ThinGC: Complete Isolation with Marginal Overhead

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Erik Österlund
Oracle
Sweden
erik.osterlund@oracle.com

Jesper Wilhelmsson
Oracle
Sweden
jesper.wilhelmsson@oracle.com

Hanna Nyblom
KTH Royal Institute of Technology
Sweden
hnyblom@kth.se

Tobias Wrigstad
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Abstract

Previous works on leak-tolerating GC and write-rationing GC show that most reads/writes in an application are concentrated to a small number of objects. This suggests that many applications enjoy a clear and stable clustering of hot (recently read and/or written) and cold (the inverse of hot) objects. These results have been shown in the context of Jikes RVM, for stop-the-world collectors. This paper explores a similar design for a concurrent collector in the context of OpenJDK, plus a separate collector to manage cold objects in their own subheap. We evaluate the design and implementation of ThinGC using algorithms from JGraphT and the DaCapo suite. The results show that ThinGC considers fewer objects cold than previous work, and maintaining separate subheaps of hot and cold objects induces marginal overhead for most benchmarks except one, where large slowdown due to excessive reheats is observed.

Keywords: read/write rationing GC, ZGC, heap partition, hot/cold classification

1 Introduction

Previous work on leak-tolerating GC [5] shows how rarely accessed objects can be offloaded from DRAM to disk to delay (or even survive) out-of-memory error and shrink GC working set. Later work on write-rationing GC [1] explores this idea further in a heterogeneous system with both DRAM and Non-Volatile Memory (NVM) to maximize NVM usage while retaining original performance (NVM is slower) and longer NVM lifetime (when NVM degrades on writes). Both works suggest that many applications enjoy a clear and stable clustering of "hot" (recently read and/or written) and "cold" (the inverse of hot) objects, which could be leveraged using stop-the-world collectors in Jikes RVM.

In this paper, we use a concurrent collector in the context of OpenJDK and divide the heap into two parts, hot and cold subheaps, accommodating hot/cold objects, respectively. This design paves the way for special treatment of the cold subheap, such as backing it on slower memory, or memory that degrades quicker than DRAM, or running programs which require a heap size larger than available DRAM (a well-known "no-no" [15]). We call our collector ThinGC (*Thermal Insulation GC*). We augment the load barrier from ZGC to track objects hotness, and try to offload cold objects from hot subheap to cold subheap. Objects in the cold subheap are never directly accessed by mutators; instead, loading of a pointer to such an object is trapped by the load barrier which triggers a *reheat* that moves the object into hot subheap. These design decisions are very close to Melt [5]. In a further extension to prior work, we manage the cold storage via a *separate* tracing collector that supports (optional) *compaction*.

This paper makes the following contributions:

- We propose a concurrent write-rationing and read-rationing GC design based on a simple hot–cold object classification (§ 3) as an extension of ZGC (§ 2). Our design does not require special OS or hardware support.
- We provide an implementation of ThinGC in OpenJDK.
- We validate our design choices using algorithms from the JGraphT library and the DaCapo suite, and find marginal overhead for most benchmarks. (§ 4).

2 A ZGC Primer

ZGC is a non-generational, mostly concurrent, parallel, mark-compact GC algorithm implemented in HotSpot JVM for 64-bit architectures, included since OpenJDK 11 as an experimental feature. ZGC departs significantly from previous GC algorithms in HotSpot by utilizing load barriers instead of write barriers in order to facilitate concurrent compaction. During compaction, GC threads may move an object without updating its incoming pointers, which then effectively become dangling. Mutators on accessing them will be trapped, and load barrier will look up the new location of the object for access. To keep track of which pointers might be dangling, ZGC uses higher-order bits in pointers. Each pointer has four "meta bits": Finalizable (F), Remapped (R), Marked1 (M1), and Marked0 (M0). The F bit is related to objects with finalizers, whose implementation is orthogonal

ISMM '20, June 16, 2020, London, UK
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to this work, so we omit its discussion, while all other meta bits will be elaborated as we go through all phases in ZGC. In this paper, we follow ZGC parlance and talk about metadata as "colors". ZGC essentially uses two colors: *good* (pointer is valid) and *bad* (pointer is potentially invalid). A pointer's color is determined by the status of the meta bits: F, R, M1, and M0. A good color is represented as one of R, M1, M0 meta bits set and the other three unset, which gives three possibilities of good colors. There is always a single, globally agreed upon, good color from the three possibilities, set in a STW pause. Once the good color is decided, all other colors are considered "bad". Object creation always yields a pointer with the current good color.

A load barrier is code executed when a pointer is loaded from the heap onto the stack, *e.g.*, when accessing $x.f$ (assuming f is not of primitive type); the load barrier is called on $x.f$, not x, because x has already gone through the load barrier when it's loaded onto the stack. If the loaded pointer has good color, the fast path of the load barrier is taken, which is effectively empty, otherwise the slow path. The slow path calculates the corresponding pointer with good color, and performs *self-healing*, meaning it replaces the old pointer with the new and good-colored pointer, so that subsequent accesses take the fast path. Regardless which path is taken, a pointer with good color is returned. The logic in mark barrier is more or less the same with load barrier except that mark barrier is used by GC threads during marking, while load barrier is used by mutators.

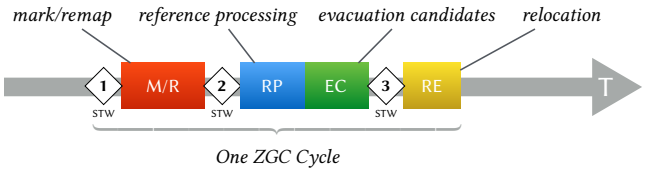


Figure 1. STW pauses and concurrent phases in ZGC.

We now continue by describing the STW pauses and concurrent phases of ZGC in left-to-right order of Fig. 1.

STW1: The Start of the ZGC Cycle. The start of the ZGC cycle is a STW pause, which performs four major tasks: decide on the good color (M0 or M1 bit set by alternating selection); mutators' thread-local allocation buffers are "flushed" so that newly allocated objects from now will be placed in fresh pages; all roots, after being tinted with good color, are pushed to mark stacks (for parallel processing by GC threads); and the GC sequence number is incremented (grows monotonically). The current GC sequence number is used as the birth cycle when a page is created, as shown in Fig. 2.

If the current GC cycle is N, pages of birth cycle N are called *allocating* pages (mutators are potentially allocating objects on those pages). ZGC treats all objects on allocating pages alive. In other words, ZGC only collects garbage objects on pages created prior to the current STW1, birth cycle smaller than N; those pages are called *relocatable* pages, since objects on them may be relocated as part of compaction.

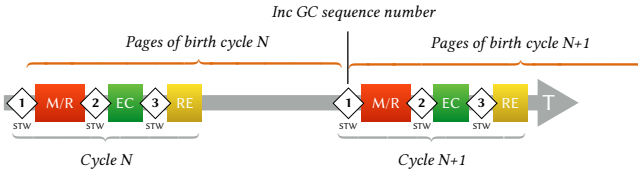


Figure 2. GC sequence number is incremented in STW1, and its current value determines the birth cycle of a page.

Marking/Remapping (M/R). As the name implies, two main tasks are performed in this phase, marking and remapping. Remapping deals with invalid pointers due to relocated objects from previous GC cycle, which we will cover later, so we only focus on marking here. GC threads consume the mark stack, and try to mark the popped objects. Marking can fail due to concurrent execution; when it succeeds, the current thread will update the liveness info of the associated page. The liveness info is the number of live bytes on a page and is used to select pages on which objects will be evicted; more is covered later. Additionally, all children (fields of reference type of the just-marked object) are pushed to the mark stack.

STW2. The marking phase terminates when mark stacks become empty.[1] With M/R phase over, we know the liveness info for all *relocatable* pages.

