocator.

The reserves are filled with pages according to their capacities, as set by the user (see §4.2). This occurs during the first STW pause of each ZGC cycle. If medium pages are disabled, the medium reserve will not reserve any pages despite a capacity larger than 0. Ideally, the reserves would claim pages after the relocation phase ends, as this is the moment where pages

\(^{2}\)However, if having one page per CPU takes up a significant amount of the maximum heap space, only one small page is shared between all mutators instead.

\(^{3}\)This page is also used by GC threads performing relocation.
have been reclaimed. However, due to the lack of a synchronisation point at the end of a cycle, something we refrain from adding following ThinGC’s goal of not adding large overhead, we delay this procedure to the first STW pause. We hope that enough pages remain available here, prior to being used for further allocation and relocation during the forthcoming cycle.

To avoid concurrency issues when multiple mutators need to use a reserve page at the same time, we protect the reserve with a lock. There are two separate locks for each reserve, and the locked portion is limited only to the allocation of an object to the first page of the reserve, and, if needed, the removal of this page following a new allocation attempt. It is worth noting that all mutators are not necessarily blocked waiting for the lock when the small reserve is being used. As small pages are held on a per-CPU basis, mutators running on a particular CPU might still have available space on a shared page, in which case they can safely reheat to that page without acquiring the lock. Thus, only mutators in need of the reserve are blocked, as opposed to all mutators performing reheat.

We argue against attempting a lock-free alternative until it has been shown that the lock itself is a bottleneck in performance, something which we have not studied further. It is noteworthy that simplicity and maintainability of code is key on a code base such as this, properties we mean the lock-based solution holds.

The object allocator drops all references to shared pages at the start of each ZGC cycle, as the pages are now considered relocating and are therefore subject to garbage collection (meaning allocating to such pages throughout the following cycle would be illegal). It is often the case that many pages of the reserves have not been used during the previous cycle. To avoid dropping these page references, only to fill up the reserves shortly thereafter – when less pages might be available – we instead reset the pages, setting their sequence number to match that of the global. This allows them to still be considered allocating pages and used throughout the forthcoming cycle.

4.2 Allowing the User to Tune Reserve Capacities

To allow the user to tune the reserve capacities to their needs, we introduce two command-line switches that sets the capacity of the small and medium reserve respectively. These can be enabled upon initiating the JVM using switches -XX:ReheatSmallReserveCapacity=cs and -XX:ReheatMediumReserveCapacity=cm, where cs and cm are the desired capacities of the small and medium reserve respectively.

If the user does not specify either, or both, switches upon initiating the JVM, the reserves default to a capacity of 0, thus mimicking the behaviour of ThinGC prior to our work. The rationale behind this is that if a program
is initiated with a big-enough heap size, the reserves are not needed and can only incur an overhead on program runtime and memory usage. Instead, the user is left to enable the reserves only as needed.

5 Evaluation

All benchmarking was done on an Intel® Core™ i7-7500U CPU @ 2.70GHz with 2 cores (2 hyper-threads per core), 12GB RAM, 32KB L1, 256KB L2, 4MB L3, running Ubuntu 18.04.3 LTS with Linux kernel version 5.0.0. The ThinGC commit we implemented on top of has ID 6530c5f21dd1053efc4291-36d8165c10abcfc20c, and we built using GCC 7.5.0 as the C/C++ compiler.

5.1 Avoiding Out-Of-Memory Errors

The main goal of our work is to allow ThinGC to be utilised without requiring an excessively large heap size compared to the actual demand of the application. In order to evaluate whether our implementation is successful in this regard, we shall study its effectiveness on four benchmarks – h2, xalan, pmd, and lusearch-fix – of the DaCapo-9.12-MR1 batch [2]. We previously discussed how different programs show varying freezing/reheating tendencies. The benchmarks that we have selected are ones identified to have a high reheating percentage [4]. It is for these types of programs that we believe our added reserves will have a high impact.

Each of the four benchmarks were run with three garbage collection configurations; ZGC with ThinGC disabled (and thereby also our reserves disabled), built on our final commit; ThinGC prior to our additions, built on the commit we implemented on top of; and ThinGC with our added reserves, built on our final commit (we call this configuration ThinGC+RS). With these configurations, not only will we be able to observe how successful ThinGC+RS is in avoiding OOM errors compared to that of ThinGC prior to our work, but also give us an indication of how closely ThinGC+RS matches ZGC in regards to how much heap size applications require.

