Compressed Forwarding Tables Reconsidered

Jonas Norlinder  
Uppsala University  
Sweden  
jonas.norlinder@it.uu.se

Erik Österlund  
Oracle  
Sweden  
erik.osterlund@oracle.com

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

ABSTRACT

How concurrent compacting collectors store and manage forwarding information is crucial for their performance.

In this paper, we propose CFW, a representation technique for forwarding information that guarantees that forwarding information for an entire heap can be stored in at most 3.14% of its size. By providing such a guarantee, we simplify the task of deploying programs with respect to memory needs. This is important given how memory is typically the dominating factor in the cost model for cloud-based deployments.

We explore the design space of our technique through a prototype implementation on-top of ZGC. A rigorous performance evaluation shows promising results.

CCS CONCEPTS

• Software and its engineering  \(\rightarrow\)  Garbage collection.

KEYWORDS

garbage collection, concurrent object relocation, forwarding information

ACM Reference Format:


1 INTRODUCTION

As part of managing memory, many garbage collectors (GC) move objects around in memory for defragmentation and memory reclaimation. Moving an object requires updating all its incoming pointers to point to the new location. As the location of such pointers generally cannot be known, they must be discovered. Starting from the relocation, to the point in time when we can be certain that all pointers to moved objects have been updated, a moving GC must (effectively) keep a map from old addresses to new. This paper is about how to manage the memory needed for this map, the so-called forwarding information, within a tight overhead bound.

A common way to store a forwarding address for a relocated object is to embed it its (now defunct) original copy. This is simple to implement, and avoids the need for any auxiliary data structure to hold forwarding information, but as a downside, it delays the time to when the space of the original copy can be reclaimed until the GC has processed all objects that may reference the moved object, which may at worst be the entire heap. As most GC’s do not reclaim the space for individual objects, but entire address ranges containing multiple objects (which we denote a page), a single forwarding address stored in a copy of an object may affect the availability several orders of magnitude more bytes.

To avoid blocking the reclamation of a page, forwarding addresses can be stored in a separate table. This is a win as long as the size of the table is smaller than the size of its corresponding page. A downside of having a separate table is that additional space for the table must be available, at least during times when objects are moved around, which increases the amount of committed memory. An additional difficulty is predicting the size of the table, to guarantee the availability of enough memory to hold all forwarding addresses. This means that not only does a programmer have to make sure there is enough of system memory for the object heap, but also the additional overhead of for a forwarding table, complicating an analysis of how much memory an application effectively needs. In the pathological case, when numerous of small objects on a page need a forwarding address, the space for forwarding information may be equal the space the GC is looking to free. Unless such headroom exists, the GC itself will run out of memory before it manages to reclaim any garbage. As the placement of objects on pages is outside of programmer control—so are any means to avoid cases that cause forwarding information costs become significantly inflated.

In this paper, we make the following contributions:

(1) We propose a compact representation for forwarding information that guarantees that the maximal overhead needed for storing forwarding information is 3.14% of the Java heap (§3.1).

(2) We propose a concurrent compaction algorithm, CFW, that uses our compact representation (Algorithm 2). In particular, it addresses latency by supporting mutators’ relocation of objects concurrent with GC threads (§3.4), as well as in-place compaction when running out of available memory for to-space (§3.3).

(3) We show an exploration of the design space to further reduce mutator latency and copying overhead (§3.7) and how to integrate our compact representation with the existing forwarding tables in ZGC (§3.6).

(4) We implement CFW on-top of generational ZGC in OpenJDK and undertake a rigorous performance evaluation that shows that some applications benefit from keeping the allocation order. As others do not benefit from this fact, adding a computational cost only incur a throughput and performance regression (§4.3).
2 FORWARDING IN CONCURRENT GC’S

Many GCs use bump-pointer allocation as it is fast and gives good object locality as two consecutive allocations typically end up in adjacent space. In a concurrent GC, each thread or core can maintain its own allocation buffer, with a pointer to free space, to avoid synchronization. It is needed for contiguous free space stresses the need to defragment memory.

For efficiency and simplicity, the heap is typically divided into individually defragmented pages, rather than defragmenting the whole heap. Common pages sizes are 4kB or 2MB (corresponding to OS page sizes). Compacting GCs recycle pages implicitly during defragmentation in a process referred to as evacuation, where live objects are relocated to another page in a quest to defragment the memory. A key challenge of object relocation is remapping of incoming pointers to the moved objects to their new location. We refer to the address space from which objects are being relocated as from-space addresses, and the address space to which objects are being relocated as to-space.

Compacting collectors have three phases: marking, selection and evacuation (which may be performed in a different order than we present them). During marking, the system identifies all objects reachable by the program threads. During selection, sparsely populated pages are selected for evacuation, although other heuristics may also be used. During evacuation, objects on the selected pages are evacuated, and pointers remapped to point to the to-space location. During this phase there is a need to map objects’ old locations in from-space to their new location in to-space. This map is called forwarding information, and is covered in §2.1.

A GC may be concurrent in one or more phases. Relocating concurrently with application threads opens the possibility for application threads to reuse pages for allocation as soon as all their objects have been evacuated. GCs that pause application threads during object relocation, naturally cannot allow re-use since they are paused. Thus, concurrently evacuating with application threads may result in memory becoming available at an earlier point in time. In OpenJDK only ZGC and Shenandoah are concurrent in all three phases. C4 [19] is a closed-source GC that supports concurrent evacuation. To synchronize GC and application threads, ZGC and Shenandoah briefly suspend the entire program. However, the duration of these pauses are independent of the heap size. ZGC and C4 allow application threads to allocate on from-space pages as soon as that page’s objects have been evacuated. Where forwarding information is stored, dictates how fast space on evacuated pages may be reused.

To minimize latency for mutators accessing objects not yet relocated, mutators may be allowed to help GC threads by relocating them to their own thread local allocation buffer. While this is great for a mutator’s latency since they don’t have to wait for a GC thread to evict the object, this destroys the object creation order. Abuaiadl et al. [1] show that preserving allocation order is important for performance and reports substantial performance regression for a compaction algorithm that does not keep object order.

To understand the performance impact on a realistic workload, the design needs to be implemented in a GC/VM that the workload would actually run on, since there are a lot of parameters that will affect performance and many of these affect each other, even more so in a concurrent GC. A popular application for large databases is Cassandra. For this application both latency and throughput are important metrics. Cassandra is punished by single-generational GC. Given these requirements we opted to prototype on a still experimental version of ZGC that supports multiple generations.

2.1 Storing and Managing Forwarding Information

Two common approaches for storing forwarding information are (1) embedding the forwarding address for an object inside its from-space copy immediately following its relocation, and (2) storing forwarding information in a separate table. The next two sections discuss these in order. In OpenJDK all GCs except ZGC store forwarding addresses in from-space. Among commercial grade GCs, only ZGC and C4 [19] use a separate data structure to store forwarding information.

2.1.1 Forward Information in From-Space. For GCs that stop the application during the evacuation phase, storing forwarding information in obsolete copies of old objects is a natural design decision since the application will not attempt to allocate more memory until the phase is over. As a consequence of this design, the time until the page can be reused is controlled by the need for the forwarding information. Because program threads may continue to allocate memory concurrently with evacuation, delaying the time until a page can be reused may contribute to increased memory pressure. Additionally, it introduces a lower bound on the size of objects: no object may be smaller than its forwarding information.

![Figure 1: Forwarding information in Shenandoah.](image)

Fig. 1 shows an example of storing forwarding addresses inside of objects themselves in the Shenandoah collector. Shenandoah uses the object header to store a forwarding address, in the current object copy. As Shenandoah manages memory on a per-page basis, a page may not be reused until all references in the heap have been remapped, i.e., when there are no incoming pointers to objects on the page from live objects.
2.1.2 Forward Information in a Separate Table. An alternative to storing forward-addresses in the from-space is to use a separate table mapping from-addresses to corresponding to-addresses. This design requires additional memory for the table, proportional to the number of evacuated objects. In return, it allows a page to be made available for allocation as soon as all its objects have been evacuated. Moreover, the need to keep memory available that is proportional to the relocation set during the relocation phase is reduced.

Fig. 2 shows an example of storing forwarding addresses in a separate table in ZGC. Each page has its own table of forwarding information, and as soon as this table is filled out and all objects relocated, the page can be freed. The forwarding table may not be free’d until all references in the heap have been remapped, meaning there are no incoming pointers to objects in the forwarding table from live objects.

Storing forwarding information in an old copy of the object lower-bounds object sizes since it needs to be able to store the forwarding pointer. A recent JDK Enhancement Proposal\(^1\) proposes to eliminate the mark word from the object header to preserve space. To enjoy the benefits of this optimization, GCs in OpenJDK may need to adopt a separate-table approach.

2.1.3 Summary. Forwarding information must be kept alive until we have a guarantee it is no longer needed. If forwarding information is stored on a page, the entire page cannot be recycled or reclaimed until after remapping is over. This promotes a smaller page size which has e.g., negative effects on locality and prefetching. A separate table of forwarding information has the potential of occupying less memory than a page, which must initially be present in memory at the same time as the corresponding page, but allows the page itself to be free’d faster. As mutators and GC workers require frequent access to forwarding information, it is crucial for latency that this can be performed efficiently.

2.2 Motivation/Problem

The goal of this work is a guaranteed, low, upper bound on the memory needed to store forwarding information. As heap residency increases, so does the number of object relocations—which favours a compact representation.

