Lazy Pointer Update for Low Heap Compaction Pause Times

Anonymous Author(s)

Abstract

To keep applications highly responsive, garbage collectors (GCs) try to minimize interruptions of the application threads. While pauses due to non moving GCs can be drastically reduced through concurrent or incremental strategies, compaction pauses remain a big problem.

A strategy to decrease stop the world compaction pauses is to compact subsets of the heap at any one time. But this only reduces the time spent in moving compacted objects, not the time spent updating all references to those objects, which may be significant in large heaps. In this paper, we propose to *only* move compacted objects during the compaction pause, replacing moved objects by low-overhead forwarding objects. References to compacted objects are lazily updated while the application is running and during the next GC marking phase, outside of the compaction pause.

We evaluate our technique on a suite of high workload (2 to 14Gb) benchmarks built from a real industrial application. Results show that not updating pointers during the compaction pause decreases the median pause up to 31% and the longest pause up to 71% on these benchmarks, while the forwarding objects slow down execution time without GC by no more than 1%.

Keywords Garbage collection, Memory management, Object oriented languages, Virtual machines

1 Introduction

Garbage collectors (GCs) are an integral part of modern mainstream virtual machines (VMs), such as those for Java and JavaScript, which typically host long running applications. A key challenge when designing and implementing a GC in these machines is how to keep the application highly responsive. Traditional stop-the-world GCs are inadequate since collections in large heaps may take so long that the program may not respond quickly enough.

Concurrent or incremental mark and sweep, and reference counting collectors [13, 19, 25, 31] keep the application highly responsive, interrupting the *mutators*[1] for very short times. Such GCs, however, do not move objects, leading to heap fragmentation [22], a slower allocation rate, and potentially bad locality, slowing code execution.

[1]We refer as *mutators* those application program threads which update objects as they execute code.

ISMM'19, June 23, 2019, Phoenix, Arizona, USA
2019.

To support efficiently long running applications, some form of partial compaction must be provided. Compaction algorithms with short mutator interruptions are, however, very difficult to implement efficiently. A compaction phase is typically composed of two tasks: (1) moving objects to compact in memory and (2) updating all references to the moved objects to their new location. A naive approach to lower compaction pauses, where only part of the heap is compacted, ends up requiring a full heap scan to update references to moved objects. The execution time of a full heap scan is proportional to the size of the heap, and therefore it is slow on large heaps. In order to keep the application responsive, two main approaches for compacting algorithms have emerged: mostly concurrent compaction [1, 10, 15–18, 32] and low stop-the-world compaction pauses [11, 29].

Mostly Concurrent Compaction. Concurrent compaction algorithms require the mutators to run while objects are moved in memory, leading to obvious data race issues. To solve these problems, the mutators usually require a read barrier [1, 10, 15–18, 32]. Each time a mutator reads an object in memory, the read barrier needs to perform extra computation to make sure it reads the right data.

Read barriers implemented in software are reported to incur significant execution time overheads. For example, the read barrier in Shenandoah [17] adds around a 40% execution time overhead, which can be lowered to 10-20% after read barrier specific optimizations performed by the Just-in-Time compiler (JIT). Alternatively, read barriers can be implemented with hardware support, either by using instructions not present in standard processors (x86, ARM) [10, 18, 21] or using virtual memory page protection [23, 32]. Choosing between hardware support and significant execution time overhead is draconian. As such some VMs instead implement compaction with small stop the world pauses.

Low Pause Stop the World Compaction. The main challenge of a GC with small stop the world compaction pauses is that references to moved objects in the whole heap need to be updated, which is expensive in large heaps.

Recent versions of Oracle's JVM feature a compacting GC as the default garbage collector, namely Garbage First [11]. Garbage First is a mostly concurrent Mark and Sweep algorithm over the whole heap, which in addition compacts part of the heap during the scavenger stop the world pause, concurrently with the scavenger. To work around the full heap scan, Garbage First splits the memory into regions and keeps track of inter-region references using efficient write

barriers [6]. This allows Garbage First to update references to moved objects during the stop the world pause by scanning only relevant references and not the full heap.

Our Solution. In this paper, we propose an alternative solution to decrease compaction pauses. During a stop the world pause, our GC compacts part of the heap and maintains a few invariants but does *not* update at all references to moved objects in the heap. Instead, while compacting the GC replaces the previous location of the moved objects by low-overhead software forwarding objects [26]. Thanks to the invariants maintained during the stop the world pause, the forwarding objects do not require the mutator to use a read barrier in most memory reads. The forwarding objects are then lazily removed by the mark phase of the following garbage collection, which can be performed incrementally or concurrently.

The contributions of this paper are:

- The design of a novel low pause stop the world compaction algorithm, where the references to compacted objects are updated lazily by the mutators and during the following garbage collection mark phase instead of during the compaction pause.
- The design of optimizations applicable to the existing forwarding objects implementation [26] to decrease their runtime overhead.
- The evaluation of our algorithm's compaction pause time, compared to an approach where the references to compacted objects are updated during the compaction pause, on benchmarks with heap sizes ranging from 2 to 14Gb.
- The evaluation of the overhead of the forwarding objects created by the algorithm on the mutator's execution time on the same benchmarks.

Outline. Section 2 states the problem, details our performance goals and summarizes the key aspects of the forwarding object implementation used in this work. Section 3 explains our solution, focusing on the compaction algorithm and how, subsequent to the work of Miranda and Béra [26], forwarding object support was improved to significantly reduce overhead. Section 4 evaluates our GC compaction algorithm against an emulation of a competitive approach, which updates references to moved compacted objects during the compaction pause. We then discuss the results, compare them with related work and conclude.

2 Low Pause Compaction

The main problem we tackle in this paper is:

► *How to build a low pause time compacting GC without hardware support that causes minimal mutator execution time overhead and scales to multi-gigabytes heaps?*

2.1 Performance goals

In what follows we set up our performance goals that we will later use to evaluate our algorithm and compare it to related work.

The first hypothesis to validate in this work is that *not* updating pointers to moved objects during the compaction pause significantly decreases pause times. In other approaches, the time spent updating pointers is very dependent on the number of inter-region references, which vary from one application to another. Our goal is to avoid pathological cases in multi-gigabytes heaps where the compaction pause is way longer due to pointer update. In these cases, a significant portion of the compaction pause is spent updating pointers. Moving pointer update away from the compaction pause is thus relevant. There is always a trade-off between performance goals, engineering time and maintainability of the system. Given the complexity of the algorithm (around 300 lines of code aside from performance improvements in free chunk management), we cannot move it to production in our VM without reducing the compaction pause time by at least 20% to avoid making the VM overly complex for the performance output. Our main performance goal is the following:

Main performance goal
Not updating pointers during the compaction pause should save at least 20% of the pause time on the worst pauses of multiple gigabytes industrial applications.