Reference Processing (RP). The reference processing phase handles Java's soft, weak and phantom references. This phase is not affected by this work (nor this work by it), and we thus omit it from the paper.

Selection of Evacuation Candidates (EC). The evacuation candidates is a collection of sparsely populated relocatable pages. By relocating all *live* objects on the EC pages into other pages, all EC pages can be reclaimed. The objective of this phase is to construct such evacuation candidates. *Relocatable* pages with live objects are added to a candidates list, and pages with zero live objects are reclaimed right away. The candidates list is sorted by live bytes. Pages from the sorted candidates list will be added to the evacuation candidates set subject to the fragmentation limit threshold.

STW3. In STW3, the good color is changed to the R bit on. A pointer of this color is guaranteed to not point to objects on pages in EC. This change of good color effectively invalidates all pointers in the heap causes mutators to take the slow path on the *first subsequent load*. STW3 visits roots

[1]There are other tasks performed here, which calls for a STW pause, but they are omitted due to little connection to this work.

and relocates all roots pointing to EC, and otherwise just changes its color to the good color. In the end, the roots will have good color. This establishes the invariant that mutators never see pointers into EC.

Relocation (RE). After STW3, the system is ready to perform concurrent relocation. This happens by GC threads migrating all live objects in EC, page by page. Forwarding tables are used to record old-to-new address mapping. Forwarding tables live outside of each page so that relocated pages could be reclaimed right away. Mutators that access an object concurrent with its relocation will hit the slow path of the load barrier, as the pointer's color will invariably be bad. In this case, mutators help the GC thread perform the relocation, potentially competing with other relocating threads. This is captured at a linearization point in the forwarding table. Threads that fail the race will read the new address from the forwarding table. Once such an entry exists, mutators hitting the slow path only need to query the forwarding table, skipping the relocation (copying).

Once all objects in EC are relocated, the ZGC cycle terminates, but there may still be pointers in the heap pointing to objects which have been moved. Those will be fixed either by the next mutator access or GC threads performing remapping in the M/R phase of the next ZGC cycle.

3 ThinGC

ThinGC divides memory into two separate spaces, hot and cold storage. Each space is managed by its own collector running independent of the other, and no collection goes through the entire heap. Reminiscent of cache hierarchies, the hot storage is occupied mostly by *hot* objects (recently accessed by the mutators), and the cold storage only contain *cold* objects (the inverse). Objects are allocated in hot storage and may be *frozen*, as part of relocation, into cold storage. There are no accesses—neither reads nor writes—to objects in cold storage from mutators. Instead, such accesses are trapped by the load barrier and cause the object to be *reheated*—copied into hot storage—before the access is performed. This maintains the invariant that mutators never see pointers into cold storage. An overview of the design of ThinGC is shown in Fig. 3. Apart from freezing, CSGC is the only source of mutation in cold storage (dominated by compaction which is optional).

This section describes how we extend ZGC to support hot and cold storage without additional STW pauses. As shown in Fig. 4, we add two new phases: *freeze-relocation*, which lies within original ZGC relocation phase, and *freeze-patching*, which occurs right after relocation.

3.1 Identifying Hot/Cold Objects

We consider an object hot if it meets one of the two conditions: a) the difference between its page birth cycle and the current GC cycle is less than N; b) it was accessed by a

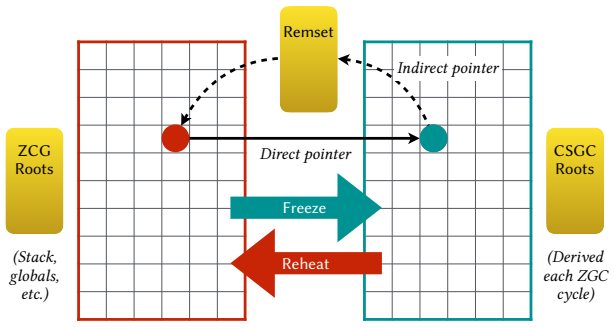


Figure 3. ThinGC overview. Hot storage (left) managed by ZGC and cold storage (right) managed by CSGC. Each space keeps its own rootset, allowing them to be GC'd individually.

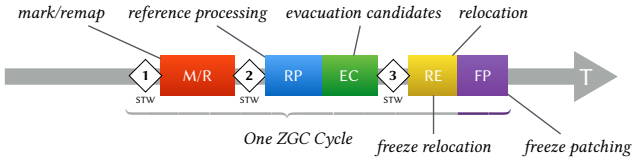


Figure 4. ThinGC extensions to ZGC.

mutator during the past W GC cycles. Otherwise, it is considered cold. N and W default to 2 and 1, respectively, and can be controlled by runtime flags (MinColdAge and HotWindows). Page birth cycle is set to be the current GC sequence number when the page is created. Since GC sequence number is only incremented in STW1 (see Fig. 2), $N = 2$ ensures that objects are *not* considered cold unless they live for at least one non-GC period, which filters out short-lived objects. A "non-GC period" is usually much longer than a GC cycle, as shown in Fig. 5. Thus, the default value of N could avoid prematurely freezing objects that will soon die.

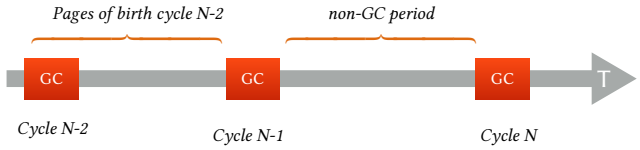


Figure 5. Objects need to survive for at least one non-GC period to be qualified for freezing. In Cycle N, objects on pages of birth cycle $N-2$, are eligible for freezing if they are not recently accessed by mutators.

We extend the ZGC load barrier to track whether an object was recently accessed; we mark objects as hot on the slow path of the load barrier (which will be hit on the first mutator access in each ZGC cycle), and set objects (not pointers) with the R bit on in their pointers as hot on the slow path of the mark barrier. Nothing is changed on the fast path. This means that an object is hot if it has been accessed by mutators somewhere between the starting of the previous relocation phase and the starting of the current relocation phase, as

shown in Fig. 6. W controls how many such windows we would like to accumulate before resetting the hotness info. In other words, this flag controls how often freezing happens.

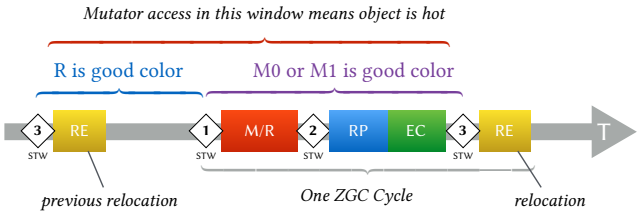


Figure 6. Hot window: mutator accesses from the start of the previous relocation phase to the start of the current relocation phase mark an object as hot.

Since objects accessed by mutators are only identified as hot on the slow path, GC threads may suppress hot marking by self-healing a good-color pointer *before* a mutator loads it. This may lead to missing some hot objects. To remedy this situation, we introduce another color in the mark barrier; instead of self-healing a pointer to good color, we self-heal it to this new color. This will cause mutators loading the pointer to hit the slow path and mark the object as hot. This way, a pointer with good color always indicates that the mutators recently accessed it, and no hot objects are missed.

We add a flag to dump all relocatable objects and their recorded hotness status every ZGC cycle.[2] By analyzing the logs, we can see how hotness status changes for an object, which can be used to answer questions like how often objects alternate between being hot and cold, the longest duration objects stay hot, etc. A property of particular interest is whether cold objects stay cold, which we call *inverse temporal locality*. The cold-to-hot transition (reheating) percentage as defined below reflects this property. We discuss our findings in the evaluation section.

$$\text{reheating percentage} = \frac{\text{total cold-to-hot transitions}}{\text{total number of relocatable objects}}$$

3.2 Revisiting Evacuation Candidates Selection

ZGC selects sparsely populated pages as Evacuation Candidates (EC) to maximize reclaimable memory and minimize copying overhead. Now that relocation bears an additional mission, we need to rethink how to construct EC. In addition to the sparsity criterion, a page that is mostly populated with cold objects could be a good choice of EC as well in ThinGC, as cold objects can be moved to cold storage, which releases memory in hot storage. In other words, cold objects are essentially the same as dead objects. We call pages added to EC because of large number of cold objects cold pages.