Fig. 3 shows the forwarding use when running BigRamTester with ZGC and max heap size of 3300 MB. Numerous pages with many live objects are selected for evacuation to meet the fragmentation limit\(^2\) resulting in a peak memory overhead of 592–1061 MB just for forwarding information, in addition to the upper bound of the Java heap size (i.e., a 18–32% overhead). Thus, a total of 3892–4361 MB of system memory is needed to avoid out-of-memory. BigRamTester \(^3\) is a benchmark that tries to simulate an “in-memory database” \([17]\).\(^4\) BigRamTester stresses the collector by creating an unevenly fragmented heap with many long-living objects.

3 COMPRESSION: FORWARDING INFORMATION

Similar to Abuaiadh et al.’s heap compaction algorithm \([1]\) and Kermany’s and Petrank’s The Compressor \([14]\), our approach to lower the amount of memory needed for forwarding information is based on replacing the lookup in the de facto forwarding table of Shenandoah or ZGC by a function that given a from-address computes the to-address using the from-space liveness structure.

The key is that a to-address for a given from-address can be calculated as the start address of the page we are relocating to plus the sum of the sizes of all preceding live objects which are (will be/have been) relocated to the same page. To make this design practical, this calculation must be made performant (c.f., §2.1.3) by precomputing as much as possible of the necessary information, at the same time taking care to ensure that the memory requirements for precomputed values stay sufficiently low. (Here, precomputing means that the values can be calculated by GC worker threads as part of the GC cycle. This moves work from mutator threads, which is key to preserving performance.) As we shall see in §4, our design is guaranteed to never use more than 3.14% of the memory it relocates for forwarding information (compared with 100% in

\(^1\)https://openjdk.java.net/jeps/8198331

\(^2\)In ZGC pages are selected for compaction given some fragmentation limit

\(^3\)https://bugs.openjdk.java.net/browse/JDK-8152438

\(^4\)Originally written to expose weaknesses in the work-sharing protocol in the G1 collector, but have also been used to find problematic cases of memory usage of the remembered set (also in G1) \([17]\).
the pathological case for a forwarding table), at a mostly negligible performance cost (c.f., §4).

3.1 A Mostly-Precomputed Forwarding Table

We start by introducing the various components of our design before explaining how they come together in a mostly-precomputed forwarding table that allows efficient relocation by mutators and GC threads.

From-pages ($P_F$) are divided into blocks ($B$). Blocks are our unit of relocation and all the live objects on a block will be relocated to the same to-page ($P_T$), and all the blocks of one from-page will maximally be spread over two to-pages. Each from-page has a forwarding table ($T$) which has an entry for each block on its from-page in the form of a fragment ($F$), and 1–2 to-pages. A table with two to-pages also keeps track of the page break, the lowest index of all blocks which will be relocated to its second page. A fragment is a data structure that stores the sum of the sizes of all live objects on the preceding fragments which will be relocated to the same to-page, a livemap of its corresponding block in the form of a bitmap where the beginning and end of live objects are demarcated by set bits, a lock-bit, and a relocated flag.

The start and stop bits in the livemap allows us to calculate not only the start-address of each object on the from-page, but importantly also their sizes. This is important as one expensive part during copying of an object is querying its class’ metadata to get the object’s size.

The key API of the forwarding table is a function that takes a from-address as argument and returns a corresponding to-address. Instead of populating this table with from-address keys mapping to to-address values as objects are relocated as in ZGC, our forwarding table computes the to-address $t$ for the from-address $f$ thus: $t = A + P_T\text{.base} + \text{offset}$.

The key API of the forwarding offset, where:

- $A$ is the sum of the sizes of all live objects that precede $f$ on its containing page $P_F$;
- $P_T\text{.base}$ is the base address of the page that $f$ will be relocated to; and
- $\text{offset}$ is the offset into $P_T$ where relocation of $P_F$ will start.

We can precompute both $A$ and $\text{offset}$, but not $P_T\text{.base}$. The reason for the latter is to keep memory low—we do not allocate a to-page until it is needed for relocation. Until then, we cannot know its address.

The reason for wanting a precomputed $A$-value is mutator relocation: if a mutator accesses an object that is about to be relocated, it must relocate the object first, before proceeding with the access. In this case, being forced to compute the sums of potentially many object sizes imposes too much work on mutators. However, storing one $A$-value per object is not space-efficient. The trade-off we choose in order to balance memory and computation is to base forwarding per block rather per object. Thus, we only need to store one $A$-value per block. This adds more forwarding work to mutators, which means that blocks must not be too large.

Fig. 4 shows three from-pages, their associated forwarding tables, and how the pages are relocated on to two to-pages. Fig. 5 shows the third from-page from Fig. 4 and its forwarding table in more detail.

We now continue with a step-by-step description of how the forwarding table is precomputed (populated).

Population of Compressed Forwarding Tables. Algorithm 1 describes the step-by-step process for creating and populating the compressed forwarding tables. For simplicity, we describe it as a sequential process. Our implementation is parallelized with worker threads concurrently removing pages from the relocation set (RS) to relocate. This activity takes place once per GC cycle, prior to the start of the relocation phase.

We make use of the functions block_offset and size. The function block_offset calculates the offset in bytes of an address on its block. The function size looks-up an object’s size in bytes by querying its class.
Algorithm 1: Populating of Forwarding Tables

```plaintext
procedure populate(RS) : TS is
    // RS is a list of pages selected for evacuation
    // TS is a list of populated compact forwarding tables
    TS ← new list // result list
    A ← 0 // accumulated live bytes
    T' ← null // previous table
    for page P ∈ RS do
        T ← new table // page’s table
        T.orig ← T'
        for i ← 0..|P.blocks| do
            F ← new fragment
            T.fragments[i] ← F
            for live object o ∈ P.block[i] do
                s ← size(o)
                A ← A + s
                j ← block_offset(P, o)
                F.livemap[j] ← true // start bit
                if j + s/8 < sizeof(F.livemap) then
                    F.livemap[j + s/8] ← true // stop bit
                if A > P.size then
                    A ← A − F.offset // reset
                    F.offset ← 0 // first on next page
                    T.page_break ← i
                    T' ← T
            TS.append(T)
        return TS
```

The algorithm proceeds page by page through the relocation set (for loop starting on Line 5) and creates one forwarding table T per page (Line 6). For each block on the page, we create a corresponding fragment (Line 9) that knows the total size of all live objects on preceding fragments that will be relocated to the same page (Line 10), plus a livemap where each bit corresponds to 8 bytes of the block (as allocations are word-aligned), and where the beginning and end of each live object are demarcated by set two bits (Lines 16 and 18). The stop bits have two purposes, they are needed to query preceding bytes after the object have been copied, as after that point the page may be free’d, and we cannot query its class. They are also used as an optimization during relocation to provide a way of quickly calculating the object’s size. As the last object may not end on the same block, its stop bit is optional (Line 17). If it is missing, the object’s size is obtained from querying its class.

The variable A keeps track of the total size of all objects relocated to a page (initialized on Line 3 and incremented on Line 13). If the objects on a fragment do not fit on the current to-page, the entire fragment is moved to the beginning of the next to-page (Lines 19–21). To track the location of the page break, we store the index of the block in the forwarding table (Line 22). Finally, each forwarding table knows the identity of the forwarding table responsible for creating its to-page (Lines 7 and 23). This is used to coordinate concurrent attempts to create the to-page.

Our algorithm introduces an order in which n from-pages are mapped to m to-pages. We call such an order a chain. In our implementation, we create (in parallel) as many chains as we have workers, and relocate the chains in parallel. The chains are important for handling situations where the system runs out of memory while collecting garbage (c.f., §3.3).

Since from-pages and to-pages have equal size, each from-page will be relocated to at most two to-pages. Thus, each forwarding table keeps track of two to-pages: to_fst and to_snd. Blocks whose index are lower than the page break will be relocated to to_fst and all other blocks to to_snd.

3.2 Using Compressed Forwarding Information

This section describes how GC workers and mutator threads use our compressed forwarding information to perform relocation. We first describe the general algorithm for relocating all objects on a block using its corresponding fragment, which is used by both GC works and mutators. We then show how GC threads can relocate entire pages in a straightforward manner. Finally, we discuss mutator relocation.

Algorithm 2: Relocation of all live objects on a block

```plaintext
procedure relocate_block(B, P_f, T) is
    // B is a block, P_f is a from-page, and a T is a table
    i ← (B.base − P_f.base)/Block_Size
    F ← T.fragments[i]
    if T.page_break < i then
        P_r ← T.to_snd
        copied ← 0
        start ← next_set_bit(F.livemap, 0)
        while 0 ≤ start do
            from ← P_r.base + i × Block_Size + start × Obj_Align
            to ← P_r.base + F.offset + copied + start
            step ← next_set_bit(F.livemap, start + 1)
            if step < 0 then
                size ← size(from)
            else
                size ← 8 × (step + start + 1)
            copy(to, from, size)
            copied ← copied + size
            start ← next_set_bit(F.livemap, step + 1)
```

3.2.1 Relocation of All Objects on a Block. Algorithm 2 describes the step-by-step process of relocating all live objects on a block. For simplicity, the description starts after we have looked up the corresponding fragment, which is a simple constant-time lookup. Obj_Align is the object alignment in bytes, which controls the Block_Size (c.f., §3.5). (In our implementation on ZGC, these numbers will typically be 8 and 256 respectively.)
We make use of the functions `next_set_bit`, `size` and the built-in `copy` in OpenJDK. The function `next_set_bit` looks up the index of the next set bit in a livemap, starting from a given index (inclusive). When there is no next set bit in the livemap, the function returns −1. An important optimization for implementing `next_set_bit` is to use a machine instruction to count trailing/leading zeroes. On x86-64 [13] and ARM64 [3] such instruction is available, so scanning of an entry can be implemented using a cheap query of trailing zeroes and shift on these platforms.