The second hypothesis of this work is that not tracking inter-region reference but using instead a partial read barrier will not slow down the mutator significantly. A very limited overhead is acceptable in exchange for high responsiveness. However, we do not want the runtime to be slowed by up to 40% as is the case in some concurrent compaction approaches [17] using a software read barrier. The overhead of fine-tuned write barriers, used to track down references in between memory regions in Garbage First to update efficiently pointers to moved objects during the compaction pause is usually up to 2% [6]. In our approach, we do not track inter-region references but we have to deal with forwarding objects. As such, the overhead of forwarding objects should be less significant than the overhead of fine-tuned write barriers used to track inter-region references, so under 1%. Our second and final performance goal is thus:

Side performance goal
Mutator execution time should be at worst 1% slower with our GC than its execution time with a GC not creating forwarding objects.

2.2 Low-overhead Forwarding Objects

In this paper, we build on the work of Miranda and Béra [26] on forwarding objects and their associated partial read barrier. In what follows, we introduce the key ideas of that work necessary to follow the contributions of this paper.

A *forwarding object* is an object storing a reference to another object; conceptually, when read by the mutator, the forwarding object redirects the reference to the other object. In that work, forwarding objects are implemented using a partial read barrier, *i.e.,* most memory reads do not require a read barrier to check if the object being read is a forwarding object, making them affordable without any hardware support. The key architectural ideas are:

- By making objects large enough to hold at least one field, all objects can become forwarding objects.
- Virtual calls (i.e. message sends to objects) provide a cheap implicit read barrier.
- The VM's fixed-sized stack zone allows for cheap bounded time scanning to eliminate a read barrier on instance variable accesses, which are the most common memory reads.

Object Representation. Object headers contain a class index (an index into a sparse table of classes) instead of a full pointer to reference the class. In addition each object has room for at least one field following the header. Hence any object can be converted into a forwarding object by changing its class index into the forwarding object class index, its slot size to 1, and setting its first field to a pointer to the object forwarded to. Method lookup caches, both first-level and inline, contain class indices not full class pointers.

Partial Read Barrier and Invariants. To intercept virtual calls to forwarding objects, the forwarding object class index is never entered in method lookup caches and look-ups to forwarding objects are never successful. When performing a virtual call, the VM checks if the receiver is a forwarding object only on cache miss and look-up failures, which are uncommon. By following[2] the forwarding pointer on the uncommon path, the receiver is guaranteed not to be a forwarding object at method activation time. Effectively virtual call binding provides an implicit read barrier.

To keep the read barrier partial, the VM maintains a set of invariants valid at all times except during the creation of forwarding objects. The key invariant is that on stack, the receiver of each frame is never a forwarding object. A second important invariant is that all methods references in method lookup caches are never forwarding objects. Thanks to these invariants, common operations other than virtual calls can also be performed without using a read barrier. For example, accessing instance variables of the receiver does not require a read barrier since the receiver is never a forwarding object.

[2]We call following a forwarding object the action of replacing a reference to the forwarding object to the object it forwards to.

Maintaining the invariants requires scanning only fixed-sized memory regions to follow forwarding objects: specific interpreter variables, objects related to process scheduling, fixed-sized caches and the fixed-sized stack zone. Its execution time is bounded, not proportional to any variable size structure[3] and especially not heap size.

Forwarding Object Deletion. Although forwarding objects are followed when met at runtime, some references may never be referenced and so never followed, potentially leading to forwarding objects accumulating and wasting memory. For this reason, the GC mark phase is aware of forwarding objects and deletes them. In the tri-color marking terminology [12], the invariant is that white forwarding objects always stay white. All forwarding objects live at the beginning of the mark phase are white, hence they are all deleted by the mark phase. When marking the fields of an object, the GC checks for forwarding objects and updates the fields holding them instead of turning forwarding objects grey. Since both the old field value and the value updated by the GC refer to the same object, they are both valid for the mutator (no synchronization is required). The mutator write barrier greys black objects when a white object is stored into them to maintain the invariant that no black object refers to a white object, hence it never greys white forwarding objects.

Implementation. Forwarding objects have been implemented in the OpenSmalltalk-VM[4], a VM for dynamic object-oriented languages in the Smalltalk family. This paper extends the basic architecture of forwarding objects [26] to design a low pause compaction algorithm. Our novel compaction algorithm is thus built in an extension to the OpenSmalltalk-VM.

Note that the VM's GC includes a generation scavenger. Since our work is focused on the old space collector, we do not discuss the scavenger and here-in we use the term GC to refer to the old space collector. In addition, old space is organized as a set of fixed-sized memory regions, aside from humongous regions which hold very large objects that could not fit in fixed-sized regions. Humongous regions are treated specifically and we omit discussing them in the paper.

3 Solution: Lazy Pointer Update

This section describes our compaction algorithm. The GC collects the heap through a mark phase, a sweep phase, and a compaction phase. The mark and sweep phases are assumed to be concurrent or incremental, with pauses to the mutator threads disjointed from the compaction pause. The section only details the compaction phase as it is the innovative one.

In a nutshell, our compaction algorithm compacts subsets of the heap and maintains the forwarding object invariants at each compaction pause. Our algorithm does not update pointers to moved objects during compaction but instead

[3]Except the number of running processes, which is usually low.

[4]https://github.com/OpenSmalltalk/opensmalltalk-vm

mutates the corpses of moved objects into forwarding objects. Forwarding objects created during the garbage collection are incrementally deleted at runtime when met, and guaranteed to be deleted at the end of the next mark phase. Compacted memory regions are effectively freed at the end of the mark phase of the next GC. By compacting small subsets of the heap, the GC is mainly a non-moving GC aside from small compaction phases used to defragment the heap.

3.1 Algorithm

Figure 1 shows the heap regions during the four phases of the algorithm which we detail in what follows. Due to the references to moved objects being updated in the following mark phase, full compaction requires two full garbage collections. This means the algorithm starts after the sweep phase of garbage collection number N and ends at the end of the mark phase of garbage collection number N+1.

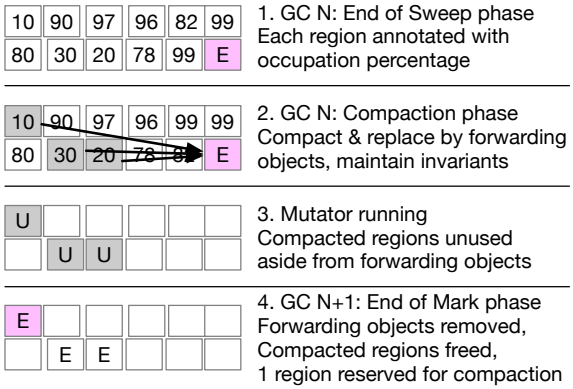

Figure 1. Algorithm Summary

Phase 1. End of Sweep Phase (Garbage Collection N). We extended the existing sweep phase to compute the occupation of each memory region once scanned. Figure 1 shows each region with its occupation as a percentage. In the common case, one region is empty and reserved from the previous garbage collection for the compaction phase, as detailed in the following paragraphs. This is depicted as a region E with a pink background.