[2]In order to minimize interference with normal application behavior, this dump is done in a STW pause.

Cold objects on cold pages are not completely "free" as dead objects; actual copying still needs to be perform as they are frozen. To allow tuning this behavior, we expose a runtime flag (ColdAsDead) that assigns a $[0.0, 1.0]$ weight to cold objects in terms of their liveness, and set a hard cap on the number of cold pages in EC to prevent RE phase from growing too long. A ColdAsDead value of 1.0 means that all cold objects are counted as dead objects. The default value is 0.0 which preserves the original EC selection logic treating all cold objects as live objects.

3.3 Freezing Objects

Freezing an object o consists of two steps:

1. *freeze-relocation*: relocating o from hot to cold storage
2. *freeze-patching*: patching o's fields

In RE phase, hot objects are relocated within hot storage (standard ZGC behavior) and cold objects in hot storage are relocated to cold storage (*freeze-relocation*). Those frozen objects are added to the CSGC roots (explained in § 3.6 and important for independent GC in the cold storage) and will be further processed in *freeze-patching* step.

As objects in hot storage are subject to relocation (within hot storage or to cold storage), direct cold-to-hot pointers may become stale. To remedy this, we introduce an additional level of indirection for such pointers in *freeze-patching*.The actual patching to a pointer depends on what kind of object it points to. There are totally three possibilities: object is...

FP1 ...already in cold storage;
FP2 ...relocated into cold storage in the same phase;
FP3 ...in DRAM.

The first case does not need patching; in the second case, we patch the field by updating it to point to the object's new address in cold storage; in the last case, we save the original address in a remembered set (which we refer to as remset), and patch the field by updating it with the corresponding remset index (a more detailed explanation follows in § 3.5).

Fig. 7 shows freeze-patching: objects A, B, and C reside in hot storage, and object D' in cold storage. A and C are identified as cold and relocated into cold storage, denoted as A' and C'. This adds A' and C' to the CSGC roots. Subsequently, their fields are patched according to the three cases above. A's pointer to D' corresponds to FP1, A's pointer to C corresponds to FP2 and A's pointer to B corresponds to FP3.

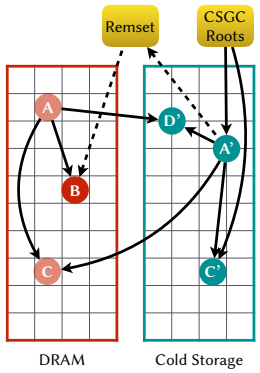


Figure 7. Overview of freeze-patching.

3.4 Reheat

ThinGC avoids mutating objects in cold storage. When a mutator loads a pointer pointing into cold storage, the frozen object it points is moved into the hot storage (keeping the invariant that a mutator only sees objects in hot storage). This requires that mutators hit the slow path of a load barrier for all cold pointers, which is achieved by introducing one more metadata bit in pointers.

Reheat is reminiscent of ZGC DRAM-to-DRAM relocation and uses the forwarding table. However, we do not record an object's final address in the forwarding table as is done in ZGC, because the address is unstable. Instead, the address after reheat is added to remset, and the returned index (more on this in § 3.5) is recorded in the forwarding table. This indirection is needed to keep the address in remset updated while avoiding remapping pointers in cold storage.

3.5 The Remembered Set (remset)

The ThinGC remembered set serves two main purposes. First, it is used to capture cold-to-hot pointers, which are treated as additional roots whenever ZGC runs, similar to many generational GC algorithms. Second, when an object is reheated, its fields pointing into the remset are patched with valid addresses from the remset, as if this object has never left hot storage. In order to fulfill these two purpose, Two invariants are maintained: all cold-to-hot pointers are recorded in remset, and these pointers always point to the updated location of an object. The latter is achieved by remapping each entry in remset after ZGC relocation.

The remset is populated in *freeze-patching* and reheating. During *freeze-patching*, if a field of a just frozen object holds a pointer into hot storage (FP3 in § 3.3), it is added to the remset. When an object is reheated, it is added to remset unconditionally, because it is possibly referenced by other objects in cold storage. In both cases, a new record is added to the remset (if it does not already exist), capturing pointers from cold to hot storage, maintaining the first remset invariant.

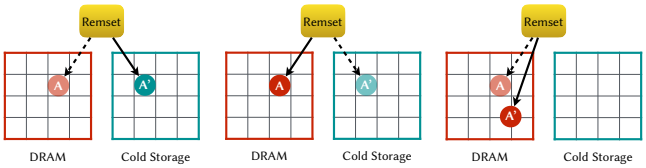
Entries in the remset are updated after the corresponding objects are relocated. There are three possible cases on the object relocation, explained below and shown in Fig. 8:

Hot → Cold remapping is performed eagerly, along with the relocation so that a reheat immediately following a freeze could still keep the address up to date (reusing the same array slot)

Cold → Hot *aka reheat*, remapping is performed eagerly, and the forwarding table is used to hold the corresponding index

Hot → Hot no update; instead, the entry is remapped at the beginning of a ZGC cycle, which is consistent with other roots of ZGC.

After an object is reheated, its fields, if pointing into remset, can be patched to the latest address by querying the remset, because of the second invariant.



(a) Hot → Cold **(b)** Cold → Hot **(c)** Hot → Hot

Figure 8. Remembered set update. Dashed and solid arrows denote state before and after ZGC relocation, respectively.

3.6 Cold Storage Garbage Collection (CSGC)

Objects in the cold storage are collected separately from those in the hot storage, which are managed by ZGC. This reduces the work spent in marking and relocation for ZGC. A CSGC cycle runs concurrently with the ZGC cycle and a single CSGC cycle can span multiple ZGC cycles. After ZGC marking, a new CSGC cycle starts unless there is already an active CSGC cycle. CSGC runs on a dedicated thread. It publishes its status after finishing a complete cycle and waits for the next invocation. The goals of CSGC are:

1. Identify unused remset slots for reclamation to reduce the remset footprint and remove ZGC roots.
2. Remap pointers to reheated objects in cold storage.
3. Reclaim memory in cold storage (see below).

Marking in CSGC starts with a root set (explained shortly) and proceeds recursively through the objects in cold storage and stops on the hot–cold boundary, marking the corresponding remset slot reachable which makes it a root in the subsequent ZGC cycle. While marking is in progress, a mutator could trigger a reheat, and alter the object graph. To preserve the reachability of remset slots, despite concurrent reheating, we use Yuasa's snapshot-at-the-beginning write barrier technique [20]. This ensures that all entries in remset reachable in the beginning of a ZGC cycle will be marked even with concurrent reheating. In addition to marking, CSGC performs remapping of reheated objects, from address in cold storage to remset index. *These are the only writes to cold storage performed by CSGC after freezing.*

CSGC roots. CSGC roots are all hot-to-cold pointers and they are discovered in ZGC during marking. In STW2, right after M/R phase (recall Fig. 1), such invariant that all pointers from hot to cold storage are captured in the CSGC roots is established. Now a new CSGC cycle can be started (unless one is already running).