A block is relocated by iterating over the livemap of its corresponding fragment (looked up on Line 3). The iteration is implemented as a loop (Line 10) that starts from a state where `start` holds the index of the start bit (Line 9). Before the iteration, we set the counter of bytes copied to 0, and look up the to-page `P_T` for the block in `T` using its index.

First, we calculate the from and to addresses of the object to be relocated. The from-address can be calculated as the base address of the from-page plus the index of the block multiplied by the block size, plus the index of the start bit multiplied by 8 (Line 11). The to-address can be calculated as the base address of the to-page plus the offset stored in the fragment, plus the tally of copied bytes from the block so far (Line 12).

To copy the object (Line 18), we must first calculate its size. We can calculate the size of objects for which there is a stop bit (check on Line 14) quickly by multiplying the number of bits they span in the livemap by 8 (Line 17). If there is no stop bit we must ask the object for its size (Line 15).

The loop ends by incrementing the tally of bytes copied from the block (Line 19), and advancing start to the index of the next start bit (Line 20), or −1 if there are no more objects, which terminates the loop (Line 10).

We use a locking protocol to ensure that no two threads attempt to relocate the same block concurrently (c.f., §3.4).

Having described the basic algorithm for relocating all objects on a block, we move on to show how this is used by the GC workers when evacuating a page.

### 3.2.2 Evacuation of Page by GC Threads

Evacuation of a page by GC threads is straightforward iteration over all fragments of all tables of all pages in the relocation set (RS) as seen in Algorithm 3.

We make use of the function `from_page` that looks up the from page for a forwarding table, and the functions `fst_page` and `snd_page` that lookup a table’s `to_fst` (or `to_snd`) and if it does not exist, gets it from a forwarding table earlier in the chain, or creates it (if the table is first in a chain, or always in the case of `to_snd`, c.f., §A in Appendix).

The algorithm iterates over the forwarding tables in the chain in chain order. For each table, we look up its from page (Line 3), and ensure that the necessary to-pages exist (Lines 4−6). We then iterate over all the blocks on the page and relocate them using the logic of Algorithm 2, which is encapsulated in the function `relocate_block` (Lines 7−9).

GC threads coordinate so that each chain is relocated in full by a single GC worker thread. However, GC threads may still race with mutators to relocate a block (c.f., §3.4).

---

**Algorithm 3:** Relocation by GC Threads of a set of pages whose forwarding tables are chained.

```plaintext
1 procedure relocate_chain(C) is
2     // C is list ("chain") of forwarding tables, C
3     for T ∈ C do
4         P_T ← from_page(T)
5         fst_page(T)
6     if T.page_break ≥ 0 then
7         snd_page(T)
8     for i ← 0..|T| do
9         B ← P_T.blocks[i] // get ith block
10        relocate_block(B, P_T, T) // Algorithm 2
```

---

### 3.2.3 Mutator-Induced Relocation

Algorithm 4 describes how mutators use the information in the compressed forwarding table. We defer discussion of coordination between mutators and GC threads to §3.4.

In ZGC, mutators that access an object that is about to be relocated will help the GC relocate the object before commencing to access it. This causes the object to be moved to the mutators own allocation buffer, which can further improve locality over just keeping the allocation order, and was explored by Yang et al. [21]. Our design differs from ZGC in two respects:

1. We force the mutator to relocate the entire block, not just the single object, and
2. relocation always uses the predefined location, so who relocates does not have any effect on locality.

By forcing mutators to relocate all objects on a block as opposed to a single object, we put more work on mutators, which may manifest itself through increased latency, c.f. §4. This stresses the importance of the size of the block. If we for example made blocks four times larger, we would save ≈40% of memory by only needing to store one offset for every four fragments, but this would slow down mutators due to additional relocation work.

We make use of a function `table` that looks up the forwarding table for a given page.

**Algorithm 4:** Mutator-Induced Relocation

```plaintext
1 procedure relocate_object(o, P_T) is
2     // o is an address o that falls on page P_T
3     T ← table(P_T)
4     if (o − P_T.base)/Block_Size
5     B ← P_T.blocks[i]  // Algorithm 2
6     if T.page_break < i then
7         fst_page(T) // Ensure to-page has been created
8     else
9         snd_page(T) // Ensure to-page has been created
10        relocate_block(B, P_T, T) // Algorithm 2
```

Given an address `o` of an object that the mutator wants to access, we obtain the page’s forwarding table (Line 2) and calculate the
When memory is full, calls to relocated which it may finally transition to warding information as the linearization point. (Failing threads can relocated, followed by changing the fragment's state to locked operation that will return the state of the fragment. If the operation attempt to put the fragment in the 𝐹, this state, the fragment may transition into the unrelocated is implemented as a monotonic state change and uses two bits relocation has been performed.

Concurrent relocation is implemented in a lock-free manner, with installing the for-
mented by copying the object to be relocated to the relocating slide all live objects in from1 down to create a situation where from1 can be used as to-page by table2 and table3. This in-place compaction is different from normal relocation in three main ways:

1. Relocation of objects during in-place compaction must take place in increasing address order.
2. While in-place compaction of a page is taking place, no other page may relocate to the place that is being in-place compac-
ted.
3. In-place compaction is only performed by GC threads. If a mutator thread sees a failing call to fst_page() or snd_page(), the thread is forced to block until a GC worker has performed the compaction.

Cases 2. and 3. above mean that a call from a mutator to fst_page() and snd_page() may block because a GC thread is currently in-place compacting the page which will eventually be the result of the call. Since GC threads do not process overlapping sets of pages, a GC thread will never block on calls to fst_page() or snd_page()—instead, they will perform any necessary compaction.

### 3.4 Coordinating Mutators and GC Workers

In this section we describe the synchronization in our system.

#### 3.4.1 Concurrent Relocation

If concurrent relocation is imple-
mented by copying the object to be relocated to the relocating thread's own allocation buffer, concurrent relocation of an object can be implemented in a lock-free manner, with installing the forwarding information as the linearization point. (Failing threads can simply undo their previous allocation.) However, in our case, as the location of a relocated object is "fixed", there is no straightforward lock-free implementation. Instead, we impose a locking scheme on relocation and threads that attempt to relocate the same object concurrently will leave all but one of the threads waiting until the relocation has been performed.

We use a locking scheme with one lock per fragment. This lock is implemented as a monotonic state change and uses two bits per fragment. A fragment starts in the state unrelocated. From this state, the fragment may transition into the locked state, from which it may finally transition to relocated. A thread that wants to relocate a block B with corresponding unrelocated fragment F will attempt to put the fragment in the locked state. This is a blocking operation that will return the state of the fragment. If the operation returns locked, it was successful and the live objects on B can be relocated, followed by changing the fragment's state to relocated. If the operation returns relocated, some other thread has succeeded in relocating the block, so the thread can proceed to its next step (e.g., lookup the forwarding address of a particular object, in the case of the mutator). See Algorithm 5 in Appendix for the skeleton logic that wraps Line 9 in both Algorithm 3 and Algorithm 4.

#### 3.4.2 Concurrent To-Page Creation

The sharing of to-pages for multiple from-pages, and concurrent relocation to a to-page which is not yet allocated can lead to multiple threads competing to allocate a page to use as the same to-page. We coordinate these by allowing page creation to succeed in each competing thread, but use compare and set (CAS) to install the new page in the target field which succeeds only if the value of the variable is null. If the compare and set fails, some other thread has succeeded in creating and installing the page. In that case, we return our own page back to the page allocator, and continue using the successfully installed page. The chaining of forwarding tables ensures that all threads that attempt to install a page serving relocation from multiple from-pages will CAS on the same to-page field. This is encoded in the fst_page() and snd_page() functions. For example, in Fig. 4, the calls fst_page(table1), fst_page(table2), and fst_page(table3) will all CAS on table1, to_fst.

### 3.5 Memory Overhead of Our Implementation on ZGC

The bulk of memory for a forwarding table is the vector of entries, as shown in Fig. 5. Each entry is a struct packed into 64-bits that holds accumulated live bytes, start/stop bits, and two metadata bits (locked and relocated). The number of entries is correlated with how many bits that are needed to describe all possible start/stop addresses on a page. We support ZGC's small and medium page sizes (large pages do not need forwarding). Objects on small pages (2 MB) are word-aligned, and objects on medium pages (4–32 MB) are aligned on a 64–512-word boundary. Thus, the number of bytes spanned by a single entry can be described as 32 × Obj_Align bytes, meaning small-page entries span 256 bytes, and medium-page entries span 16–128 kB bytes. Therefore, we need 8192 and 256 entries for small and medium pages respectively. The memory needed for forwarding information is a function of the amount of entries. For small pages this amounts to \(8192 × \frac{32}{2} = 3.125\%\) and for medium pages less than 1%. Each forwarding table holds some additional per-page data e.g., the address of the from-space page to evacuate, to_fst, to_snd, etc., making the overhead for small pages settle at 3.14% and for medium pages still < 1%.