For each benchmark, for each configuration, we performed 20 runs each on a set of heap sizes ranging from sizes that ThinGC handles (that is, does not crash on) down to lower sizes handled only by ZGC. Each benchmark is run using the large input size.

We set HotCycles (dictating how many cycles objects must live to be considered hot) to 2 throughout (ThinGC requires this to be specified, as it otherwise defaults to 0), to not have it be a varying factor on the success rates, and left other values to their defaults. We selected reserve capacities individually for each benchmark, shown to be successful based on brief empirical trials. Specifically, h2 uses a small reserve capacity of 8 and a
Figure 3: The amount of runs that passed benchmarking without failing due to OOM errors (there were no other causes of failure), for each of the four selected benchmarks. Note that OOM errors received when running ZGC are caused by other means than reheats.

medium capacity of 2, whereas all others use a small capacity of 4 and medium capacity of 1.

We present our evaluations in Figures 3a–3d. The figures illustrate how many runs were successful – that is, did not crash due to OOM – out of the 20 runs, for each given heap size, for each of the three configurations.

In all of the selected benchmarks, we can see that our reserve build outperforms that of ThinGC in regards to success rate as the heap size gets lower. As ThinGC starts seeing failed runs, ThinGC+RS holds up well and only starts seeing failed runs a few hundred megabytes further down. We take this to suggest that our strategy has been successful in avoiding failed reheats, and that enabling the reserves can indeed help programs seeing a lot of reheats to run even with limited heap sizes.

The evaluations also show that ThinGC+RS gets close to matching that of ZGC in regards to how small heap sizes it can support. Ideally, we want ThinGC to be applicable whenever ZGC is. However, we know that ThinGC will incur some overhead on memory usage for storing of internal data structures. Exactly how big this overhead is, and thus how closely
matched we can expect our reserve build to be to that of ZGC has not been calculated, and as such we draw no conclusion on how well this is managed. Overall, however, we find that ThinGC+RS performs satisfactory in this regard, and has at the very least taken a step in the right direction.

5.2 Performance Overhead

A secondary goal of our implementation was to keep runtime performance overhead low, especially when the reserve capacities are set to 0, which was the rationale for our defaults. To study this, we shall compare the average execution times for a set of selected reserve capacities of ThinGC+RS, and compare these to that of ThinGC. For every configuration, we run the h2 benchmark (selected as it has the longest running time of the benchmarks studied) 20 times, noting their execution time as output by the benchmark, and calculate the average. All benchmarks are done with 1200 MB heap size, a size at which we saw ThinGC passed all 20 runs in Figure 3b. The results are presented in Figure 4.

Our intentions with this comparison is to study how our reserves affect performance relative to ThinGC. Especially, we want to confirm that the reserves does not incur significant overhead when not used, as we wanted our build to behave similarly to ThinGC when the capacities were set to 0. Furthermore, we want to study the impact of poorly chosen capacities. We note that these studies do not show how much extra time is spent when the reserves are actually being used, as this would require studying the execution times running benchmarks with smaller heap sizes, of which ThinGC alone can not handle. We do not compare the execution time with unmodified ZGC, as the overhead added by ThinGC (excluding our reserves) is not known.

The results presented in Figure 4 confirms that our implementation does not
incur any noticeable overhead when the reserve capacities are set to 0. This result is reassuring, as it indicates that our changes need not incur penalties even if introduced in applications where ThinGC was already being utilised successfully.

We can also see that increasing the capacities slightly does not seem to add any noticeable overhead. This could mean that larger default capacities than 0 would be a viable fit. However, having fixed non-zero numbers would likely not be ideal, as their combined memory usage would not be proportional to the application requirements (say, the selected maximum heap size). Setting default values as a percentage of the maximum heap size, for instance, is an interesting idea which we leave as future work.

With the small capacity set to 64 and the medium capacity set to 16, meaning that roughly half of the 1200MB heap is reserved, we notice a considerable increase in execution time. This confirms that poorly set capacities can greatly impact runtime performance. It is worth noting that this does not necessarily imply that our implementation itself is slow. It is likely that the increased memory usage prompts more GC cycles, which can be a big factor. To verify this, we re-ran the two configurations with GC logging information turned on. We saw that ThinGC alone went through 18 GC cycles, while on our build, with a small capacity of 64 and a medium capacity of 16, 57 GC cycles were performed, confirming our suspicion.

6 Related Work

As our work consisted of fixing an erroneous behaviour in ThinGC, we shall review some work similar to ThinGC itself. We further discuss whether or not these works are likely to exhibit problems similar to that of failed reheats.