Compaction in the Cold Storage. CSGC compaction borrows many ideas from ZGC compaction: candidates for evacuation are identified based on liveness info, adding them

into EC, and relocating objects in EC. CSGC compaction moves objects around in cold storage, mirroring normal ZGC relocation. A forwarding table is used, which is the same forwarding table used for reheat. In other words, we use the forwarding table as the linearization point to determine if an object is moved to hot storage (in case of a reheat) or moved to another location in the cold storage (in case of compaction). After CSGC compaction is done, all pointers, both in hot and cold storage, into CSGC EC need to be updated, which will happen in the next ZGC M/R phase, and CSGC M/R phase. CSGC compaction can be turned on or off with a flag, CSCompaction, and is off by default.

3.7 Cycles Across Hot/Cold Boundary

ThinGC does not have memory leaks from cycles across the hot–cold boundary, regardless of whether the hot and cold storages are collected separately or not (*i.e.,* cold storage simply grows). Consider an object in hot storage that is kept alive only by pointers from cold storage. As such an object cannot be accessed by mutators, it is effectively cold and will be frozen (moved to cold storage) during next relocation if the page it resides on is selected for relocation. Thus, any cycle will now be in cold storage only.

3.8 Known Limitations

For brevity, these are discussed in Appendix A.

4 Evaluation

All benchmarking is done on an Intel® Core™ i7-4600U CPU @ 2.10GHz with 2 cores (2 hyper-threads/core), 12GB RAM, 32KB L1, 256KB L2, 4MB L3, running Debian 11 (bullseye) with Linux kernel version 5.4.19 and GCC 9.2.1. The OpenJDK commit we build ThinGC on is authored on 2020-02-06.

In order to explore the effect of various runtime flags and how they interact with each other, we run all benchmarks in 9 configurations as shown in Table 1. Config 0 is our baseline, unmodified ZGC from the OpenJDK commit on which we build ThinGC; config 1 is built from ThinGC code base, but all flags are turned off, which should behave the same as the baseline. Config 2 turns on hotness tracking, but does not use it. Configs 3–4, 5–6, and 7–8 are ThinGC with and without cold storage compaction for different ColdAsDead.

Table 1. Benchmarking configurations. Config 0 is our baseline: unmodified ZGC. 0/1 means a flag is off/on. As for ColdAsDead, we pick 0, .5 and 1. Config 2 collects hotness info but does not use it.

Tuning Knobs	ZGC	ThinGC Configurations							
	0	1	2	3	4	5	6	7	8
HotWindows	n/a	0	1	1	1	1	1	1	1
ColdAsDead	n/a	0	0	0	0	.5	.5	1	1
ThinGC	n/a	0	0	1	1	1	1	1	1
CSCompaction	n/a	0	0	0	1	0	1	0	1

4.1 Data Collection and Visualization

In order to study the impact of ThinGC, we collect data on three aspects: execution time, ZGC working set, and data moved into/from cold storage. Next, we go through each of them and describe how we collect the data and visualize it.

Execution Time. For each config in Table 1, we measure the wall-clock execution time of 31 JVM launches. The result for the first is dropped, because it may have accumulative effective (e.g. loading of dynamic libraries) on the subsequent runs. We visualize the execution time for the last 30 runs using box plots [14], as exemplified by the top plot in Fig. 10. The middle plot shows the mean estimate along its 95% confidence interval, constructed from bootstrap sampling, picking the 2.5 and 97.5 percentiles. *If the confidence interval of two configurations do not overlap, then we can conclude, with a confidence level of* 95%*, that there is a significant difference between the two configurations.* The bottom plot shows the relative difference against the baseline, using the mean estimate from previous step. Negative number means reduced execution time. This methodology is taken from [11].

Due to the accumulative effect of freezing/reheating in ThinGC, it's hard to exclude it within a single JVM launch, so for DaCapo suite, we are forced to forgo the built-in iteration logic, only running a single iteration and reporting the wall-clock time from start of the VM to its termination. This means that warm-up time is included in our measurements.

ZGC working set. We use ZGC's built-in and our instrumented logging to record the number of ZGC cycles, and size of relocatable objects for each cycle. For each run, we calculate the max and average size of relocatable objects. Fig. 10 shows how these three metrics are visualized. The relocatable size is obtained through our instrumentation; hence, not available for config 0.

Frozen/Reheat data. We add instrumentation to record the sizes of frozen and reheated objects. For configs 3–8 where ThinGC is enabled, we collected the total size of frozen and reheated objects for each run. One example of such metrics is shown in Fig. 10, and in the same plot, they are normalized against the average relocatable size and frozen size, respectively. Additionally, before running the benchmarks with configs from Table 1, we run the benchmarks once to calculate the reheating percentage in order to check the *inverse temporal locality* property as mentioned in (§ 3.1).

Per-Benchmark Figure Layout. For each benchmark we present a group of 3 plots as three subfigures (a, b, c), each corresponding to one aspect aforementioned; laid out thus:

Execution Time (a)	*ZGC Statistics (b)*	*Frozen/Reheat (c)*
Wall-clock time in seconds	Average GC cycles per run	frozen size and ratio
–"–, w/ mean and confidence interval	Average of average relocatable size	reheated size and ratio
–"–, normalised against ZGC	Average of max relocatable size	

4.2 Graph Algorithms with JGraphT

We run three benchmarks from from the JGraphT library [13]: *(weakly) connected components (CC)* (`BiconnectivityInspector`), which implements biconnected components algorithm [12], *maximal clique (MC)* (`BronKerboschCliqueFinder`), which implements the Bron-Kerbosch maximal clique enumeration algorithm [16], and *page rank (PR)* (`PageRank`), which contains an iterative implementation of page rank algorithm. Inspired by recent GC work [18, 19], we use the graph datasets *uk-2007-05@100000* and *enwiki-2018* from Laboratory for Web Algorithms (LAW) [3, 4]. We implement a minimal driver which does nothing more than calling APIs from LAW to load the graph, inserting all nodes to a new graph from JGraphT, and calling a method from JGraphT on the graph where almost all processing time is spent.

Table 2. LAW Graph nodes and edges.

Dataset	Algorithm	Nodes	Edges	Heap (MB)
UK	–	100 000	3 050 615	*n/a*
	CC	28 128	900 002	1024
	MC	5099	239 294	4096
	PR	100 000	3 050 615	2048
enwiki	–	5 616 717	128 835 798	*n/a*
	CC	28 126	80 002	400
	MC	43 354	170 660	4096
	PR	739 053	5 000 002	4096

For some algorithms, using the complete graph is extremely time consuming, so we only use the partial graph instead. The actual graph size used and heap size for each benchmark is detailed in Table 2. Table 3 shows the reheating percentage, and we can see that *maximal clique (MC)* stands out, while others have relatively small number for this metric. Fig. 10–Fig. 15 show the evaluation results. In the top 2 subplots of subfigure (c) in each group, we can see that there are very few frozen objects using the default EC selection criterion (configs 3 and 4), while this number increases significantly when we start treating cold objects as dead ones. This indicates that large number of cold objects reside on fairly populated pages. For different benchmarks with different graphs, the ratio between frozen size and average relocatable size differs, roughly falling in the range [15%, 100%]. In the bottom 2 subplots of subfigure (c) in each group, we can see that reheating occurs rarely (<10%) for *(weakly) connected components (CC)* and *page rank (PR)*, which indicates that these benchmarks have clear and stable cold–hot classification, and this is consistent with what's shown by the reheating percentage metric. The reheated ratio for *maximal clique (MC)* is decent (~20%), smaller than what's reported by the reheating percentage metric. This is probably because reheated objects are placed in fresh pages, which keeps them hot for at least one non-GC period, preventing from frequent frozen-to-reheating transition.

The large execution difference, shown in subfigure (a), may seem a bit surprising, since it's not aligned with the reheated ratio. The real cause is the affected program locality in configs 5–8. As ColdAsDead increases, more pages are selected for relocation. Mutators accessing objects in those pages will relocate them, which may or may not improve locality, depending on if such access is stable and recurring. Execution time, as an aggregated metric, shows the combinational effect from changed program locality and reheating overhead. Probably due to the intrinsic irregularity of graph traversal, program locality is the dominating factor here, while the reheating overhead is comparatively low.