### 3.6 Compressed Forwarding Tables as a Fallback

The space for a table-based storage of forwarding information scales linearly with the number of forwarded objects, regardless of how the table is represented (separate table, embedded in old copy). In contrast, the cost of our forwarding table design scales linearly with the amount of pages selected for relocation. Fig. 6 and Fig. 7 show a comparison of these two cost functions in the ZGC implementation. Fig. 6 shows that for sparsely populated pages, the table-based solution uses less memory than compact forwarding tables, but that as soon as the number of forwarded objects exceed 4096, the

---

*ZGC dynamically sizes medium pages depending on the size of the heap.*
Figure 6: Memory cost for forwarding information per page, zoom in on $x = [0, 4800]$.

Figure 7: Memory cost for forwarding information per page. Cost model for single page.

Tables are turned (as illustrated by Fig. 7). In conclusion, programs where at most 3.14% of objects survive the next GC cycle, compact forwarding tables inflate the average memory costs. Indeed, Fig. 8 reports on experiments that confirm this suspicion. Our experiments align with Yang et al. [21] observation that about 1% of all objects survive a GC cycle in SPECjbb2015.

At the cost of increased technical debt in maintaining two parallel ways to manage forwarding information, it is easy to combine the two approaches and only use compressed forwarding tables when the memory requirements of the hash-table based solution exceeds a given limit. In ZGC, the number of pages selected for relocation as well as the density of their live objects is known before the creation of forwarding tables. Thus, it is possible to calculate which approach would be most space-efficient to use in each GC cycle and proceed accordingly.

3.7 Explorations of the Design Space

In a concurrent collector, a mutator that touches an object slated for relocation should ideally be able to relocate the object itself, rather than wait for a GC thread to do so. A side-effect of our block-based design is that mutators are forced to relocate an entire block as opposed to a single object, which may have negative effect on latency-sensitive workloads. We explored three different designs for partial evacuation of blocks.

3.7.1 Partial Evacuation Using Ordinal Bits. In this design, we leverage the fact that not all bits that we allocate for storing accumulated live bytes are needed: we can reappropriate 11 adjacent bits in each fragment. We let each bit signify the ordinal of the objects on the fragment i.e., the nth bit corresponds to the nth object on the block, and use that bit to track the relocation status of the object. This approach is pragmatic as the blocks in our implementation can hold up to 16 objects, meaning we only support partial evacuation for the first 11 objects on a block.

3.7.2 Partial Evacuation Using Vector Array. This design is the same as the ordinal bit, but using a separate bit array that is able to track the status of all objects on a block.

3.7.3 Partial Evacuation Using Inline Vector. A downside of the previous design is that it requires an additional load to access the partial evacuation map during relocation. To mitigate this, we explore a design where we interleave fragments and partial evacuation maps in memory (effectively the same as growing the size of the fragment). This ensures that loading one, loads the other in cache.

In addition to the above, we also explored the effects of reducing the number of calls to copy.

3.7.4 Coalesced Evacuation. In this design, adjacent objects in a block will be copied as a single object. Adjacency is not precomputed.

4 EVALUATION

The motivation for this work is to guarantee that the memory required for forwarding information does not exceed $\approx 3\%$ of the Java heap, with a limited performance regression. We replace a hash table that offers constant-time lookup by computation based on mostly precomputed values. As such, we expect some performance regression in most work-loads, as translating from-addresses to to-addresses becomes a more computationally intensive operation. Additionally, mutators that are relocating may end up having to relocate additional objects which may manifest itself as increased latency or decreased throughput.
Since our compressed forwarding table implementation replaces ZGC’s existing forwarding tables (and also parts of the relocation logic, as shown in §3), we benchmark against ZGC, and show our performance numbers relative to ZGC (absolute numbers can be found in Appendix). It should be noted that our ZGC baseline is generational ZGC, which is about to be, but has not yet been released in OpenJDK. We compared the performance of generational ZGC with single-generational ZGC in JDK 17 using the DaCapo benchmarks and found performance to be on-par, with some expected differences due to the introduction of multiple generations.

We have implemented a version of generational ZGC that falls back to using our approach if the memory overhead of a GC cycle exceeds a given threshold, and otherwise uses the ZGC forwarding table and relocation logic. In our benchmarking, we set the threshold to 4%, the nearest integer for which memory would be saved by using CFW. Likewise, we have also implemented partial evacuation and coalesced evacuation to explore optimizations for mutator latency.

We evaluate using the DaCapo benchmark suite [4] to target a wide range of different workloads. Moreover, Cassandra 4 [2] and SPECjbb2015 [18] are used to estimate the impact on mutator latency. SPECjbb2015 also gives score for maximum throughput for its workload. Because of poor scaling in h2 and tradebeans, we run them with 4 application threads (see Fig. 12 in Appendix for additional details). We did not analyze scalability of other DaCapo applications for as their required heap sizes are substantially smaller for the rest. For the other benchmarks we let the application decide how many threads they used. While there is a decade old concurrency bug for tradebeans/tradesoap8 potentially leading to deadlock we opted to still attempt sampling tradebeans. Fortunately, we never encountered a deadlock while sampling data for tradebeans.

Heap sizes were selected to be large enough to almost entirely avoid allocation stalls. For tight heap configurations we searched for the lowest heap that would not result in in-place relocation, but permitted allocation stalls. We tried to avoid in-place as these will severely hurt performance and increase a lot of variability in the results.

4.1 Statistical Methodology
For DaCapo, we follow Georges et al. [6] for shorter running benchmarks and run with a number of iterations that we roughly estimated were beyond the initial warm-up phase of the JVM. The last five iterations are used to calculate an average, which is then reported as a data point. Note here that the reported results in Table 1 (and Table 8 in Appendix) is therefore an average of averages. We only considered the largest workloads that were available as only a larger number of objects is assumed to reveal any performance impact. Nevertheless, we run h2 small as well as large and huge to gauge the performance of all three sizes offered in DaCapo for at least one benchmark.

A lot of effort is often put into minimizing the bias (e.g., Georges et al. [6]) during data collection. Some papers that have also attempted to perform a statistically rigorous analysis on the collected data in the field of garbage collection have used the percentile bootstrapping method [21], Student’s t-test [11, 12], and ANOVA [7]. The percentile bootstrapping method assumes that the sampled data is a good approximation of the true distribution (which would not be safe to assume in our case). Moreover, it does not correct for skewness or bias. Student’s t-test and ANOVA assume that the data is normally distributed, that the variances do not differ, and that variables are independent. As we are comparing different solutions e.g., ZGC vs. CFW, we assume that variance are non-equal among groups. While, the variables are weakly dependent due to caching, dynamic CPU frequency scaling etc., we will assume that they are independent given that enough samples should nullify this effect. We aim to sample at least 30 times9 so that we can assume that the mean of the sampled data is following a normal distribution according to the central limit theorem. We use Grubbs’ outlier test [10] to detect if there are outliers in data. If there are no outliers then Welch’s t-test [20] is used to determine if means are of significant difference with a 95% confidence, otherwise we use Yuen’s t-test [23] where we set the trim variable to 10% for each tail to compensate for the outliers. Welch’s t-test compared to Student’s t-test does not assume that variance is equal and is robust to different sample sizes and will thus produce more accurate results [5]10. Yuen’s t-test is an extension of Welch’s t-test that can, unlike Welch’s, compensate for outliers.

4.2 Machine Configuration
Cassandra was executed on a 2-node cluster, with one client and one server, both machines using two Intel Xeon E5-2630 per box. The server has 256 GB and the client has 128 GB of memory. As CFW does currently not consider NUMA we pin the benchmarks to one of the CPUs through taskset. The machine running SPECjbb2015 and DaCapo used an AMD Ryzen 9 3900XT and had 128 GB of memory. See §B for a thorough description of the setup.

4.3 Results
Using compressed forwarding tables results does not result in an across-the-board regression/improvement. We believe that improvements in performance are due to a cache effect from how CFW preserves allocation order between objects more than ZGC. The effect is most prevalent in sunflow (Table 1) where a run time improvement of 53% could be observed. This speed-up is largely explained by 79% fewer cache misses. It should also be noted that sunflow reports a higher heap usage. This could be explained by a different heap layout leading to less fragmentation, allowing garbage to float for longer, since the selection criteria for recycling pages in ZGC is based on fragmentation. With a normal heap size, we can expect a regression to stay below 4% in DaCapo. Notably for SPECjbb2015 (Table 2) we find a performance improvement on 7% for the throughput score by using CFW. For Cassandra (Table 3) a large regression can be observed on maximum throughput and latency. Most metrics for Cassandra have an observable regression. Alas, we were unable to pinpoint the source of the regression.