Phase 2. Compaction Phase (Garbage Collection N). The GC computes which memory regions to compact based on its occupancy. Regions are selected from the least occupied up until enough regions have been selected to fill up a reserved empty region to compact into. Regions which are more occupied than a threshold (70% by default in our implementation) will not be compacted. If all regions are almost full, the GC does not compact an almost full region into the region to compact into, but instead skips entirely the compaction phase. The GC ensures a memory region to compact into was reserved, otherwise it allocates a new one, or it aborts compaction if the allocation of a new memory region is impossible.

Once the regions to compact are selected, the VM iterates over all entities in all these regions. Found free chunks are removed from the free chunk structures so they cannot be allocated any more. Found objects are "moved" to the region to compact into; an object is moved by creating a copy of it in the region and mutating the original into a forwarding object pointing to the copy. Last, the algorithm maintains the forwarding objects invariants.

Figure 1 shows that the algorithm selected the three least occupied regions (in gray) which are compacted into the reserved empty region (E with a pink background).

Phase 3. Mutator Running. At the end of the compaction phase, all the compacted regions now contain only forwarding objects and the free chunks present inside them cannot be allocated. The mutator can run without using the compacted memory regions, aside from forwarding objects which are followed when met at runtime. In the figure, these regions are shown with the letter U (unoccupied, aside from forwarding objects).

Phase 4. End of Mark Phase (Garbage Collection N+1). During the mark phase, the GC follows all forwarding objects (as explained in [26]). Memory regions marked as being compacted from the previous full GC can now be freed since they contained only forwarding objects, which are no longer referenced, and non allocatable free chunks. One of the unoccupied regions is reserved so that the next compaction phase can compact into it.

Figure 1 shows that one of the F regions, now empty, is reserved (E with pink background) while the two others are also freed (E), but can be directly re-used by the VM. Once the sweep phase terminates, the algorithm goes back to phase 1.

Implementation Details In our implementation at the OpenSmalltalk-VM, the first (lowest) memory region is never selected for compaction since it contains kernel objects that the VM expects not to move. In addition, some memory regions may be marked as pinned, *i.e.,* they include objects which were requested by the program run not to move in memory. Pinned regions are also never compacted by our algorithm.

3.2 Optimizing Forwarding Objects

The original implementation [26] suffered from a significant overhead in tight loops where each iteration of the loop requires to take a slow path to follow a forwarding object.

A good example would be a method computing the sum of all the values present in an array passed as an argument. Figure 2 shows such an example. The code simply initializes the local variable `sum`, iterates over the array and add the value of the current item of the array to the sum. Lastly, the method answers the sum computed.

The loop body is very simple and can be executed very quickly. The problem lies if the argument passed, `array`, is a

```
sum: array
  | sum |
  sum := 0.
  1 to: array size do:
    [:i | sum := sum + (array at: i)].
  ^ sum
```

Figure 2. Pathological Smalltalk Case

forwarding object. Then, at each iteration of the loop, when performing the virtual call `at:` on the array[5], the partial read barrier of the cache logic fails and the VM executes code through a slow path to follow the forwarding object. Since the loop body is normally executed quickly, adding the overhead of a slow path drastically slow-down performance.

When a virtual call to a forwarding object fails, the reference in the register to the forwarding object was always followed in the existing implementation, so execution was always correct. But, if the register is loaded from memory at each iteration of the loop (which typically happens in the template JIT), the value in memory also needs to be followed. To solve this problem, existing heuristics would check, in addition to the register, if the local variables of the current stack frame and the literals of the current method executed were forwarding objects, and follow them too to avoid repeatedly following the same object and slowing down the loop.

In this work, we extended the heuristics to check for forwarding objects in the arguments of the current frame, the receiver's instance variables and specific instance variables of literals (to avoid getting a forwarding object when reading a global for example). The new heuristics successfully eliminated a source of slow down.

Since all forwarding objects inducing overhead we found in the deployed applications we work on and the benchmarks discussed in the paper are covered by these new heuristics, we conclude that this optimization significantly reduced the overhead of forwarding objects. Such an optimization is the key in achieving our second performance goal, which we will further discuss in the validation section.

4 Validation

In this section, we first give an overview of the validation, specifically we describe what we are trying to compare. Second, we discuss the benchmarks we designed and built to evaluate our approach against our customer use-case. Then, we detail the methodology applied for the validation and our set-up. Next, we evaluate the distribution of compaction pauses on the benchmarks. We then show that forwarding objects do not impact execution time without GC. Lastly, we estimate the number of forwarding objects and references to them created during the GC compaction phase.

[5]In Smalltalk, `+` and `at:` are normal virtual calls, which are resolved by the look-up logic to primitive methods.

4.1 Overview

In this paper we claim both that not updating pointers during compaction pauses decrease significantly the pause times and that the overhead introduced by the forwarding objects of our approach is very limited. To validate these claims, we need to compare our approach against another approach which updates pointers during the compaction pause and does not introduce forwarding objects. Such a competitive approach is very similar to Garbage First [11]. We however built our approach on top of existing work [26], re-using the same framework, OpenSmalltalk-VM. There is no such a thing as a Garbage First implementation in OpenSmalltalk-VM, so we built a poor man's version of it. In the next two paragraphs we briefly describe the two approaches compared, our approach, that we call lazy pointer update, and the competitive, Garbage First style approach, that we call eager pointer update.

Lazy pointer update is our approach where part of the heap is compacted during the compaction pause, but references to moved objects are updated lazily outside of the compaction pause.

Eager pointer update emulates a Garbage First style approach, where part of the heap is compacted but the references to moved objects are also updated during the compaction pause. Pointer update is performed using a per memory region remembered table. The table holds addresses of references from other memory regions to objects inside the region. Since this algorithm is used only as a reference, the remembered tables are computed during the compaction pause, and the time spent for their computation is then subtracted from the pause. In practice they are implemented in Garbage First through fine-tuned write barriers that we could not easily emulate. We implemented eager pointer update in such a way because we believe it approximates a theoretical optimal pointer update phase in terms of execution time. Further details on the eager pointer update implementation are mentioned in Section 5.2.

4.2 Benchmarks Built and Used

The design of this GC algorithm is driven by the goal of decreasing garbage collection pauses in production applications using large heaps. In particular, the main motivating production application belongs to *fee*nk[6], a company building applications on OpenSmalltalk-VM using heaps up to 12Gb in production while needing a responsive environment.

Motivation. We first look for benchmark suites dealing with heaps of similar size. It turns out existing VM benchmarks and even memory management intensive suites such as Dacapo [5] do not deal with the kind of targeted workloads (2 to 14Gb). Benchmark suites dealing with large workloads

[6]https://feenk.com/

exist, such as Graph CHI [24], but the workload is on disk, not in RAM.