4.3 DaCapo Suite

Next we look at the DaCapo benchmarks [2], version 9.12-bach-MR1. Not all DaCapo programs run on JDK 13 (the release ThinGC is built upon). Thus, we include the following programs: *h2*, *jython*, *sunflow*, *xalan*, *lusearch-fix*, *pmd*, *avrora*, *fop*, and *luindex*. (There is an open issue concerning a potential concurrency bug in `tradebeans` and `tradesoap`, so they are not included.) Since ThinGC focuses on long-lived objects, short-running applications are not intended targets. We select the largest input size for each benchmark (different benchmarks in DaCapo support different input size) in order to prolong the running time. The input size selected and heap size are shown in Table 3. Heap sizes are selected so that a sufficient number of GC cycles occurs. Table 3 shows that most benchmarks have a relative small reheating percentage except for *h2*. Figs. 16 to 20 show results for the DaCapo benchmarks (except *fop*, *luindex*, *lusearch-fix*, and *pmd*, since they run under 5 seconds). Except *h2*, all benchmarks have large portion of cold objects frozen (the ratio with average relocatable size in range 30%–120%) with low reheated ratio. Consequently, the execution overhead is marginal (<2%); there's no significant difference due to overlapping confidence intervals.

h2 exhibits high number of reheats, and large execution overhead (80%) is observed, which indicates that it does not have a stable cold–hot classification, and this is consistent with the reheating percentage metric. It is possible to tune the hotness collection window width and the freezing frequency using HotWindows. Fig. 21 shows the result of increasing HotWindows to 2 from 1 (default). As the frozen size is reduced, so is reheating, decreasing the execution overhead

to 20%. Melt [5] uses an older release of DaCapo, which does not include *h2*, and the result for *h2* is missing in KG-W [1], so we are unable to compare the high reheating for *h2* with others'.

Table 3. Reheating %-age (see p. 4) for checking *inverse temporal locality*, remset memory overhead, input and heap size.

Benchmark	Reheating	Remset	Input	Heap (MB)
CC (uk)	1.0 %	8.6 %	See Table 2	
CC (enwiki)	0.3 %	3.1 %		
MC (uk)	46.9 %	4.1 %		
MC (enwiki)	59.3 %	5.5 %		
PR (uk)	0.1 %	9.3 %		
PR (enwiki)	1.4 %	9.6 %		
h2	113.9 %	4.8 %	huge	6000
jython	6.8 %	0.9 %	large	3000
xalan	22.3 %	3.2 %	large	1800
sunflow	3.3 %	2.4 %	large	1200
lusearch-fix	1.6 %	1.9 %	large	600
pmd	2.0 %	3.2 %	large	500
avrora	2.9 %	2.3 %	large	200
fop	0.2 %	1.1 %	default	150
luindex	0.0 %	0.6 %	default	80

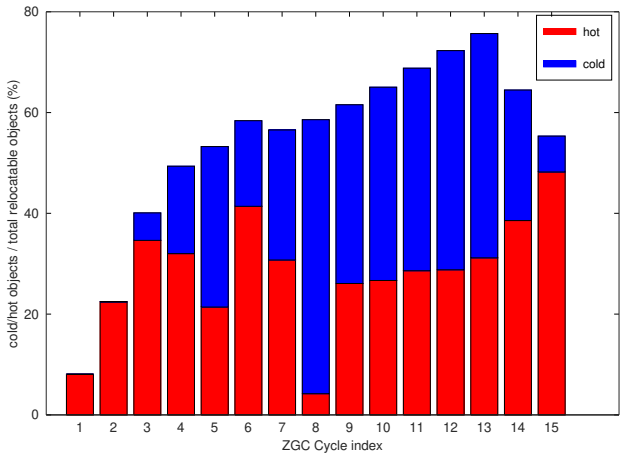


Figure 9. %-age of cold and hot objects per cycle in h2.

In order to investigate the large number of reheating in h2, we rerun h2 with logging to capture per object hotness info every GC cycle. The cold/hot ratio for each GC cycle is shown in Fig. 9, and we can see that a reasonable portion of objects are hot, and in the last GC, almost all objects become hot, which explains the large number of reheats observed.

4.4 Impact of Compaction

Recall in Table 1, we enable cold storage compaction, CSCompaction for configs 4, 6, and 8, and the corresponding compaction off configs are 3, 5, and 7. As is visible from the plots in Figs. 10 to 21, compaction in cold storage has no visible impact on the execution time, which can probably be attributed to the fact that ZGC does not need to "wait" for CSGC.

4.5 Space Overhead of the Remembered Set

In order to assess the space overhead of the remset, we record the number of remset entries in every ZGC cycle, and calculate the average. Assuming each entry takes 16 bytes, the total remset space overhead normalized by frozen bytes[3] is shown under the *Remset* column of Table 3. The percentage is rather small, <10% for JGraphT, and <5% for DaCapo, reflecting a small number of cold-to-hot pointers.

4.6 Summary

Results of JGraphT and DaCapo show that most long-running benchmarks exhibit clear and stable cold-hot classification (low reheating percentage and reheated ratio), with *maximal clique (MC)* and *h2* notable exceptions. When ThinGC is enabled, especially with non-zero ColdAsDead, the execution time is influenced by both changed program locality and reheating overhead, making it harder to isolate the effect of the latter. For JGraphT, the changed program locality plays more significant role in impacting the execution time, while the effect of reheating overhead is almost invisible. For DaCapo, less influenced by altered program locality, show large execution time degradation only in *h2*, and both execution time and reheating overhead drops with less frequent freezing.

5 Related Work

This work is most similar to Akram et al.'s work on a write-rationing GC, KG-W [1] and Bond and McKinley's work on tolerating memory leaks, Melt [5]. Table 4 overviews similarities and differences with these works.

Table 4. Overview of closest related work.

	Rationing				
	Reads	Writes	Reheats	GC	VM
KG-W [1]	✗	✓	On GC	STW	Jikes RVM
Melt [5]	✓	✓	On access	STW	Jikes RVM
ThinGC	✓	✓	On access	Conc.	OpenJDK

Akram et al. [1] show that a very small number of objects see a majority of all writes, and use a moving, copying collector (GenImmix) to relocate objects based on monitoring writes to objects in a portion of the heap. Instead of reheating objects in cold storage on access, they wait until the next GC, which does not isolates the cold storage from direct mutator access. They build on top of Jikes RVM. ThinGC is concurrent, also read-rationing, and implemented in OpenJDK.

[3] We are using frozen bytes from config 5; results are very similar for configs 6–8. Configs 3–4 are less interesting since most cold objects are not frozen because of the default ZGC EC selection criterion.

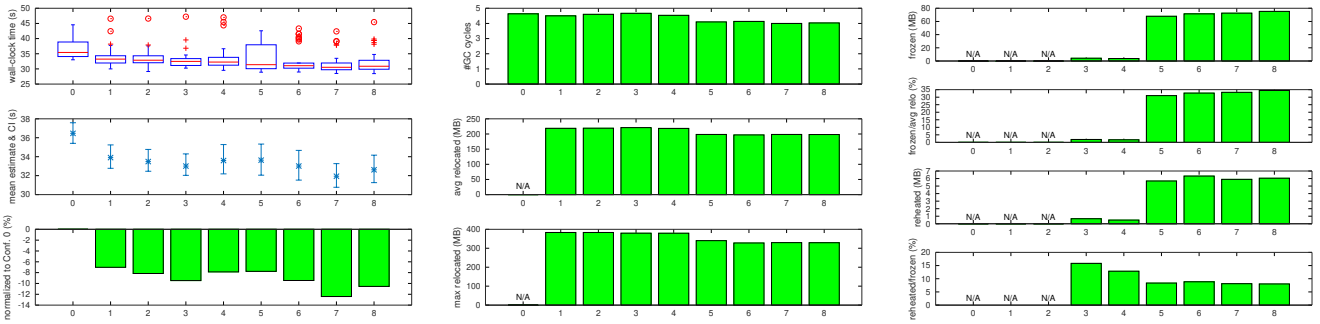


Figure 10. Connected components (CC) with JGraphT using the uk dataset.