8Artifact is available [16]
9https://github.com/dacapobench/dacapobench/issues/99
Table 1: Performance evaluation for DaCapo. Results relative to ZGC. We highlight statistically significant regressions and improvements, and mark them with a *. Heap sizes reduced to increase heap residency are marked ▽. Fallback configurations use ZGC’s forwarding tables when they require less than 4% of heap size, otherwise our forwarding tables are used.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Heap Size (MB)</th>
<th>Config</th>
<th>Performance</th>
<th># GC Cycles</th>
<th>Average Heap</th>
<th>Memory (avg.)</th>
<th>Memory (peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>180</td>
<td>Default</td>
<td>-4.22% *</td>
<td>7.83% *</td>
<td>-43.46% *</td>
<td>-41.55% *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▽ 35</td>
<td>Default</td>
<td>-1.61% *</td>
<td>9.08% *</td>
<td>44.13% *</td>
<td>55.62% *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>Fallback</td>
<td>-4.89% *</td>
<td>2.88% *</td>
<td>1.28%</td>
<td>0.08%</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>160</td>
<td>Default</td>
<td>0.00%</td>
<td>10.76% *</td>
<td>6.98%</td>
<td>63.73% *</td>
<td>63.76% *</td>
</tr>
<tr>
<td></td>
<td>▽ 80</td>
<td>Default</td>
<td>1.54% *</td>
<td>-6.38% *</td>
<td>6.89%</td>
<td>63.73% *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>Fallback</td>
<td>1.08%</td>
<td>2.04%</td>
<td>-16.80%</td>
<td>-25.56%</td>
<td>-24.43%</td>
</tr>
<tr>
<td>h2 (small)</td>
<td>190</td>
<td>Default</td>
<td>1.94%</td>
<td>0.00%</td>
<td>1.13%</td>
<td>-142.12%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▽ 130</td>
<td>Default</td>
<td>-3.92%</td>
<td>2.33%</td>
<td>-7.01%</td>
<td>-9.31%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>190</td>
<td>Fallback</td>
<td>0.51%</td>
<td>1.95%</td>
<td>-2.20%</td>
<td>-1.45%</td>
<td></td>
</tr>
<tr>
<td>h2 (large)</td>
<td>1200</td>
<td>Default</td>
<td>-1.22%</td>
<td>-14.74%</td>
<td>7.68%</td>
<td>-104.57%</td>
<td>-123.44%</td>
</tr>
<tr>
<td></td>
<td>▽ 450</td>
<td>Default</td>
<td>-0.22%</td>
<td>-2.26%</td>
<td>1.49%</td>
<td>35.88%</td>
<td>47.49%</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>Fallback</td>
<td>0.83%</td>
<td>4.73%</td>
<td>-2.30%</td>
<td>0.60%</td>
<td>2.12%</td>
</tr>
<tr>
<td>h2 (huge)</td>
<td>4000</td>
<td>Default</td>
<td>-0.58%</td>
<td>-80.91%</td>
<td>20.57%</td>
<td>-90.23%</td>
<td>-121.59%</td>
</tr>
<tr>
<td></td>
<td>▽ 2000</td>
<td>Default</td>
<td>0.22%</td>
<td>-5.83%</td>
<td>1.99%</td>
<td>52.55%</td>
<td>64.13%</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>Fallback</td>
<td>0.38%</td>
<td>0.41%</td>
<td>-0.07%</td>
<td>-0.31%</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>400</td>
<td>Default</td>
<td>0.51%</td>
<td>0.04%</td>
<td>1.32%</td>
<td>-38.50%</td>
<td>-52.66%</td>
</tr>
<tr>
<td></td>
<td>▽ 80</td>
<td>Default</td>
<td>10.68%</td>
<td>8.98%</td>
<td>0.57%</td>
<td>-16.77%</td>
<td>-20.4%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>Fallback</td>
<td>0.85%</td>
<td>0.04%</td>
<td>-0.01%</td>
<td>7.92%</td>
<td>7.32%</td>
</tr>
<tr>
<td>luindex</td>
<td>40</td>
<td>Default</td>
<td>0.05%</td>
<td>18.86%</td>
<td>37.67%</td>
<td>48.23%</td>
<td>59.25%</td>
</tr>
<tr>
<td></td>
<td>▽ 25</td>
<td>Default</td>
<td>16.27%</td>
<td>2.71%</td>
<td>-2.30%</td>
<td>0.54%</td>
<td>-16.45%</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>Fallback</td>
<td>1.43%</td>
<td>5.95%</td>
<td>0.54%</td>
<td>-12.25%</td>
<td>-16.45%</td>
</tr>
<tr>
<td>lusearch</td>
<td>2800</td>
<td>Default</td>
<td>-0.07%</td>
<td>-1.94%</td>
<td>2.71%</td>
<td>-1077.57%</td>
<td>-1185.58%</td>
</tr>
<tr>
<td></td>
<td>▽ 100</td>
<td>Default</td>
<td>-1.94%</td>
<td>-1.94%</td>
<td>2.71%</td>
<td>-1077.57%</td>
<td>-1185.58%</td>
</tr>
<tr>
<td>pmd</td>
<td>1200</td>
<td>Default</td>
<td>-3.71%</td>
<td>-2.79%</td>
<td>3.48%</td>
<td>37.07%</td>
<td>40.32%</td>
</tr>
<tr>
<td></td>
<td>▽ 120</td>
<td>Default</td>
<td>2.64%</td>
<td>5.81%</td>
<td>2.94%</td>
<td>57.61%</td>
<td>61.68%</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>Fallback</td>
<td>-3.85%</td>
<td>-2.30%</td>
<td>1.50%</td>
<td>2.08%</td>
<td>1.60%</td>
</tr>
<tr>
<td>sunflow</td>
<td>1200</td>
<td>Default</td>
<td>53.53%</td>
<td>28.73%</td>
<td>-39.89%</td>
<td>-483.87%</td>
<td>-471.06%</td>
</tr>
<tr>
<td></td>
<td>▽ 400</td>
<td>Default</td>
<td>17.20%</td>
<td>6.10%</td>
<td>-6.11%</td>
<td>-190.71%</td>
<td>-154.84%</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>Fallback</td>
<td>-0.26%</td>
<td>4.10%</td>
<td>-2.14%</td>
<td>-1.24%</td>
<td>-1.40%</td>
</tr>
<tr>
<td>tradebeans</td>
<td>4000</td>
<td>Default</td>
<td>-4.05%</td>
<td>0.40%</td>
<td>3.63%</td>
<td>-84.21%</td>
<td>-39.75%</td>
</tr>
<tr>
<td></td>
<td>▽ 2000</td>
<td>Default</td>
<td>-8.35%</td>
<td>-4.64%</td>
<td>1.99%</td>
<td>51.56%</td>
<td>62.56%</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>Fallback</td>
<td>0.21%</td>
<td>-0.04%</td>
<td>-0.06%</td>
<td>0.87%</td>
<td>0.90%</td>
</tr>
<tr>
<td>xalan</td>
<td>1600</td>
<td>Default</td>
<td>0.77%</td>
<td>1.04%</td>
<td>3.91%</td>
<td>-1219.57%</td>
<td>-1346.11%</td>
</tr>
<tr>
<td></td>
<td>▽ 150</td>
<td>Default</td>
<td>2.39%</td>
<td>2.49%</td>
<td>0.01%</td>
<td>-408.42%</td>
<td>-367.79%</td>
</tr>
<tr>
<td></td>
<td>1600</td>
<td>Fallback</td>
<td>-0.71%</td>
<td>-0.11%</td>
<td>0.11%</td>
<td>-1.53%</td>
<td>-1.84%</td>
</tr>
</tbody>
</table>

Using compressed forwarding tables as a fallback seems to have no negative effects on throughput or latency, except for two small benchmarks in DaCapo. Neither Cassandra nor SPECjbb2015 see a statistically significant difference in performance. None of our attempts to improve mutator latency resulted in improvements over baseline CFW (Table 4).

As we argued in §3.6, our approach is likely to result in higher memory usage than using a hash table for applications with a low heap residency, which is confirmed in our evaluation. To increase heap residency, we re-ran DaCapo using a heap size just over the limit to avoid in-place relocation (Table 1). With higher heap residency, the effect of CFW on forwarding information is seen immediately (decreased in 7 cases, increased in 2).
Table 2: Performance evaluation for SPECjbb2015: ZGC vs. CFW.* Statistically significant. Non-compliant scores

<table>
<thead>
<tr>
<th>Configuration</th>
<th>max-jOPS</th>
<th>critical-jOPS</th>
<th>GC Cycles</th>
<th>Average Heap Usage (avg.)</th>
<th>Forwarding Usage (peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>6.89%*</td>
<td>5.71%</td>
<td>-17.22%*</td>
<td>-12.60%</td>
<td>-139.14%* -159.62%*</td>
</tr>
<tr>
<td>Fallback</td>
<td>7.66%*</td>
<td>8.73%*</td>
<td>1.76%</td>
<td>-2.73%</td>
<td>0.19% -1.12%*</td>
</tr>
</tbody>
</table>

Table 3: Cassandra evaluation

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Min</td>
<td>Max</td>
<td>Mean</td>
<td>Median</td>
<td>Max</td>
<td>Mean</td>
<td>Median</td>
<td>Max</td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>Default</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fallback</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Optimization evaluation in SPECjbb2015 for CFW.* Statistically significant. Non-compliant scores

<table>
<thead>
<tr>
<th>Optimization</th>
<th>max-jOPS</th>
<th>critical-jOPS</th>
<th>GC Cycles</th>
<th>Average Heap Usage (avg.)</th>
<th>Forwarding Usage (peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coalesce</td>
<td>-2.79%*</td>
<td>-8.37%</td>
<td>4.64%</td>
<td>0.06%</td>
<td>-0.57%</td>
</tr>
<tr>
<td>Partial: ordinal</td>
<td>-2.72%*</td>
<td>-4.49%</td>
<td>2.28%</td>
<td>-0.29%</td>
<td>-0.76%</td>
</tr>
<tr>
<td>Partial: vector array</td>
<td>-3.09%*</td>
<td>-12.17%</td>
<td>0.21%</td>
<td>1.13%</td>
<td>-0.67%</td>
</tr>
<tr>
<td>Partial: vector inline</td>
<td>-4.49%*</td>
<td>-13.44%</td>
<td>2.24%</td>
<td>1.39%</td>
<td>-98.58%* -98.39%*</td>
</tr>
</tbody>
</table>

5 RELATED WORK

In §2, we discussed how forwarding information is stored in C4, Shenandoah and ZGC. Beyond OpenJDK, the concurrent copying collector in Android Runtime [8] uses the from-space copy’s space. Neither V8 [9] nor .NET [15], supports concurrent evacuation. V8 uses the from-space object header for forwarding information and .NET uses free spaces on the heap. In this section, we focus on past proposals for computing forwarding information based on liveness.