The Squeak speed center[7] evaluates on a regular basis the performance of OpenSmalltalk-VM to track down performance regressions through 90 benchmarks. However, the benchmarks focus on the quality of the code generated by the just-in-time compiler and the scavenger (young space GC), none of the benchmark trigger multiple old space collections. Hence, these benchmarks were not appropriate to measure our algorithm, designed for the old space collector.

We therefore decided to build open-source benchmarks from the *fee*nk industrial use-case. The benchmarks built in such way are at least representative of one industrial use-case and target workloads of the sizes we are interested in.

Benchmarks Built. Part of the *fee*nk business includes the analysis of deployed industrial software to help solve specific problems in that software using the Moose software analysis framework [14, 28]. The analysed software is parsed into a graph of objects (up to 12Gb currently in production). Analysis can then be performed interactively on the graph of objects through small scripts (hence the need for responsiveness). The graph of objects is released once the analysis ends to free memory.

We designed four main benchmarks, which can be used on different programs to analyse, *i.e.,* different workloads of different sizes.

Load: The Load benchmark imports the software analysed into a graph of objects.

Exp: The Exp benchmark expands properties, *i.e.,* it analyses the software with a frequently used set of analyses and caches some properties doing so.

ExpCache: The ExpCache benchmark is the same as Exp, but cached properties are already cached so they do not need to be recomputed and the heap is larger.

Release: The Release benchmark releases the graph of objects and requests three garbage collections.

Workloads Used. Since the benchmarks work with a program to analyse, the specific program analysed has an important impact on the benchmarks. Specifically, the size of the heap during the benchmarks is highly dependent on the size of the program analysed. In this paper, we ran the benchmarks with two Java projects, WildFly and NetBeans, which grow the heap from the base runtime of 300Mb to 3Gb and 14Gb respectively. Table 1 gives approximate workload sizes for the four benchmarks on the two projects.

Table 1. Approximate Workload sizes in Gb, at the start and the end of each benchmark.

	Load	Exp	ExpCache	Release
WildFly	0.3 → 2	2 → 3	3	3 → 0.3
NetBeans	0.3 → 8	8 → 14	14	14 → 0.3

[7] http://speed.squeak.org/

4.3 Methodology and Set-up

The benchmarks were run in parallel on 10 nodes of the same cluster, each node having the exact same specifications. Each node is a HP ProLiant DL20 Gen9, with a Intel Xeon CPU E3-1240 v5 (3.5 GHz) and has 32 Gb of RAM. The OS installed on each node is Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-133-generic x86_64). Although each node has four cores, we ran only one benchmark at a time on each node, to avoid exhausting available RAM and suffering paging side-effects.

The benchmarks were run on a custom compiled version of the OpenSmalltalk-VM, derived from version 2419 of VMMaker. The two differences compared to the open-source production VM is that we added (1) performance analysis features to compute exact pause times and (2) the eager pointer update algorithm. The lazy pointer update algorithm is in the production VM under the name "SpurSelectiveCompactor". Note that the OpenSmalltalk-VM features an interpreter and a template JIT, which provides strong baseline performance, but does not feature a JIT performing speculative optimizations. We discuss the side-effects of the lack of a speculative optimizer later in Section 5.4.

To obtain results, we ran each benchmarks for each configuration 30 times, 3 times on each node. Benchmarking with the NetBeans workload takes a significant amount of time (around 4 hours per benchmark, on an evaluation with 2 configurations that gives us a total of 240 hours total cpu time, 24h on each node).

4.4 Compaction Pause Time Experiments

In this experiment, for each benchmark we measured the compaction pause of each full garbage collection pause. In the context of lazy pointer update, the pause is composed mainly of the time spent compacting the selected memory regions, the rest (maintaining the invariants, etc.) represents a very small fraction of the pause. In eager pointer update, the pause includes in addition the time updating the pointers in the heap.

Measurements. We extended the GC for this experiment to store in a buffer the UTC[8] time in micro seconds at the start and the end of the compaction phase. In lazy pointer update, these two time stamps are the only values recorded. In eager pointer, the implementation used is required to compute a remembered table in between the phase where objects are moved and the phase where pointer are updated, as discussed further in Section 5.2. The time computing the remembered tables is irrelevant in the context of fine-tuned high performance write barriers which maintain such remembered tables. Hence, we compute four time stamps in this case to compute the compaction pause and subtract the time spent computing the remembered table from it.

[8] Coordinated Universal Time

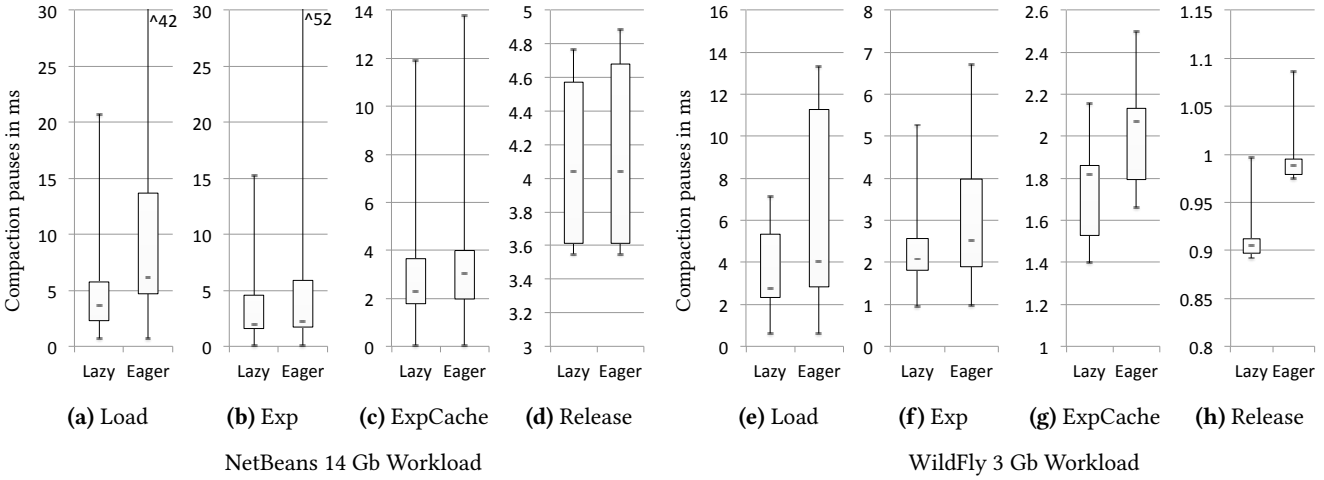

Figure 3. Compaction pauses

Table 2. Compaction pauses (All values in μs)