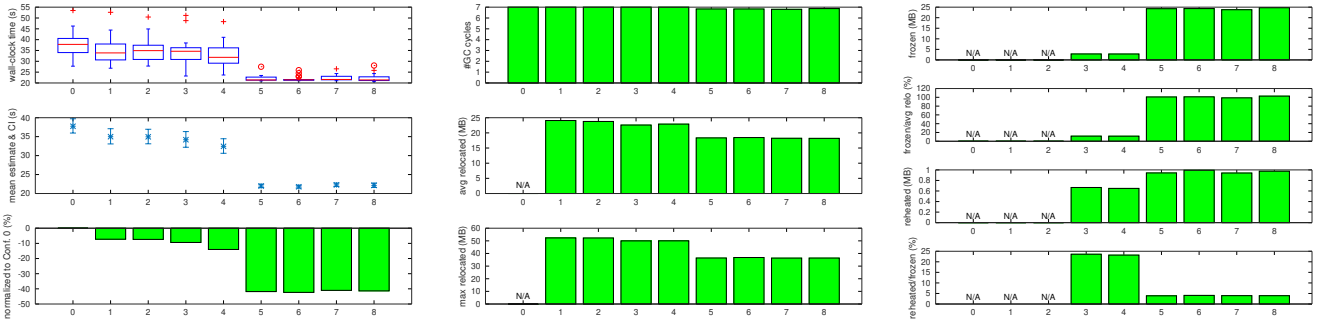


Figure 11. Connected components (CC) with JGraphT using the enwiki dataset.

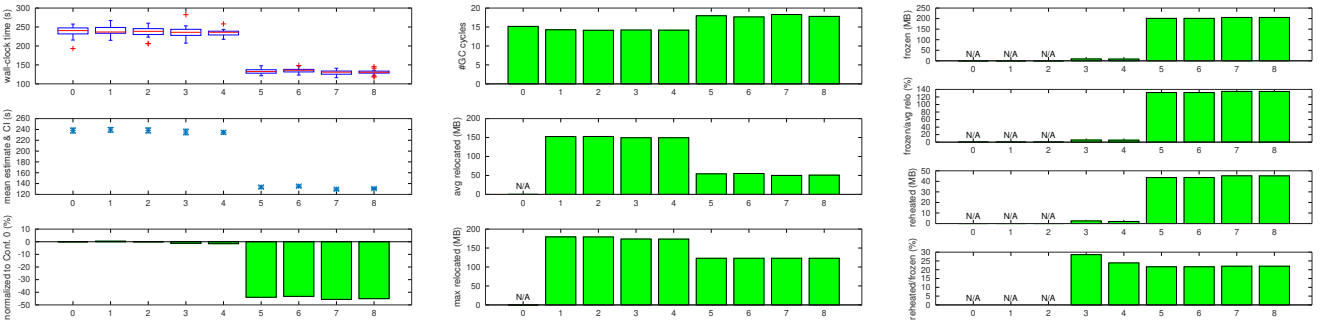


Figure 12. Bron-kerbosch (MC) algorithm with JGraphT using the UK dataset.

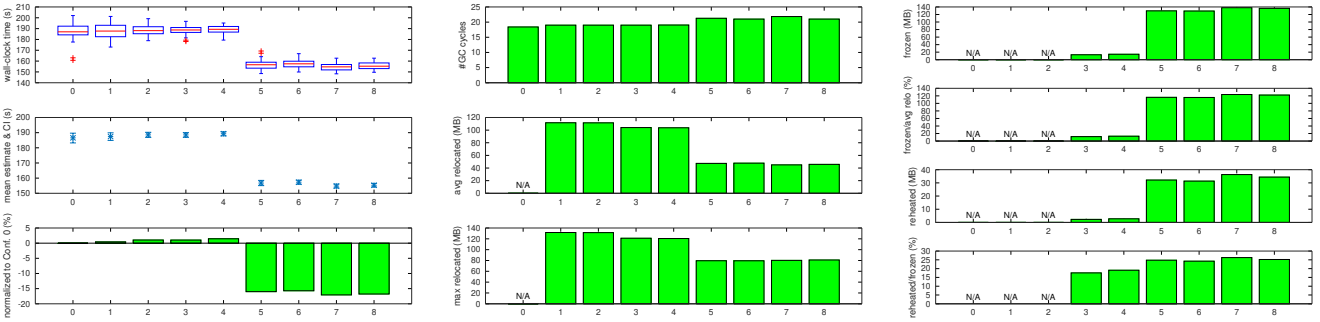


Figure 13. Bron-kerbosch (MC) algorithm with JGraphT using the enwiki dataset.

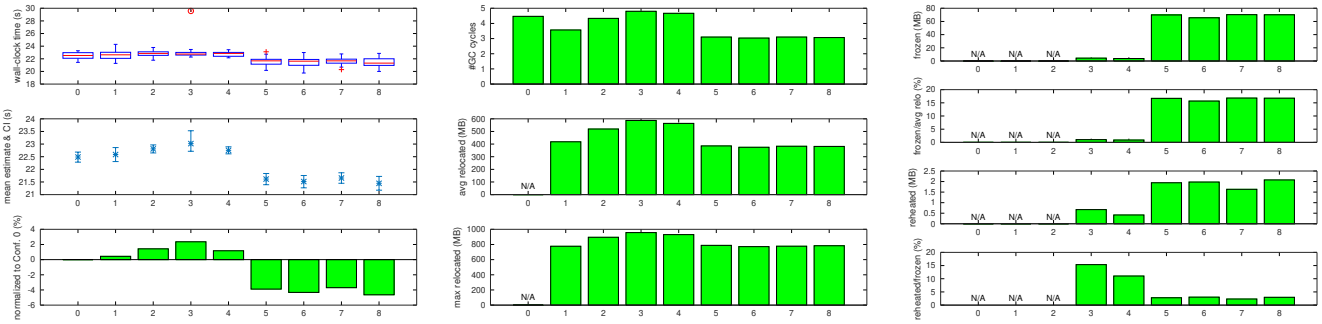


Figure 14. Page Rank (PR) algorithm with JGraphT using the UK dataset.

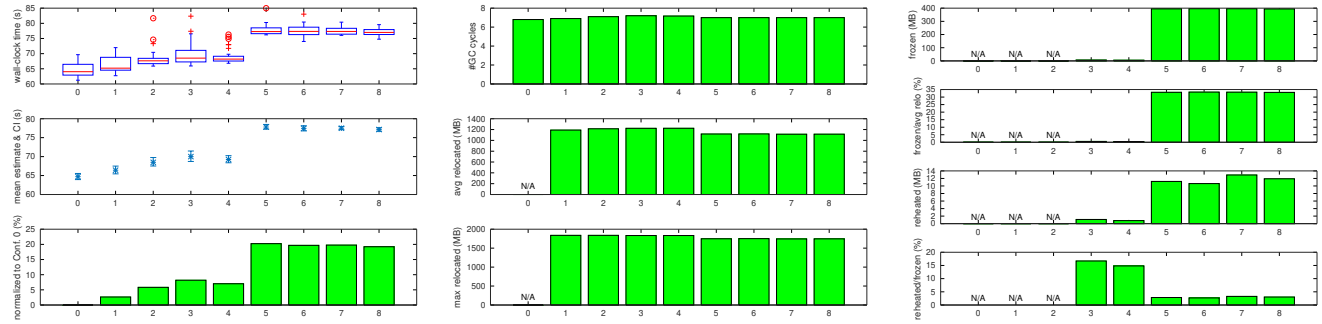


Figure 15. Page Rank (PR) algorithm with JGraphT using the enwiki dataset.