We are aware of at least two prior works by Abuaiadh et al. [1] and Kermany and Petrank [14] which we describe in the following.

Abuiaiadh et al. [1] describe a design to compact the heap in parallel in a stop-the-world GC. They divide the heap into areas of at least 4 MB, and further divide each area into blocks. In their implementation they used 256 bytes as the size of their blocks. Three vectors are needed to calculate to-space addresses: a from-space livemap, a to-space livemap and an offset vector that holds the to-space address of the first object for each block. A lookup begins by calculating which block the address belongs to. If the address is corresponding to the first object on the block the address in the offset table is the to-space address, otherwise the ordinal number of the object is counted. The ordinal number can then be mapped to the bit set in the to-space vector to find the offset that should be added to the offset address in the offset table. Their algorithm supports in-place compaction and lazily maps from-space pages to to-space pages.

Kermany and Petrank [14] develop The Compressor, a concurrent GC that represents forwarding information similarly to our design and that of Abuaiadh et al. [1]. The Compressor divides pages into blocks. Each block has an associated address of the first to-space-object in that block, stored in an offset vector. Moreover, each page has also a from-space livemap, a first-object vector and a status table. The first-object vector maps the new to-space page referencing with the first object in from-space. The status table has a state (UNHANDLED, HANDLED, BUSY) for each to-space page. Kermany and Petrank [14] claim that the fact the allocation order is kept will keep The Compressor at or above the performance of a STW compaction algorithm. This is not fully supported when evaluating our design as performance suffers substantially in workloads like Cassandra. Only in some workloads will the performance benefit from this design. Kermany and Petrank [14] implement their design in Jikes RVM, a research-oriented JVM not necessarily optimized for performance. Moreover, The Compressor used an expensive page-trap to implement read-barriers. CFW instead relies on ZGC’s barriers which are implemented using a cheap user-mode barrier. Since traps are expensive they evacuate 8 pages, with a page size of 4096 bytes, per trap. This increases mutator latency compared to our 256 bytes. Using the same parameters as CFW, the overhead needed for The Compressor’s forwarding information is 6.25%, about three percent more than our design. The Compressor did not support in-place relocation as opposed to our design. Given that the designs are implemented on different JVM’s and they use different read-barriers, a performance comparison between The Compressor and CFW would not be useful.

6 CONCLUSION

We have argued that storing forwarding information in the from-space increases the memory pressure since pages cannot be reclaimed until all objects incoming pointers have been remapped.
Storing forwarding information separately leads to lower memory pressure since pages can be reclaimed sooner, when the from-space objects have been evacuated. However, pages with a large number of objects that need forwarding information can severely inflate the cost of forwarding information, in pathological cases causing an application’s memory requirements to grow 100%.

Our approach stores forwarding information at a lower memory bound in exchange for additional computation. As our approach upper-bounds memory requirements for forwarding information to 3.14% of the Java heap, it is considerably easier to deploy Java programs under tight memory constraints. Our design will also keep the allocation order of objects, which impacts performance of some workloads positively, others negatively and some not at all.

Our evaluation on-top of generational ZGC showed that it is possible to provide a guarantee for an upper-bound of the memory required for forwarding information without sacrificing performance in many workloads. Notably, sunflow shows a 54% speed-up and SPECjbb2015 shows a 7% improvement in its throughput score. In other workloads we observe a neutral impact or regression up to 4% in application throughput. For heaps with high residency we note mostly a positive/negative impact in application throughput. In Cassandra we can observe a substantial regression in latency and throughput. Peak latency and performance regresses with ~451% and ~127%, respectively. The source of regression for Cassandra is unclear and should be further explored in future work. Except for two workloads in DaCapo, neither uses a large heap, using CFW only in cases when a hash table would use more memory than our design does not introduce regression. None of the explored optimizations for mutator latency did result in a positive change.

ACKNOWLEDGMENTS

We thank our shepherd Richard Jones and the anonymous reviewers for their input to this paper. This work was supported by the Swedish Foundation for Strategic Research (SM19-0059).

A HELPER FUNCTIONS

The relocation lock coordinates threads concurrently relocating the same block.

Algorithm 5: The relocation lock

```
1 status ← wait_until_locked_or_relocated(F)
2 if status = locked then
3    relocate_block(B, P, T) // continue at Algorithm 2
4    mark_as_relocated(F)
```

The helper functions fst_page and snd_page ensure that to-space pages exist before blocks are relocated.

If the sought to-space page already exists, it is simply returned. However, if it does not exist, we must find it or create it ourselves. This requires coordination, as multiple tables commonly share the same to-page.

As we saw on Line 7 in Algorithm 1, each forwarding table knows the identity of the table responsible for creating its to_fst page. If no such table exists, or if we are looking for a to_snd page, we must create the page ourselves. Algorithms 6–8 show this logic.

Algorithm 6: Getting the first page

```
1 procedure fst_page(T) : P is
2    // T is a forwarding table, P is a to-page
3    if T.to_fst = null then
4        if T.orig = null then
5            T.to_fst ← last_page(T.orig)
6        else
7            T.to_fst ← new page
8        return T.to_fst
```

Algorithm 7: Getting the second page

```
1 procedure snd_page(T) : P is
2    // T is a forwarding table, P is a to-page
3    if T.to_snd = null then
4        T.to_snd ← new page
5    return T.second
```

Algorithm 8: Getting the last page

```
1 procedure last_page(T) : P is
2    // T is a forwarding table, P is a to-page
3    if T.page_break ≥ 0 then
4        return snd_page(T)
5    else
6        return fst_page(T)
```

BENCHMARK CONFIGURATION

B.1 DaCapo

We used an AMD Ryzen 9 3900XT and 128 GB, running Ubuntu 20.04.4 LTS (kernel 5.13.0-35). DaCapo benchmarks were started as depicted in Fig. 9 using the settings from Table 5.

B.2 SPECjbb2015

We used an AMD Ryzen 9 3900XT and 128 GB of memory, running Ubuntu 20.04.4 LTS (kernel 5.13.0-35). In Fig. 10 we show how we start SPECjbb2015. Running with the original ZGC the flag ZMaxOffHeap is removed since it is a CFW specific flag.

Figure 9: How DaCapo benchmarks are started
We show the raw numbers of our results in Tables 6 to 11.

We used a 2-node cluster, with one client with 128 GB of memory well with the number of Java threads. Best performance is achieved with only using one or two threads.

Figure 12 shows that h2 and tradebeans in DaCapo do not scale transparent huge pages are disabled.

Moreover, transparent huge pages are disabled.

The supplied compiler file is found in Fig. 17. The machines are running Oracle Linux Server 7.5 (kernel 4.1.12) and sysctl settings (see Fig. 11) are set according to https://docs.datastax.com/en/dse/5.1/dse-admin/datastax_enterprise/config/configRecommendedSettings.html. Moreover, huge pages are disabled.

We used Cassandra 4.0 beta-4 and the built-in tool cassandra-stress and executed it according to Fig. 13. In cassandra.yaml we changed concurrent_writes to 32. Fig. 15 lists all relevant parameters used. The supplied compiler file is found in Fig. 17. The machines are running Oracle Linux Server 7.5 (kernel 4.1.12) and sysctl settings (see Fig. 11) are set according to https://docs.datastax.com/en/dse/5.1/dse-admin/datastax_enterprise/config/configRecommendedSettings.html. Moreover, transparent huge pages are disabled.

C RESULTS IN RAW NUMBERS

We show the raw numbers of our results in Tables 6 to 11.

D SCALABILITY OF H2 AND TRADEBEANS

Figure 12 shows that h2 and tradebeans in DaCapo do not scale well with the number of Java threads. Best performance is achieved with only using one or two threads.

Figure 11: Settings in sysctl for Cassandra machine

B.3 Cassandra

We used a 2-node cluster, with one client with 128 GB of memory and one server with 256 GB of memory, both using two Intel Xeon E5-2630. We used taskset to ensure we only use one of the CPUs for the server, since CFW does not take NUMA into account. We used Cassandra 4.0 beta-4 and the built-in

Figure 12: Scalability of h2 and tradebeans.
Table 5: Settings used for DaCapo benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Heap sizing</th>
<th>Workload size</th>
<th>Heap (MB)</th>
<th>Iterations</th>
<th>Threads</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>normal</td>
<td>small</td>
<td>190</td>
<td>100</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>1200</td>
<td>30</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>huge</td>
<td>4000</td>
<td>15</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>small</td>
<td>130</td>
<td>100</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>450</td>
<td>30</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>huge</td>
<td>2000</td>
<td>15</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>small</td>
<td>130</td>
<td>100</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>tradebeans</td>
<td>normal</td>
<td>huge</td>
<td>4000</td>
<td>10</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>huge</td>
<td>2000</td>
<td>10</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>avrorra</td>
<td>normal</td>
<td>large</td>
<td>180</td>
<td>15</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>35</td>
<td>15</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>normal</td>
<td>default</td>
<td>160</td>
<td>200</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>default</td>
<td>80</td>
<td>200</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>jython</td>
<td>normal</td>
<td>large</td>
<td>400</td>
<td>50</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>80</td>
<td>50</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>luindex</td>
<td>normal</td>
<td>default</td>
<td>40</td>
<td>200</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>default</td>
<td>25</td>
<td>200</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>lusearch</td>
<td>normal</td>
<td>large</td>
<td>2800</td>
<td>50</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>100</td>
<td>50</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>pmd</td>
<td>normal</td>
<td>large</td>
<td>1200</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>120</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>sunflow</td>
<td>normal</td>
<td>large</td>
<td>1200</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>400</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>xalan</td>
<td>normal</td>
<td>large</td>
<td>1600</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>tight</td>
<td>large</td>
<td>150</td>
<td>100</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 6: Optimization evaluation in SPECjbb2015 for CFW ∗ = Statistically significant. Non-compliant scores as they are not verified with Student’s t-test