Workload	Benchmark	GC per run	Comp. per run	Pointer Update	First Quartile	Min	Median	Max	Third Quartile	Average ± Std Deviation
NetBeans	Load	23	19	Lazy	2292	679	3673	20697	5747	4779 ± 4163
				Eager	4691	679	6128	42318	13677	9418 ± 8761
	Exp	81	67	Lazy	1588	84	1929	15245	4563	3288 ± 2708
				Eager	1720	84	2194	52213	5886	4296 ± 4942
	ExpCache	96	88	Lazy	1776	51	2287	11894	3651	2872 ± 1873
				Eager	1973	51	3002	13751	3991	3491 ± 2330
	Release	3	3	Lazy	3613	3551	4040	4765	4572	4140 ± 471
				Eager	3613	3551	4040	4886	4680	4195 ± 524
WildFly	Load	12	12	Lazy	2326	603	2746	7133	5338	3530 ± 1838
				Eager	2832	610	3993	13347	11275	6315 ± 4336
	Exp	9	8	Lazy	1811	954	2082	5269	2566	2308 ± 1191
				Eager	1895	970	2536	6714	3981	3059 ± 1714
	ExpCache	5	5	Lazy	1528	1396	1816	2158	1861	1726 ± 192
				Eager	1794	1662	2071	2497	2133	1999 ± 203
	Release	3	2	Lazy	897	892	905	996	912	912 ± 24
				Eager	979	975	988	1086	995	998 ± 28

Results. The compaction pauses measured on the two workloads and the four benchmarks, for lazy and eager pointer update, are displayed in Table 2 and Figure 3. Table 2 shows for each benchmark the average number of full garbage collection per run. Since there were 30 runs, one needs to multiply that number by 30 to get the total number of pauses considered in the evaluation. A compaction phase starts during a full garbage collection only if at least one memory segment is less than 70% occupied. The next column in the table indicates the average number of compactions performed per run, which can be lower than the total number of full garbage collections. The following columns indicate the first quartile, min, median, max and third quartile of the compaction pauses measured. Figure 3 includes whisker charts drawn using these five values. Table 2 also includes the computed average pause and the standard deviation.

Analysis. In the Load and Exp benchmarks, the heap grows considerably. In this context it seems the number of inter-region references is higher and the difference between lazy and eager pointer update is large, especially in the context of the maximum pause and the third quartile. Growing to 8Gb and then to 14Gb in NetBeans shows maximum compaction pauses up to three times longer with eager pointer update. In the ExpCache benchmark, when the heap size remains stable, the number of inter-region references look lower and the difference between lazy and eager pointer update is minimal (up to 2ms difference in the worst case, median is up to 0.7ms worse). In the Release benchmark, inter-region references

are rare since the heap is shrinking from either 14Gb or 3Gb down to 300Mb. A few full garbage collections are enough to shrink the heap in these cases. Most segments are entirely empty, leading the first garbage collection to shrink the heap by freeing over a hundred segments, but some objects, including for example objects related to process scheduling remain. It takes therefore two full garbage collections to free these segments (they are freed at the end of the marking phase of the next garbage collection).

Discussion. In our context, we tried to find compaction pauses where the pointer update phase represents more than 20% of the compaction pause time, making it relevant to move that phase away from the compaction pause. In these benchmarks, with these workloads, not updating the pointers reduces the median pause up to 31% and the longest pause up to 71%.

In the WildFly workload, the Load and Exp benchmarks have worst pauses that are 47% and 22% smaller with lazy pointer update. In the Load benchmark, the median pause is even 32% smaller without updating the pointers. In the NetBeans workload, the Load and Exp benchmarks have worst pauses 51% and 71% smaller with lazy pointer update. The Load and ExpCache benchmark even have median pause 41% and 24% smaller without updating the pointers. A considerable amount of compaction pauses therefore benefit from lazy pointer update in these benchmarks, and we conclude that our first performance goal is met.

4.5 Execution Time Experiments

The original implementation of forwarding objects in [26] relied on forwarding objects to be uncommon to avoid overhead on execution time. In that work, forwarding objects were met at runtime exclusively when created from the Smalltalk *become* primitive. Since our algorithm creates a higher number of forwarding objects that we had ever seen before in production applications, we needed to re-evaluate the forwarding object overhead. As explained in the previous section, to decrease such overhead, we extended the VM so that when a forwarding object is followed, the memory location where the forwarding object came from (receiver instance variables, literals, arguments or temporary variables) is immediately followed.

Measurements. In this experiment, we ran our benchmarks with lazy and eager pointer update, but we compare execution time without GC instead of compaction pauses. We did not calculate the execution time of the Release benchmark since the benchmark only sets a variable to nil before running the garbage collector three times (execution time without garbage collection is always below 10μs and irrelevant). The production OpenSmalltalk-VM records by default the time spent in scavenges and full garbage collections. The results shown are computed from the total execution time, to which the total time spent in scavenges and full garbage collections have been subtracted.

Results. Table 3 and Figure 4 show the results. Median, worst and average execution time between eager and lazy pointer update are within 1% performance difference from each other. We therefore safely conclude based on these results that the higher number of forwarding objects has no significant impact on execution time of the application, excluding garbage collection time. As such, we also met the second performance goal.

4.6 Approximate Number of Forwarding Objects

In this experiment, we ran the same benchmarks as the previous subsection with the WildFly workload. We counted the number of forwarding objects created by the compaction algorithm (sum of the number of forwarding objects present at the end of each GC). We counted the number of old space collection including a compaction phase. We estimated the number of forwarding objects created by each compaction pause (Total number of forwarding objects created in the benchmark divided by number of old space collection with a compaction phase). We evaluated the number of objects on heap at the beginning and end of each benchmark. We counted the number of references updated at runtime through partial read barrier failures during the benchmark and counted the total number of references to update because they refer to forwarding objects created by the compaction algorithm (sum of the number of references to forwarding objects present at the end of each GC). Table 4 summarizes the results found.

Results. In the Load benchmark, 1.78 millions of forwarding objects are created across 12 GCs. 2.55 millions of references needs to be updated across the benchmark to avoid referencing forwarding objects. In the case of eager pointer update, the time spent updating these references is spread across the twelve compaction pauses. In the case of lazy pointer update, 628 thousands of references are updated lazily when met at runtime, while the rest of the references are updated through the next GC marking phase. The Load benchmark is specific because almost one quarter of the references are updated at runtime with our algorithm, lazy pointer update. In the two other benchmarks, most references to forwarding objects are unused at runtime: they are updated by the following GC marking phase before being accessed. It makes sense since for example in the ExpCache benchmark the heap contains around 21 millions objects and only approximately 8 thousands of them (0.038%) are converted to forwarding object on compaction pauses.