Akram et al. [1] shows how to make practical the use of a combination of both DRAM and Non-Volatile Memory (NVM) for main memory. They extend GenImmix, a generational mark-region garbage collector, with write-rationing mechanics to move objects between DRAM and NVM. Specifically, their Kingsguard-writers (KG-W) monitor objects’ writes and moves them between DRAM and NVM, decided on an individual basis. This is reminiscent of ThinGC’s hot- and coldness classification and its partition into hot and cold storage. In KG-W, only writes trigger object relocation whereas ThinGC also considers reads a criteria for relocation. Furthermore, KG-W does not completely isolate NVM by moving objects to DRAM upon writes, as opposed to ThinGC’s invariant that cold storage never sees mutation from mutators. They build on Jikes Research VM, while ThinGC builds on HotSpot JVM. ThinGC is concurrent, whereas KG-W is not.
KG-W allows access to objects residing in NVM, as opposed to ThinGC disallowing access in cold storage. Furthermore, objects are moved from NVM to DRAM in a more controlled manner, as opposed to in ThinGC where reheats force a relocation and can occur at any time. As such, it is unlikely for KG-W to present similar issues as the problem of failed reheats in ThinGC. This is further strengthened by the fact that GenInmix, the collector used in KG-W, is not concurrent, meaning it could trigger a collection cycle when memory is running low, before letting mutator threads continue.

Bond and McKinley [3] introduce a design similar to that of ThinGC in their work of tolerating memory leaks. Their approach, called Melt, isolates stale objects (objects that go unused for a while, reminiscent of cold objects) from in-use objects by moving them to secondary storage. As such, they are not included in the GC working set. Similarly to reheats, Melt activates stale objects upon access, moving them back into main memory, and thus disallows accesses to stale objects in secondary storage. Melt is built on Jikes Research VM, using a STW collector, as opposed to the concurrent ZGC which ThinGC builds on top of on HotSpot JVM.

Given the similarities between stale (in Melt) and cold (in ThinGC) objects, it could be the case that Melt faces issues similar to that of failed reheats when activating objects. A key difference is that Melt keeps space proportional to in-use memory, unlike the sporadic inflation we see in ThinGC. Furthermore, as Melt runs a STW collector, activates can trigger a collection cycle if memory is running low. Although it is noted that this cycle is deferred until a following safe-point, it is likely triggered early enough to avoid running out of memory unless the application actually uses more memory than is available.

7 Conclusion

In this thesis, we set out to reduce the amount of Out-Of-Memory errors produced by reheats in ThinGC. We determined that the errors were caused by the reheating procedure failing to receive an empty page, leaving the object still frozen. This effectively forced the program to crash as the object otherwise would need to be accessed directly in cold storage, breaking an invariant that ThinGC maintains to allow different both hardware- and software-designs for how the cold storage is managed without introducing much overhead.

A few mitigation strategies were proposed; both strategies aimed at reducing the likelihood that reheats occur, as was the case with tweaking of ThinGC’s switches; and aimed at ensuring that there are pages left for use by the reheating procedure, by introducing a reserve. We also proposed a possible improvement to the latter, by having the reserve scale its capacity dynam-
ically at runtime. We did not manage to find a scaling-strategy refined enough to justify an implementation, however, leaving it as a candidate for future work.

The reserve strategy was realised into an implementation using two reserves, that are used to supply the reheating procedure with allocation space when it fails to get pages by other means. We emphasised the importance of not introducing large complexity into an already complex code base, and of maintaining ThinGC’s goal of introducing only minimal overhead to ZGC.

Our implementation was evaluated and showed very promising results – we were successful in avoiding failing reheats in programs known to exhibit many reheats, even a few hundred megabytes below the point where ThinGC as it was prior to our changes started seeing OOM errors. Furthermore, we bridged the gap of programs’ minimum required heap size to that of ZGC.

The evaluation also showed that our implementation seemed to not introduce any noticeable overhead in regards to performance, particularly when the reserve capacities are set to 0, but also when they are slightly larger. However, we also saw an example of how poorly selected capacities can greatly reduce performance. It is evident that our strategy puts the burden on the user to select sane capacities, and that our implementation is lacking in that there is no obvious way of determining these besides empirical studies. Automatically inferring default capacities, for example by reserving a certain percentage of the heap size, is an interesting idea, which together with our additions could make ThinGC work better “out of the box”.

References