<table>
<thead>
<tr>
<th>Optimization</th>
<th>max-jOPS</th>
<th>critical-jOPS</th>
<th>GC Cycles</th>
<th>Average Heap (MB)</th>
<th>Forwarding Usage (avg. MB)</th>
<th>Forwarding Usage (max MB)</th>
<th>Cache misses (10^9 hits)</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline CFW</td>
<td>25868.07</td>
<td>21466.07</td>
<td>771.36</td>
<td>28827.03</td>
<td>1179.58</td>
<td>1693.73</td>
<td>3547.78</td>
<td>14</td>
</tr>
<tr>
<td>Coalesce</td>
<td>25146.80</td>
<td>19670.40</td>
<td>735.60</td>
<td>28809.04</td>
<td>1186.35</td>
<td>1700.80</td>
<td>3391.07</td>
<td>5</td>
</tr>
<tr>
<td>Partial: ordinal</td>
<td>25163.80</td>
<td>20220.70</td>
<td>753.80</td>
<td>28910.77</td>
<td>1188.55</td>
<td>1702.20</td>
<td>3491.08</td>
<td>10</td>
</tr>
<tr>
<td>Partial: vector array</td>
<td>25069.10</td>
<td>18852.80</td>
<td>769.70</td>
<td>28501.55</td>
<td>1187.54</td>
<td>1709.42</td>
<td>3521.48</td>
<td>10</td>
</tr>
<tr>
<td>Partial: vector inline</td>
<td>24705.80</td>
<td>18582.10</td>
<td>754.10</td>
<td>28426.98</td>
<td>2342.41</td>
<td>3360.18</td>
<td>3420.78</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 7: Performance evaluation for SPECjbb2015: ZGC vs. CFW. ∗ = Statistically significant. Non-compliant scores as they are not verified with Student’s t-test

<table>
<thead>
<tr>
<th>Configuration</th>
<th>max-jOPS</th>
<th>critical-jOPS</th>
<th>GC Cycles</th>
<th>Average Heap (MB)</th>
<th>Forwarding Usage (avg. MB)</th>
<th>Forwarding Usage (max MB)</th>
<th>Cache misses (10^9 hits)</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZGC</td>
<td>24200.88</td>
<td>20305.75</td>
<td>658.04</td>
<td>32984.08</td>
<td>0.52</td>
<td>0.68</td>
<td>3467.06</td>
<td>24</td>
</tr>
<tr>
<td>Default</td>
<td>25868.07</td>
<td>21466.07</td>
<td>771.36</td>
<td>28827.03</td>
<td>1.24</td>
<td>1.78</td>
<td>3547.78</td>
<td>14</td>
</tr>
<tr>
<td>Fallback</td>
<td>26053.78</td>
<td>22078.56</td>
<td>646.44</td>
<td>33883.20</td>
<td>0.52</td>
<td>0.69</td>
<td>3513.08</td>
<td>9</td>
</tr>
</tbody>
</table>
## Table 8: Performance evaluation for DaCapo: ZGC vs. CFW with a normal heap sizing

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Configuration</th>
<th>Time (s)</th>
<th>GC Cycles</th>
<th>Average Heap (MB)</th>
<th>Forwarding Usage (avg. MB)</th>
<th>Forwarding Usage (peak MB)</th>
<th>Cache misses (10^9 hits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>ZGC</td>
<td>20.49</td>
<td>4.88</td>
<td>57.23</td>
<td>0.83</td>
<td>1.16</td>
<td>8.91</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>21.36</td>
<td>5.09</td>
<td>52.75</td>
<td>1.19</td>
<td>1.65</td>
<td>9.16</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>21.50</td>
<td>5.12</td>
<td>55.58</td>
<td>0.82</td>
<td>1.16</td>
<td>9.14</td>
</tr>
<tr>
<td>fop</td>
<td>ZGC</td>
<td>0.11</td>
<td>1.03</td>
<td>44.47</td>
<td>1.52</td>
<td>1.47</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.11</td>
<td>1.01</td>
<td>53.39</td>
<td>1.14</td>
<td>1.04</td>
<td>2.67</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>0.11</td>
<td>1.01</td>
<td>51.94</td>
<td>1.91</td>
<td>1.83</td>
<td>2.52</td>
</tr>
<tr>
<td>h2 small</td>
<td>ZGC</td>
<td>0.66</td>
<td>1.00</td>
<td>177.89</td>
<td>0.42</td>
<td>0.46</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.65</td>
<td>1.00</td>
<td>75.88</td>
<td>1.03</td>
<td>1.05</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>0.66</td>
<td>1.00</td>
<td>174.42</td>
<td>0.43</td>
<td>0.47</td>
<td>2.15</td>
</tr>
<tr>
<td>h2 large</td>
<td>ZGC</td>
<td>19.46</td>
<td>2.45</td>
<td>787.89</td>
<td>7.37</td>
<td>9.90</td>
<td>30.21</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>19.70</td>
<td>2.81</td>
<td>727.41</td>
<td>15.07</td>
<td>22.13</td>
<td>31.02</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>19.30</td>
<td>2.33</td>
<td>806.00</td>
<td>7.32</td>
<td>9.69</td>
<td>30.24</td>
</tr>
<tr>
<td>h2 huge</td>
<td>ZGC</td>
<td>179.44</td>
<td>5.95</td>
<td>2678.08</td>
<td>27.31</td>
<td>37.57</td>
<td>146.76</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>180.48</td>
<td>10.77</td>
<td>2127.26</td>
<td>51.94</td>
<td>83.25</td>
<td>150.65</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>178.76</td>
<td>5.98</td>
<td>2679.98</td>
<td>27.39</td>
<td>37.68</td>
<td>147.77</td>
</tr>
<tr>
<td>jython</td>
<td>ZGC</td>
<td>8.04</td>
<td>40.60</td>
<td>214.56</td>
<td>0.44</td>
<td>0.82</td>
<td>5.69</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>8.00</td>
<td>40.74</td>
<td>211.73</td>
<td>0.79</td>
<td>1.26</td>
<td>5.63</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>0.98</td>
<td>40.58</td>
<td>214.59</td>
<td>0.41</td>
<td>0.76</td>
<td>5.69</td>
</tr>
<tr>
<td>luindex</td>
<td>ZGC</td>
<td>0.31</td>
<td>1.18</td>
<td>28.71</td>
<td>0.41</td>
<td>0.46</td>
<td>2.71</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.31</td>
<td>1.73</td>
<td>23.30</td>
<td>0.26</td>
<td>0.30</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>0.31</td>
<td>1.25</td>
<td>28.56</td>
<td>0.46</td>
<td>0.53</td>
<td>2.74</td>
</tr>
<tr>
<td>lusearch</td>
<td>ZGC</td>
<td>0.27</td>
<td>1.15</td>
<td>1606.56</td>
<td>4.60</td>
<td>5.10</td>
<td>4.87</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.27</td>
<td>1.18</td>
<td>1563.08</td>
<td>54.15</td>
<td>65.59</td>
<td>4.90</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>0.26</td>
<td>1.15</td>
<td>1591.58</td>
<td>4.59</td>
<td>5.10</td>
<td>4.99</td>
</tr>
<tr>
<td>pmd</td>
<td>ZGC</td>
<td>0.96</td>
<td>2.88</td>
<td>218.41</td>
<td>10.21</td>
<td>14.43</td>
<td>13.91</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>1.00</td>
<td>2.96</td>
<td>210.82</td>
<td>6.42</td>
<td>8.61</td>
<td>14.23</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>1.00</td>
<td>2.95</td>
<td>215.14</td>
<td>10.00</td>
<td>14.20</td>
<td>14.25</td>
</tr>
<tr>
<td>sunflow</td>
<td>ZGC</td>
<td>2.09</td>
<td>7.51</td>
<td>509.73</td>
<td>3.86</td>
<td>4.93</td>
<td>41.13</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.97</td>
<td>5.35</td>
<td>713.04</td>
<td>22.56</td>
<td>28.13</td>
<td>8.78</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>2.10</td>
<td>7.20</td>
<td>520.61</td>
<td>3.91</td>
<td>5.00</td>
<td>41.50</td>
</tr>
<tr>
<td>tradebeans</td>
<td>ZGC</td>
<td>54.20</td>
<td>10.30</td>
<td>2580.66</td>
<td>40.56</td>
<td>69.21</td>
<td>102.02</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>56.40</td>
<td>10.25</td>
<td>2486.87</td>
<td>74.72</td>
<td>96.73</td>
<td>110.66</td>
</tr>
<tr>
<td></td>
<td>Fallback</td>
<td>54.09</td>
<td>10.30</td>
<td>2582.13</td>
<td>40.21</td>
<td>68.59</td>
<td>101.33</td>
</tr>
<tr>
<td>xalan</td>
<td>ZGC</td>
<td>1.56</td>
<td>8.23</td>
<td>864.44</td>
<td>0.68</td>
<td>0.96</td>
<td>133.88</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>1.55</td>
<td>8.14</td>
<td>832.60</td>
<td>9.03</td>
<td>13.92</td>
<td>137.04</td>
</tr>
</tbody>
</table>
|           | Fallback      | 1.57     | 8.23      | 865.47            | 0.69                      | 0.98                      | 135.87                   