Execution Time Overhead. We estimated with a micro-benchmark that the time spent updating a forwarding object met at runtime to an order of magnitude of ten nanoseconds. It represents the time spent exiting the code generated by

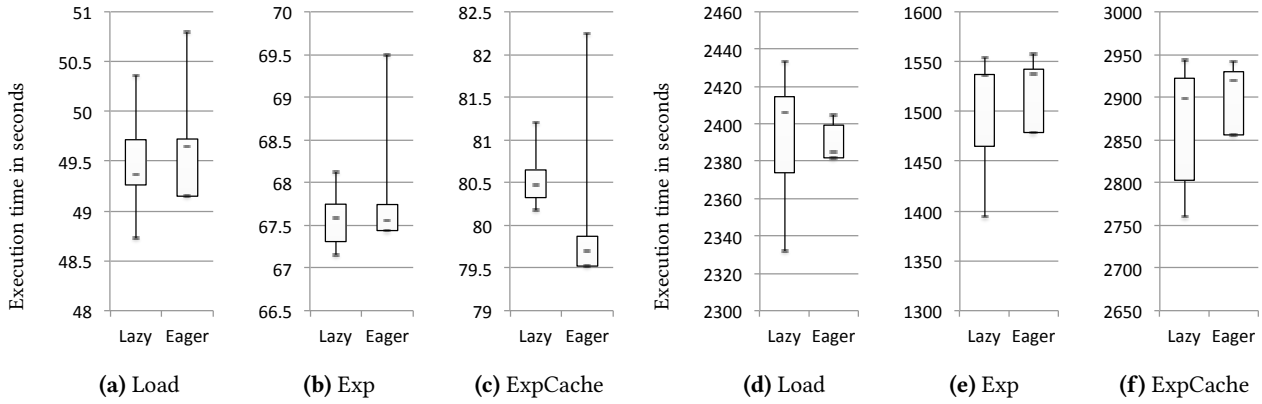

Figure 4. Execution time without GC

Table 3. Execution time

Workload	Benchmark	Pointer Update	First Quartile	Min	Median	Max	Third Quartile	Average ± Std Deviation
NetBeans Values in s	Load	Lazy	2373	2331	2406	2433	2414	2398 ± 29.6
		Eager	2381	2381	2384	2404	2399	2394 ± 9.40
	Exp	Lazy	1464	1394	1536	1554	1537	1509 ± 51.7
		Eager	1478	1478	1536	1557	1542	1532 ± 27.6
	ExpCache	Lazy	2802	2759	2898	2943	2922	2876 ± 63.4
		Eager	2855	2855	2919	2942	2929	2916 ± 31.4
WildFly Values in ms	Load	Lazy	49262	48723	49363	50361	49716	49560 ± 450
		Eager	49149	49149	49649	50796	49723	49560 ± 507
	Exp	Lazy	67309	67150	67592	68128	67747	67637 ± 294
		Eager	67438	67438	67550	69495	67743	67979 ± 700
	ExpCache	Lazy	80324	80169	80476	81197	80648	80617 ± 314
		Eager	79522	79522	79704	82254	79871	80418 ± 1130

Table 4. Number of Forwarding Objects Created and References Updated in the WildFly Workload (Average ± Std Deviation)

Benchmark	Number of Forwarding Objects Created	Number of Compacting GCs	Approx. Number of Forwarding Objects Created per Compacting GC	Number of Objects at Start-up (millions)	Number of Objects at Shut-down (millions)	Refs Updated at Runtime	Total Refs to Update
Load	1777670 ± 19768	12	148139	3	16	627909 ± 9706	2548602 ± 29793
Exp	119443 ± 21809	8	14930	16	21	2394 ± 951	120278 ± 22015
ExpCache	39027 ± 24256	5	7805	21	21	563 ± 1112	39027 ± 24256

the JIT to go to C code due to the forwarding object met, executing the C code to follow with heuristics fields of different objects to avoid looping repeatedly on the same forwarding object and lastly resuming execution of the code generated by the JIT. Even in the Load benchmark, where 628 thousands of references are updated lazily when met at runtime, the total execution time overhead of such updates is estimated to 6 milliseconds in a benchmark taking approximately 49.5 seconds and a standard deviation of 500 milliseconds. The overhead of lazy update is therefore negligible and hard to see in the total execution time without GC of the benchmark.

5 Threats to Validity

In this subsection we discuss specific concerns which may threaten the validity of our evaluation, and why we think the validation is still very strong. Specifically we detail the representativity of our benchmarks, the different issues and work-arounds in the eager pointer update implementation, the compaction heuristics and the interactions with the JIT compiler.

5.1 Benchmarks Representativity

The first threat to validity is how representative our benchmarks are to production applications. Given how we designed and build our benchmarks, they are representative of at least one industrial deployed application. However, few industrial users need both a responsive runtime and a large heap. The benchmarks built may not cover all the possible large production workloads. For example, another OpenSmalltalk-VM company is doing proofs on state machines, having multi Gb heaps holding mainly machine states in huge byte arrays. However huge objects are not moved by our GC (because they are typically the only object in their region) and byte arrays do not contain references to other objects, so this use case does not really stress the GC nor cause compaction pauses.

5.2 Experimental Eager Pointer Update

One of the main threat to validity of our evaluation is the implementation of the eager pointer update. We implemented it only as a reference for comparison in this paper. To make sure we implemented a fair comparison, we implemented multiple versions of the eager pointer update algorithm and evaluated our approach against the version with the lowest compaction pause.

First, we did not have the machinery in our VM to easily implement a remembered table through card marking. We implemented the remembered table first to hold objects referencing objects in other regions, then to hold directly pointers to the inter-regions references inside the objects. The second implementation requires a larger remembered table, but compaction pauses were much smaller so we kept it for the evaluation.

Second, we did not have the machinery in our VM to implement efficient write barriers tracking inter-regions references using bit masks (our memory regions are not aligned on high boundaries, etc.). We tried to implement a slow write barrier and we could evaluate the GC that way, but we could not evaluate the execution time without GC due to the massive slow-down. Instead, we implemented the solution described in Section 4.1, paragraph *Eager Pointer Update*.

5.3 Compaction Heuristics

Another threat to validity is the compaction heuristics that we employed. Recall that our algorithm picks memory regions to compact only based on occupation, *i.e.,* least occupied memory regions are the ones being compacted. In the context of lazy pointer update, we believe this is optimal since the number of references to moved objects does not impact the compaction pause. However, it matters in the case of eager pointer update. The high maximum pause time we have with eager pointer update in the benchmarks Load and Exp in the NetBeans workload come from large remembered tables. If the eager pointer update algorithm were to use different heuristics, for example, compacting the least occupied memory regions only if they have a small remembered table, the results may be different.

5.4 Allocation Folding in Optimizing JITs

The OpenSmalltalk-VM JIT compiler is a template-based JIT compiler, similar to the C1 Java hotspot compiler or to the FullCodeGen V8 JavaScript JIT compiler. It includes some optimizations a few which remove some object allocations in specific cases. However, JITs of VMs such as the Java HotSpot C2 compiler, or the JavaScript V8 CrankShaft/Turbofan compilers perform much more aggressive optimizations, which can lead in some methods to avoiding entirely the allocation of some objects in the common execution path. In this context, object demography might differ from what we observe and the impact on compaction pauses has yet to be confirmed as being similar. However, in practice such optimizing JITs remove mainly (if not only) allocations of short-lived objects, which do not escape an optimization unit. Short-lived objects usually never survive until they are moved to old space, so the impact on the old space collector should be minimal.

