Concurrent Compaction using a Field Pinning Protocol

Erik Österlund
Linnaeus University, Sweden
erik.osterlund@lnu.se

Welf Löwe
Linnaeus University, Sweden
welf.lowe@lnu.se

Abstract

Compaction of memory in long running systems has always been important. The latency of compaction increases in today's systems with high memory demands and large heaps. To deal with this problem, we present a lock-free protocol allowing for copying concurrent with the application running, which reduces the latencies of compaction radically. It provides theoretical progress guarantees for copying and application threads without making it practically infeasible, with performance overheads of 15% on average. The algorithm paves the way for a future lock-free Garbage Collector.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—memory management, garbage collection; D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming

Keywords Lock-free, compaction, garbage collection

1. Introduction

Assume a set of mutually referring objects allocated in a block of continuous heap memory. Over time, some objects become unreachable, hence *garbage*, while *mutator* threads still modify *live objects* and allocate new ones. Just deallocating the garbage for later reuse leads to *memory fragmentation*, i.e., memory blocks consist of live objects and free gaps. Robson [34] and [35] showed that the heap memory overheads due to these gaps could be a factor as big as $\frac{1}{2}log(n)$ where n is the ratio of the smallest and largest allocatable object. It is therefore understood that any long running application needs to deal with fragmentation.

To avoid fragmentation, *compaction* relocates live objects to contiguous heap memory regions. Compaction is used by *moving* Garbage Collection (GC) algorithms. They find live objects and relocate them, i.e., perform compaction. Then the unused memory (garbage) is reclaimed.

We distinguish logical objects from their physical memory locations referred to as *cells*; a fragmented heap memory region is called *from-space*, a continuous heap memory region is called *to-space*.[1]

There are two different actors in GC.[2] The *garbage collector*(s) and the *mutator*(s). The garbage collector finds live objects by scanning the stacks and globals for references to *root objects*, identifies other live objects by computing the transitive closure of these roots, and determines the *condemned set* of potential objects for relocation. The garbage collector compacts memory by *relocating* live objects of the condemned set: it *copies* the payload of cells in from-space to cells in to-space and then *remaps* incoming references, i.e., updates them to refer to the to-space cell. Finally it *reclaims* from-space. The mutators run some client application.

The present paper focuses on concurrent copying embedded in a *host garbage collector* responsible for root scanning, tracing, condemned set computation, remapping, and memory reclamation.

The garbage collector thread(s) (short GC threads) may run concurrently with the mutator thread(s). Then they need to synchronize copying of cells to prevent, e.g., that a mutator thread modifies a from-space cell while a GC thread has already relocated it to to-space. Synchronization might block access to objects being relocated, or even stop the world.

However, since the introduction of 64-bit processors, memory has quickly expanded and latencies of blocking compaction is an issue for an ever increasing number of application contexts from Big Data to (soft) real-time. Therefore, a concurrent *lock-free* solution is preferred.

We present a lock-free algorithm for copying objects concurrent with mutators using a Field Pinning Protocol (FPP). It provides a sequentially consistent view of objects being copied under concurrent mutator accesses. It is also the first such algorithm to provide copy progress guarantees: if the

[1] We inherited the terminology from Baker's algorithm [3] but do not assume the heap to be split into exactly two regions like a normal copying GC. Conversely, we assume it is split into many regions.

[2] Actually, we restrict ourselves to moving, tracing GCs, acknowledging that other GC approaches like reference counting exist.

ISMM'15, June 14, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3589-8/15/06…$15.00.
http://dx.doi.org/10.1145/2754169.2754177

GC picks N objects to be copied, it successfully copies at least $N - \epsilon, \epsilon$ is $O(t^2)$ objects, where t is the number of threads. Moreover, it does not rely on either safepoints or handshakes and is, hence, lock-free for both mutator and GC threads. This is an important step towards a fully lock-free GC dealing with fragmentation.[3]

Concurrent copying schemes without a stop-the-world synchronization phase cannot guarantee (at least with currently known techniques) the copying of any particular object while preserving all updates to the object. Fore example, the "Chicken" [31] collector simply gives up on copying an object that is being modified. Our idea is to re-assign responsibility for copying an object to another thread if that thread prevents the copying of an object due to trying to modify the object. Hazard pointers indicate that a thread may be trying to modify an object. Responsibility for copying an object may be bounced back and forth between threads, but with a theoretical upper bound to the number of "responsibilities" that may be in the process of being bounced back and forth at any given time.

Our compaction algorithm is not intrusive: it does not have any special requirements on the hardware or on the operating system (OS). It comes with a low memory overhead of a single (already existing) header word per cell, and is compliant with the Java Native Interface (JNI), which is crucial for integration into a real Java Virtual Machine (JVM).

We demonstrate the practical feasibility by integrating concurrent copying into the Garbage-First (G1) GC of OpenJDK [15]. The resulting GC comes with a 15% performance overhead compared to the original G1 run on the DaCapo benchmark suite, but with lower latencies. We achieve the performance by avoiding memory fences in the fast paths of memory access barriers, i.e. mutator loads and stores.