Figure 13: How the Cassandra benchmark is started

cassandra-stress user profile=./tools/cqlstress-insanity-example.yaml ops\{insert=3_simple1=7\} duration=20m no-warmup ← cl=ONE -log interval=10s hdrfile=./hdrs/${LOG}.hdr -pop dist=UNIFORM\{1..10000000\} -mode native cql3 -node $← {IP} -rate threads=100
Table 9: Performance evaluation for DaCapo: ZGC vs. CFW with a tight heap sizing.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Configuration</th>
<th>Time (s)</th>
<th>GC Cycles</th>
<th>Average Heap (MB)</th>
<th>Forwarding Usage (avg. MB)</th>
<th>Forwarding Usage (peak MB)</th>
<th>Cache misses ($10^9$) hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>ZGC</td>
<td>20.93</td>
<td>85.24</td>
<td>26.04</td>
<td>0.63</td>
<td>1.15</td>
<td>11.06</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>21.26</td>
<td>76.06</td>
<td>23.68</td>
<td>0.35</td>
<td>0.51</td>
<td>10.96</td>
</tr>
<tr>
<td>fop</td>
<td>ZGC</td>
<td>0.26</td>
<td>1.37</td>
<td>76.17</td>
<td>5.15</td>
<td>5.73</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.25</td>
<td>1.46</td>
<td>70.86</td>
<td>1.87</td>
<td>2.07</td>
<td>3.05</td>
</tr>
<tr>
<td>h2 small</td>
<td>ZGC</td>
<td>0.74</td>
<td>3.41</td>
<td>109.64</td>
<td>0.49</td>
<td>0.67</td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.72</td>
<td>3.55</td>
<td>107.09</td>
<td>0.52</td>
<td>0.73</td>
<td>2.33</td>
</tr>
<tr>
<td>h2 large</td>
<td>ZGC</td>
<td>19.83</td>
<td>17.14</td>
<td>399.34</td>
<td>4.54</td>
<td>7.40</td>
<td>31.76</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>19.88</td>
<td>17.53</td>
<td>393.37</td>
<td>2.91</td>
<td>3.88</td>
<td>32.19</td>
</tr>
<tr>
<td>h2 huge</td>
<td>ZGC</td>
<td>179.85</td>
<td>19.86</td>
<td>1779.39</td>
<td>36.83</td>
<td>65.52</td>
<td>150.45</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>179.46</td>
<td>21.02</td>
<td>1743.96</td>
<td>17.47</td>
<td>23.50</td>
<td>151.97</td>
</tr>
<tr>
<td>jython</td>
<td>ZGC</td>
<td>22.53</td>
<td>243.63</td>
<td>76.65</td>
<td>0.42</td>
<td>0.80</td>
<td>11.36</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>20.12</td>
<td>221.77</td>
<td>76.21</td>
<td>0.49</td>
<td>0.81</td>
<td>10.63</td>
</tr>
<tr>
<td>luindex</td>
<td>ZGC</td>
<td>0.45</td>
<td>4.04</td>
<td>23.84</td>
<td>0.48</td>
<td>0.76</td>
<td>3.03</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>0.37</td>
<td>2.90</td>
<td>22.16</td>
<td>0.25</td>
<td>0.31</td>
<td>2.95</td>
</tr>
<tr>
<td>pmd</td>
<td>ZGC</td>
<td>1.79</td>
<td>10.99</td>
<td>110.24</td>
<td>5.74</td>
<td>9.22</td>
<td>12.57</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>1.74</td>
<td>10.35</td>
<td>107.00</td>
<td>2.43</td>
<td>3.53</td>
<td>14.76</td>
</tr>
<tr>
<td>sunflow</td>
<td>ZGC</td>
<td>1.98</td>
<td>11.07</td>
<td>329.02</td>
<td>3.63</td>
<td>4.78</td>
<td>26.33</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>1.64</td>
<td>10.40</td>
<td>349.13</td>
<td>10.57</td>
<td>12.17</td>
<td>8.41</td>
</tr>
<tr>
<td>tradebeans</td>
<td>ZGC</td>
<td>53.101</td>
<td>145.07</td>
<td>544.54</td>
<td>17.50</td>
<td>33.16</td>
<td>105.80</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>57.54</td>
<td>151.80</td>
<td>533.70</td>
<td>8.47</td>
<td>12.41</td>
<td>118.61</td>
</tr>
<tr>
<td>xalan</td>
<td>ZGC</td>
<td>5.71</td>
<td>63.18</td>
<td>145.26</td>
<td>0.63</td>
<td>0.94</td>
<td>148.90</td>
</tr>
<tr>
<td></td>
<td>Default</td>
<td>5.57</td>
<td>61.61</td>
<td>145.25</td>
<td>3.19</td>
<td>4.40</td>
<td>149.49</td>
</tr>
</tbody>
</table>

Table 10: Cassandra performance evaluation

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Samples</th>
<th>Throughput (ops/sec)</th>
<th>Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Min</td>
</tr>
<tr>
<td>ZGC</td>
<td>37</td>
<td>2297258.21</td>
<td>209320.42</td>
</tr>
<tr>
<td>Default</td>
<td>37</td>
<td>2473361.08</td>
<td>215531.79</td>
</tr>
<tr>
<td>Fallback</td>
<td>37</td>
<td>2317938.63</td>
<td>206578.53</td>
</tr>
</tbody>
</table>

Table 11: Cassandra GC metrics evaluation

<table>
<thead>
<tr>
<th>Configuration</th>
<th>GC Cycles</th>
<th>Average Heap (MB)</th>
<th>Forwarding Usage (avg. MB)</th>
<th>Forwarding Usage (max MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZGC</td>
<td>139.53</td>
<td>69622.22</td>
<td>1075.23</td>
<td>1477.44</td>
</tr>
<tr>
<td>Default</td>
<td>2109.37</td>
<td>41208.80</td>
<td>1430.67</td>
<td>2782.78</td>
</tr>
<tr>
<td>Fallback</td>
<td>238.34</td>
<td>66701.94</td>
<td>1061.88</td>
<td>1523.25</td>
</tr>
</tbody>
</table>
Figure 14: JVM flags used for the Cassandra server

Figure 15: Supplied ${JPMS}

Figure 16: Supplied ${LIBS}
dontinline org.apache.cassandra.db.Columns$Serializer::serializeLargeSubsetSize (Ljava.util.Collection;ILorg.apache.cassandra.db.Columns;I)I
dontinline org.apache.cassandra.db.commitlog.AbstractCommitLogSegmentManager::advanceAllocatingFrom (Lorg.apache.cassandra.db.commitlog.CommitLogSegment;)

dontinline org.apache.cassandra.db.transform.BaseIterator::tryGetMoreContents ()Z
dontinline org.apache.cassandra.db.transform.StoppingTransformation::stop ()V
dontinline org.apache.cassandra.db.transform.StoppingTransformation::stopInPartition ()V
dontinline org.apache.cassandra.io.util.BufferedDataOutputStreamPlus::doFlush (I)V
dontinline org.apache.cassandra.io.util.BufferedDataOutputStreamPlus::writeSlow (JI)V
dontinline org.apache.cassandra.io.util.RebufferingInputStream::readPrimitiveSlowly (I)V
exclude org.apache.cassandra.utils.JVMStabilityInspector::forceHeapSpaceOomMaybe (Ljava.lang.OutOfMemoryError;)V
"newpage
line org.apache.cassandra.net.FrameDecoderWith8bHeader::decode (Ljava.util.Collection;Lorg.apache.cassandra.net.ShareableBytes;I)V
line org.apache.cassandra.service.reads.repair.RowIteratorMergeListener::applyToPartition (ILjava.util.function.Consumer;)V
inline org.apache.cassandra.utils.BloomFilter::indexes (Lorg.apache.cassandra.utils.IFilter.FilterKey;)[J
inline org.apache.cassandra.utils.BloomFilter::setIndexes (JJIJ)[J
inline org.apache.cassandra.utils.ByteBufferUtil::compare (Ljava.nio.ByteBuffer;[B)I
inline org.apache.cassandra.utils.ByteBufferUtil::compare (Ljava.nio.ByteBuffer;[B)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareUnsigned (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareUnsigned (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareTo (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareTo (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareTo (Ljava.lang.Object;JILjava.lang.Object;JI)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareTo (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.ByteBufferUtil::compareTo (Ljava.nio.ByteBuffer;Ljava.nio.ByteBuffer;)I
inline org.apache.cassandra.utils.VIntCoding::encodeUnsignedVInt (JI)
inline org.apache.cassandra.utils.VIntCoding::encodeUnsignedVInt (JI)
inline org.apache.cassandra.utils.VIntCoding::writeUnsignedVInt (JLjava.io.DataOutput;)V
inline org.apache.cassandra.utils.VIntCoding::writeUnsignedVInt (JLjava.io.DataOutput;)V
inline org.apache.cassandra.utils.VIntCoding::writeVInt (JLjava.io.DataOutput;)V

Figure 17: Supplied file for CompileCommandFile for Cassandra
REFERENCES