6 Discussion

In this section we discuss the applicability of our compacting algorithm to parallel compaction, to other virtual machines and a few other details worth mentioning.

6.1 Parallel Compaction

In our implementation, we implemented the compaction algorithm in a single thread since it is difficult in the OpenSmalltalk-VM to build it as a parallel compactor. Since the compaction phase that moves objects in memory is moving objects from one memory region at a time, it should be possible to parallelize it. In this case, the region to compact into has to be chopped into separate allocation buffers, each buffer's size being based on the occupation of the segments being compacted, so that objects can be moved without overlapping with each other. With parallel compaction, the compaction pause would theoretically be lower. However, parallel compaction can at most divide the compaction pause by the total number of threads. This is orthogonal to the problem we tackle in this paper, i.e. to avoid having compaction pause depending on scanning remembered tables of variable sizes. We believe that our results would apply in the context of parallel compaction.

6.2 Applicability to other Virtual Machines

In order to apply our algorithm in mainstream VMs such as JavaScript and Java VMs, the algorithm needs to be parallelised and work with an optimizing JIT (as discussed before). In addition, the algorithm relies on the partial read barrier to have efficient forwarding objects [26]. The original partial

read barrier has been implemented on all OpenSmalltalk-VM languages (Pharo [4], Squeak [3], Cuis, Croquet and Newspeak [7]). Although we also believe it is possible to implement such a design for other languages, it has yet to be proven.

6.3 Implementation Considerations

During the design and development of our lazy pointer approach we employed the production OpenSmalltalk-VM as our yardstick to compare our work. This allowed us to fix some pathological cases. Indeed, no sweep algorithm was in the production OpenSmalltalk-VM before this work, and the sweep algorithm had to be improved to deal efficiently with free chunk management. The compaction pauses are obviously way smaller in our approach than the existing compaction algorithm since it compacts the whole heap from the first (lowest) free chunk to the end of memory.

It might be also worth to note that without card marking and inter-regions references, our memory regions do not have a specific alignment requirement. In fact, regions are mmap'ed and freed at runtime by the VM though system calls based on need. The VM does not reserve any large chunk of memory aside from regions dedicated to huge objects and does not require mmap regions to have specific alignments.

7 Related Work

Previous research on compacting GCs has focused on either reducing the compaction pause or trying to make the compaction phase concurrent. In this section, we compare our approach to both families of approaches.

Although we focus our discussion on compacting GC approaches, other research has explored removing GC pauses through concurrent non-moving GCs (*i.e.,* mark and sweep and reference counting collectors [13, 19, 25, 31]).

7.1 Reducing the Compaction Pause

As previously mentioned, reducing the compaction pauses can be achieved by compacting part of the heap instead of the full heap. The challenge then is how to update references to moved objects without scanning the whole heap during the compaction pause.

Similar to our work, Yoav Ossia *etal.* [29] compacting pauses only require to move objects in memory and to maintain some stack invariants, but it is not necessary to update pointers. However, mutators suffer from a read barrier during execution, implemented with hardware support through virtual memory page protection. With this hardware implementation, the execution time does not suffer from the read barrier if no moved objects are read. In contrast, our partial read barrier implementation is purely software-based and does not require dealing with virtual memory page protection, which is in practice quite difficult to implement portably across OSs and processors (as we further discuss later). In both their and our work, references to moved objects are updated lazily and incrementally when met by the mutators during execution. However, in our work, references to moved objects are entirely eliminated during the next GC marking phase instead of a separate pointer update phase.

As mentioned before, Garbage First [11] splits the heap into many memory regions of the same size aside from regions holding huge objects. In contrast to our approach, each region keeps a remembered table through a card marking strategy, such that it knows where the references to objects inside them are. During the compaction pause, specific memory regions are compacted and the pointers to moved objects updated using the remembered tables. Since pointer update time can be significant, Garbage First selects regions to compact both based on occupation and remembered table size. To reduce further the compaction pause, Garbage First's compaction phase is parallel, and compacts selected regions of old space concurrently to the scavenger. There is no real old space compaction pause, there are instead scavenge pauses or mixed pauses (scavenge and old space compaction).

In contrast, our runtime does not need to maintain inter-region remembered tables through write barriers. Not maintaining such tables saves execution time, though fine-tuned efficient write barriers have usually less than 2% execution time overhead [6]. Our approach instead follows forwarding objects when met. We did not work on making our compaction phase be performed in parallel or concurrently to the scavenger, but we believe there is no major constraint which would make this impossible.

7.2 Concurrent compaction

Concurrent compaction solution often use read barriers like our approach. We now compare our partial read barrier to other related works in read barriers.

Software Read Barriers. The two most common read barriers are Brooks's forwarding pointers [8] and Baker's read barrier [2]. With Brooks' design, the first field of each object is reserved for a forwarding pointer. Each time an object is read, the first field of the object is read afterwards unconditionally, forwarding the reference either to itself or another object. This means that each read requires two reads, but the read barrier is unconditional, which is faster in some cases.

With Baker's design, each time a mutator reads an object, it checks if the address is in the range of object addresses being moved and if so, takes a slow path to read the correct object location. The read barrier has at least one branch, but in the common case it does not require an extra memory read. Baker-style read barriers usually have around a 20% overhead after various optimizations on the runtime [33].

Some GCs with concurrent compaction use Brooks forwarding pointers [1, 15, 17]. OpenJDK's Shenandoah [17] tries to mitigates Brooks forwarding pointer overhead through additional optimizations in the JIT which remove the read

barriers. Such additional optimizations reduce the read barrier overhead from up to 40% down to 10 or 20% depending on benchmarks. In the Chicken and Clover GCs [16], the mutator uses Brooks forwarding pointers in Chicken, while it uses a conditional Baker-style read barrier in Clover.

Software Write Barrier. P. Cheng and G. E. Blelloch [9] implemented a concurrent compactor inspired by Baker's read barrier [2], but based on the work of Scott Nettles and James O Toole [27]; they moved the invariant from the to-space to the from-space, allowing the use of a write barrier instead of a read barrier. The write barrier requires synchronization between mutator and GC threads to mutate the replica of the heap without concurrency issues, the write barrier is not just about setting a bit in a card marking table. The work of Richard L. Hudson and J. Eliot B. Moss [20] is similar, the Sapphire GC is able to avoid the read barrier through a write barrier with multiple synchronisations.

Hardware Read Barrier. In order to implement efficiently a read barrier, some approaches use hardware support using specific assembly instructions. In particular, Azul and IBM Pauseless GC [10, 18] relies on Z14 and Vega hardware support, respectively, to implement efficient read barriers.