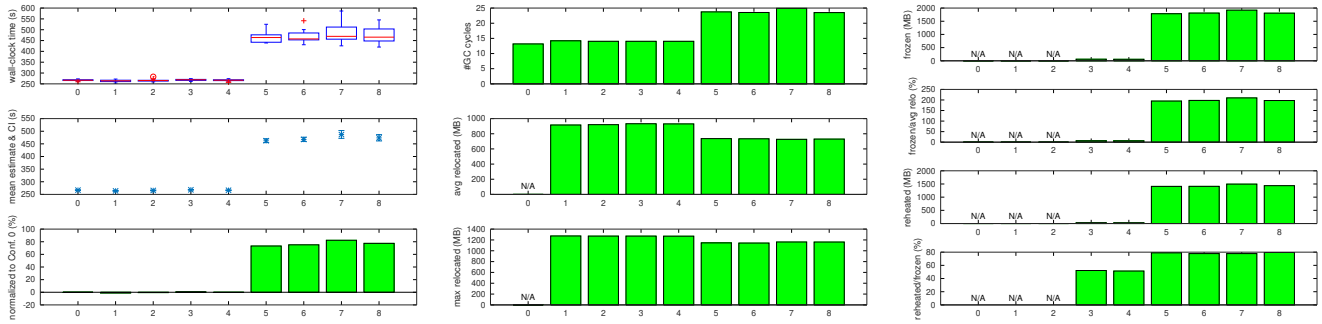


Figure 16. DaCapo's h2.

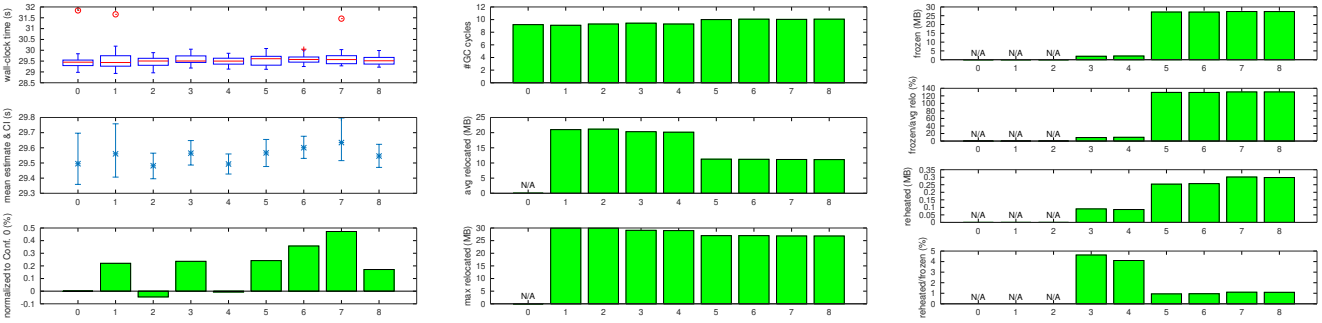


Figure 17. DaCapo's jython.

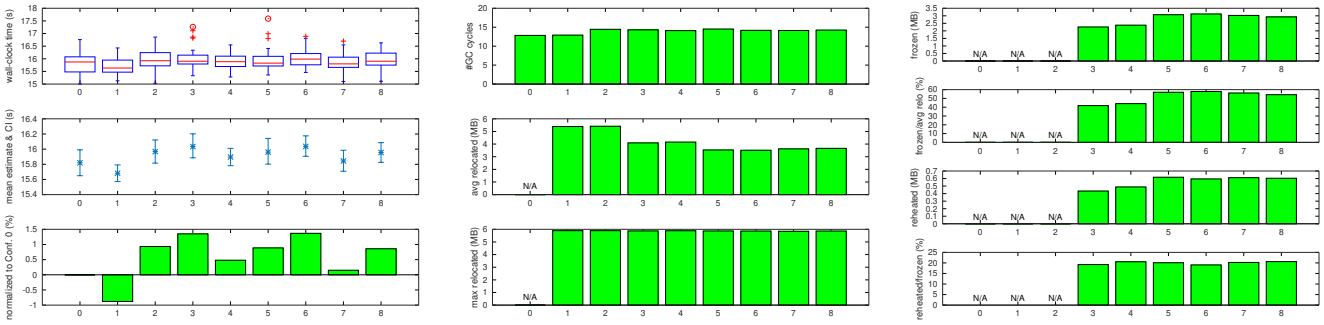


Figure 18. DaCapo's xalan.

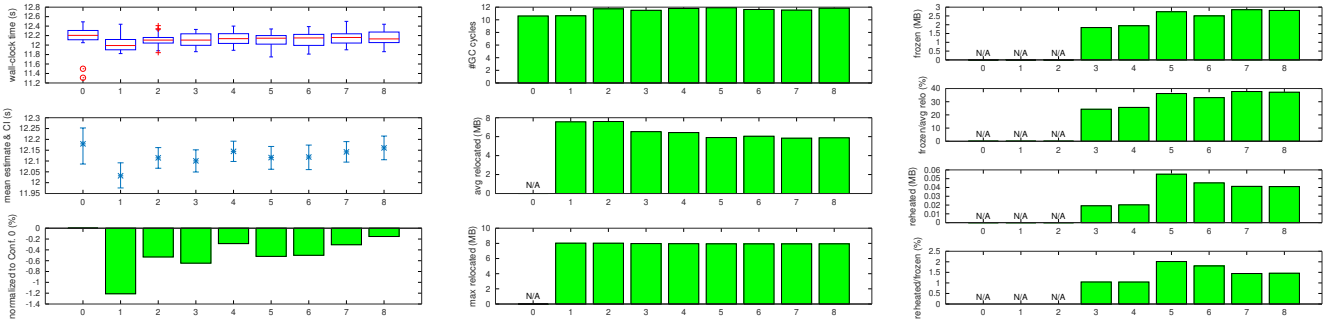


Figure 19. DaCapo's sunflow.

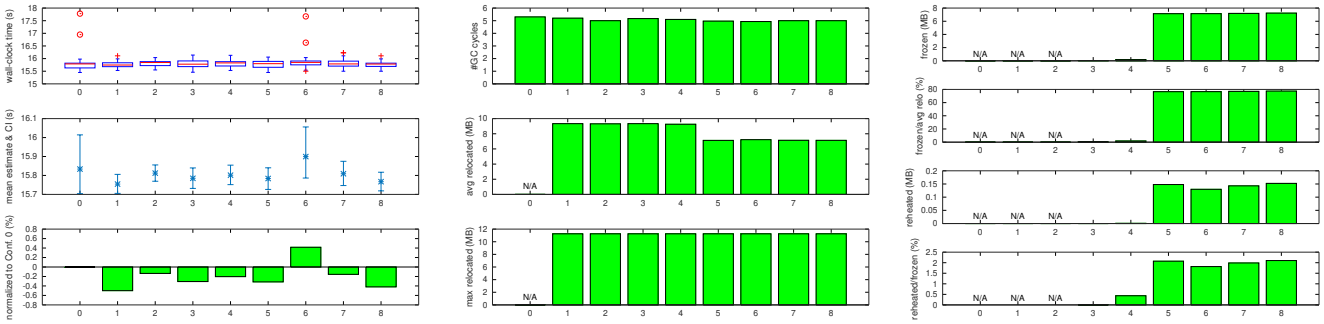


Figure 20. DaCapo's avrora.