IBM's Pause-Less GC [18] features concurrent compaction using forwarding objects which are very similar to ours memory wise (an object header and a pointer to the forwarded object in the first field). The main difference is that IBM's Pause-Less GC does not have to maintain any invariants upon forwarding object creation, allowing them to be created atomically, since the hardware supports writing the object's header and first field together atomically. The read barrier on forwarding objects is then implemented using Z14 processor guarded memory read instructions. The operation has the same latency as normal memory reads unless specific bits are set, in which case the operation fails and code dealing with a forwarding object is called instead. It is not clear how such techniques could apply to Intel or ARM processors.

Azul's Pauseless GC [10], featured concurrent compaction which features a "remapping phase" to traverse the object graph and modify stale references to the new address of those copied objects . While this phase is similar to our algorithm, we do not rely hardware support for the read barrier. Azul later developed Collie GC [21] features concurrent compaction using a form of hardware transactional memory supported by the Azul Vega architecture[9]. Azul's Collie GC is potentially compatible with x86 architectures using the transactional memory TSX instructions[10]. However, these instructions are disabled on many processors currently in use due to security issues and not present on older ones.

Instead of using specific assembly instructions, it is possible to implement read barriers on standard processors (like ARM and x86) using virtual memory page protection. The pages including objects that require a read barrier are read and write protected. The processor directly traps memory accesses to objects on the page, which can be handled at software level. Azul's C4 GC [32] features concurrent compaction on x86 and Linux using page protection to implement the read barrier. In theory, it should be possible to port it to other OSs and processors, but in practice it requires significant engineering effort. To support Linux and x86, the virtual and physical memory management of Linux for x86 was rewritten (OS changes) by the GC engineers to support the high rate of virtual memory operations required by the GC. Support for other OSs and processors would require a similar engineering effort. Aside from Azul's C4 GC, garbage collector in [23] also features concurrent compaction using page protection.

Constrained Software Implementation. Finally, concurrent compaction can also be implemented using special runtime assumptions. For example, Schism [30] is mainly a non-moving real-time GC, but specific data structures called array spines are managed by a concurrent compacting GC. Since the runtime has assumptions on how the array spines are accessed, it can perform a fully concurrent compaction without concurrency issues.

8 Conclusion

In this paper, we propose a new GC compaction algorithm which provides very small compaction pauses by compacting a subset of regions in the heap at any one time, by moving the pointer update logic out of the compaction pause and by following forwarding pointers lazily.

To the best of our knowledge, this is the first work to show a low pause compaction GC algorithm working with pure software-based forwarding objects which are kept efficient using a partial read barrier. The benchmarks conducted using workloads from 2 to 14 Gb showed that our algorithm reduces the median pause by up to 31% and the longest pause by up to 71%, while execution time ignoring GC slows down by no more than 1%.

References

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops.*

[2] Henry G. Baker, Jr. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* (1978). https://doi.org/10.1145/359460.359470

[3] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2007. *Squeak by Example.* Square Bracket Associates.

[4] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg,

[9] https://www.azul.com/products/vega-processor/

[10] http://software.intel.com/file/41604

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 *Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. https://doi.org/10.1145/1167473.1167488

[6] S. M. Blackburn and A. L. Hosking. 2004. Barriers: Friend or Foe?. In *International Symposium on Memory Management (ISMM 2004)*.

[7] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules As Objects in Newspeak. In *European Conference on Object-oriented Programming (ECOOP'10)*.

[8] Rodney A. Brooks. 1984. Trading Data Space for Reduced Time and Code Space in Real-time Garbage Collection on Stock Hardware. In *Symposium on LISP and Functional Programming (LFP '84)*. https://doi.org/10.1145/800055.802042

[9] P. Cheng and G. E. Blelloch. 2001. A parallel, real-time garbage collector. In *Programming Language Design and Implementation (PLDI '01)*.

[10] 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)*. https://doi.org/10.1145/1064979.1064988

[11] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. https://doi.org/10.1145/1029873.1029879

[12] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (1978). https://doi.org/10.1145/359642.359655

[13] Damien Doligez and Georges Gonthier. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In *Principles of Programming Languages (POPL '94)*.

[14] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. 2005. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*. Franco Angeli, Milano.

[15] David F. Bacon, Perry Cheng, and Vadakkedathu Rajan. 2003. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Principles of Programming Languages (POPL '03)*. https://doi.org/10.1145/640128.604155

[16] E. Petrank F. Pizlo and B. Steensgaard. 2008. A study of concurrent realtime garbage collectors. In *Programming Language Design and Implementation (PLDI '08)*.

[17] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. https://doi.org/10.1145/2972206.2972210

[18] C. Gracie. 2017. Pause-Less GC for Improving Java Responsiveness , Virtual Machine Meet-up talk (VMM'17).

[19] Harel Paz Hezi Azatchi, Yossi Levanoni and Erez Petrank. 2003. An on-the-fly mark and sweep garbage collector based on sliding view. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*.

[20] Richard L. Hudson and J. Eliot B. Moss. 2001. Sapphire: Copying GC without stopping the world. In *Java Grande (ISCOPE '01)*.

[21] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. 2012. The Collie: A Wait-free Compacting Collector. In *PInternational Symposium on Memory Management (ISMM '12)*. https://doi.org/10.1145/2258996.2259009

[22] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management. Section 7.3 Fragmentation.* Chapman and Hall.

[23] Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Programming Language Design and Implementation (PLDI '06)*. https://doi.org/10.1145/1133981.1134023

[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Operating Systems Design and Implementation (OSDI'12)*. USENIX Association.

[25] Yossi Levanoni and Erez Petrank. (2001). An on-the-fly reference counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems* ((2001)). https://doi.org/10.1145/504311.504309

[26] Eliot Miranda and Clément Béra. 2015. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *International Symposium on Memory Management (ISMM '15)*.

[27] Scott Nettles and James O'Toole. 1993. Real-time replication garbage collection. In *Programming Language Design and Implementation (PLDI '93)*.

[28] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. 2005. The Story of Moose: An Agile Reengineering Environment. In *European Software Engineering Conference Held Jointly with Foundations of Software Engineering (ESEC/FSE-13)*. 10. https://doi.org/10.1145/1095430.1081707

[29] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. 2004. Mostly Concurrent Compaction for Mark-sweep GC. In *International Symposium on Memory Management (ISMM '04)*. https://doi.org/10.1145/1029873.1029877

[30] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: Fragmentation-Tolerant Real-Time Garbage Collection. In *Programming Language Design and Implementation (PLDI '10)*.

[31] Elliot Kolodner Tamar Domani and Erez Petrank. 2000. A generational on-the-fly garbage collector for Java. In *Programming Language Design and Implementation (PLDI '00)*.

[32] 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)*. https://doi.org/10.1145/1993478.1993491

[33] Benjamin Zorn. 1990. *Barrier Methods for Garbage Collection.* Technical Report. Department of Computer Science, University of Colorado at Boulder.