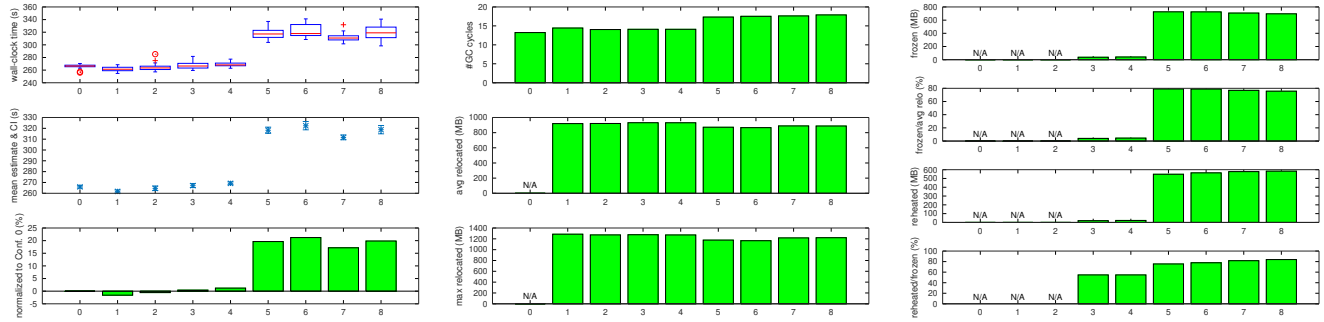


Figure 21. DaCapo's h2, with HotWindows=2.

Bond and McKinley's [5] work on tolerating memory leaks (Melt) uses a very similar design, moving cold (called "stale" in their paper) objects out of the GC working set. Load barriers are used to identify cold objects and to trap accesses to cold storage and reheat (called "activate") cold objects. They evaluate using DaCapo, and in their results, a significant part of heap is cold—the size in stable space could be up to $5x$ larger than the heap. In our case, the cold storage is much smaller, [15%–120%]. The explanation for this large discrepancy is not obvious, considering the significant differences on the implementation level, VM (HotSpot vs Jikes RVM), hot storage GC (concurrent vs STW), cold storage memory (GC vs none), etc. That the Melt evaluation was performed on a uniprocessor might also be contributing.

Chen et al.'s [6] work on heap compression for memory-constrained environments allows an application to run in a heap *smaller* than its footprint. The proposed Mark–Compact–Compress (MCC) algorithm compresses objects when heap compaction is not sufficient for creating space for the current allocation request, and decompresses them when accessed by mutators. Our freeze–reheat design is similar in spirit to the compression–decompression, but there are two key differences. First, frozen objects are not traversed by the main GC (ZGC in our case); instead, their existence is simulated by the remembered set. Second, only objects that are not accessed recently are frozen, in contrast to all live objects are compressed in MCC. Compressing cold objects is an interesting idea, which we leave for future work.

Pauseless GC [8] and its generational successor, C4 [17], use reference metadata to trigger load barrier, which is very similar to colored pointer in ZGC, so the design of ThinGC should be applicable as well.

Cross-component GC [9] introduces an algorithm for GC over component boundaries, where each component has its own collector. The per-component GC is similar to ThinGC, but it assumes homogeneous memory access, because a cross-component garbage collection includes synchronous tracing in both components.

Shenandoah [7, 10] is another concurrent compacting collector in OpenJDK. We have not investigated how easy it would be to extend Shenandoah to track hotness, but if this information could be obtained cheaply, we expect Shenandoah could be used in place of ZGC for ThinGC extension.

6 Conclusion

We have presented ThinGC, a concurrent, read- and write-rationing garbage collector implemented on-top of ZGC in OpenJDK. ThinGC partitions the heap into hot-cold storages, and moves objects which are recently not used by the mutators to the cold storage through an operation called freeze. Objects in cold storage are neither read nor written by mutators. Attempts by mutators to accesses objects in cold storage cause the objects to be reheated and moved back into hot storage. From a mutator's perspective, the cold storage does not exist. Validation of the design against benchmarks from JGraphT and DaCapo shows that ThinGC freezes a significant portion of the heap and rare reheating occurs with marginal overhead for most applications. This reflects that clear and stable cold-hot classification exists widely, and each clustering could be properly contained without much heat leaking.

A Known Limitations

Reheat. Reheating a cold object entails memory allocation in hot storage, which could fail. Currently, the JVM will crash with out-of-memory error. Potential mitigation strategies could be to increase heap size, reduce freezing frequency, or better heuristics for identifying stable hot objects.

Freezing Large Cold Objects. Because large pages do not participate in relocation, large cold objects are currently never moved to cold storage. This is not implemented because we believe reheating large objects could be prohibitively expensive. We leave exploring this for future work.

Non-strong Reference Processing. Reference processing in ZGC requires complete reachability info, which is obtained after the M/R phase is done. As CSGC may span multiple ZGC cycles, we cannot acquire the complete reachability information any more. Therefore, all non-strong references and their referents are always marked hot. In other words, reference processing is performed by ZGC, and only references that have been cleared are subject to being frozen.

In order to estimate how much freezing opportunity is missed due to this limitation, we use instrumentation to count the number of non-strong references identified during marking of each GC cycle. We calculate the average, over all GC cycles, to approximate the memory usage of non-strong references, and assume each non-strong reference is 48 bytes. For JGraphT benchmarks, non-strong references occupy ~130KB, while ThinGC freezes memory on the magnitude of megabytes with very few reheats. The results are similar for DaCapo benchmarks also; *e.g., jython* has the largest number of non-strong references, ~250 KB, while ThinGC freezes ~20 MB.

References

[1] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[5] Michael D. Bond and Kathryn S. McKinley. 2008. Tolerating Memory Leaks. *SIGPLAN Not.* 43, 10 (Oct. 2008), 109–126. https://doi.org/10.1145/1449955.1449774

[6] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. 2003. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications - OOPSLA '03*. ACM Press. https://doi.org/10.1145/949305.949330

[7] Iris Clark, Roman Kennke, and Aleksey Shipilev. 2019. Shenandoah GC. OpenJDK Wiki. https://wiki.openjdk.java.net/display/shenandoah, (v. 88) accessed 2019-04-05.

[8] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

[9] Ulan Degenbaev, Jochen Eisinger, Kentaro Hara, Marcel Hlopko, Michael Lippautz, and Hannes Payer. 2018. Cross-component Garbage Collection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 151 (Oct. 2018), 24 pages. https://doi.org/10.1145/3276521

[10] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 13:1–13:9. https://doi.org/10.1145/2972206.2972210

[11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[12] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* 16, 6 (June 1973), 372–378. https://doi.org/10.1145/362248.362272

[13] jgrapht 2019. https://jgrapht.org/

[14] Robert Mcgill, John W. Tukey, and Wayne A. Larsen. 1978. Variations of Box Plots. *The American Statistician* 32, 1 (1978), 12–16. https://doi.org/10.1080/00031305.1978.10479236

[15] Scott Oaks. 2014. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. O'Reilly Media, Inc.

[16] Ram Samudrala and John Moult. 1998. A graph-theoretic algorithm for comparative modeling of protein structure11Edited by F. Cohen. *Journal of Molecular Biology* 279, 1 (1998), 287 – 302. https://doi.org/10.1006/jmbi.1998.1689

[17] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[18] Po-An Tsai and Daniel Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 229–242. https://doi.org/10.1145/3297858.3304006

[19] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 347–362. https://doi.org/10.1145/3314221.3314650

[20] T. Yuasa. 1990. Real-time Garbage Collection on General-purpose Machines. *J. Syst. Softw.* 11, 3 (March 1990), 181–198. https://doi.org/10.1016/0164-1212(90)90084-Y