2. Field Pinning Protocol

The core idea of FPP is to let mutators pin the address of fields using hazard pointers [26] before accessing them. It is also possible to pin whole cells if desired (cf. Section 5.2).

2.1 Hazard Pointers

In concurrent programming, hazard pointers are used for manual memory management of objects in lock-free data structures. Hazard pointers and manual memory management, referred to as Safe Memory Reclamation (SMR), are needed today because automated GCs are not lock-free. Our approach is to solve the progress problems in GCs using concepts derived from SMR.

A hazard pointer is a thread-local pointer used to mark objects that a thread currently accesses. In general, each thread maintains a list of such hazard pointers. When a thread does not need access to an object any longer, it adds it to a list of removable objects, but does not actually deallocate it until no other thread has a hazard pointer to that object. Instead of synchronously deallocating the object, the threads asynchronously and collaboratively deallocate the object eventually when consensus is reached.

FPP uses hazard pointers for asynchronous and collaborative copying, not for deallocation. The GC thread tries to immediately copy a cell of a live object in the condemned set but, if preempted by a mutator, the mutator threads asynchronously and collaboratively copy the cell eventually when consensus is reached. Note that this consensus may for some cells never be reached.

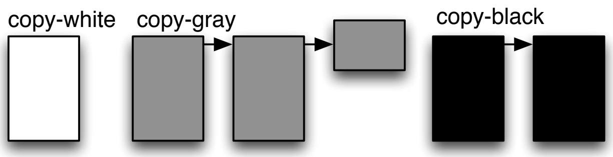

Figure 1. Colors of object during copying

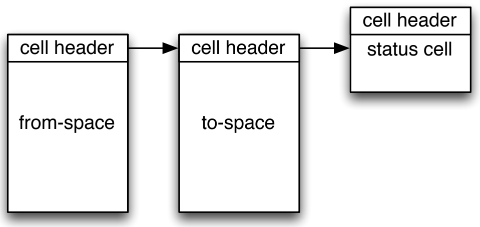

Figure 2. A single copy-gray object during copying

2.2 Object Copy-Tricoloring

During the relocation of an object, the object goes through different states with respect to copying. These states have a tricoloring: *copy-white*, *copy-gray* and *copy-black*.[4]

An object is copy-white before and after relocation; such an object is implemented with a single cell without a forwarding pointer. A copy-gray object is in the process of being relocated and consists of a from-space cell, a to-space cell and an additional status cell for storing status information needed for fine grained synchronization between mutator and GC threads. A copy-gray object may partially reside in from-space and partially in to-space. An object is copy-black if its payload resides completely in to-space and it awaits remapping. After remapping, the object becomes copy-white again and the relocation cycle is completed.

The cell layout of each object color is depicted in Figure 1; the cell layout of a copy-gray object is shown in Figure 2 in more detail. The from-space cell initially contains the payload of an object. Its header contains a forwarding pointer to a to-space cell to which the payload will ultimately be copied. The to-space cell header refers to a status cell.

[3] Note that fully lock-free GC also requires lock-free tracing and root scanning which is out the scope of the present paper.

[4] The terminology should not to be confused with Dijkstra's tricolor scheme [17] of objects during marking.

The transition from copy-white to copy-gray is made by allocating a cell in to-space and a status cell in a separate status space. The to-space cell is then connected to the status-cell by writing a pointer to it in its header. Then, the to-space reference is set in the from-space header using a Compare-And-Swap (CAS) operation. This indicates the color shading from copy-white to copy-gray. In one atomic operation, the to-space cell and status cell become reachable and the color change becomes visible for the mutators.

Once the whole cell has been copied to to-space, the copying thread (mutator or GC) removes the link to the status cell (shading the object copy-black) which may now be reclaimed using SMR. Therefore we reuse the hazard pointer pinning from-space cells. Any hazard pointer pointing inside a from-space cell means that its thread could still use the status cell. If there are no such hazard pointers, the status cell is deallocated immediately; otherwise it is deallocated asynchronously. Its memory is reused as the status cell for copying the next object if it could be immediately reclaimed.

Our copying goes from copy-white to copy-black without the use of any checkpoints, i.e., without safepoints or handshakes, making it the first copying algorithm to be lock-free both for mutator accesses and copying done by the GC thread. It does not need a handshake to install the forwarding pointer to the to-space cell because, if a mutator thread wants to pin an object field (for safe access) it first writes a hazard pointer to the from-space field and then finds out the copy color of that object. Especially, the mutator wrote its hazard pointer before it finds out the object is copy-white. This protects the accessed field from moving into to-space after its object has been shaded copy-gray by the GC thread. An implementation must carefully consider memory reordering as described in Section 5.4.

After copying, the host GC eventually remaps references. Once it finishes, the cell gets shaded copy-white again, meaning the object has been fully relocated.

2.3 Fine Grained Copying with FPP

To copy the payload (object field values) from from-space to to-space, the GC thread could split the from-space cell into its value words and move them to to-space one by one using FPP.[5] For optimization reasons, we almost always copy whole objects as discussed in Section 5.2.

The core idea of FPP is to use hazard pointers to count the mutators pinning the physical location of a field (or actually the word in which the field resides). The hazard pointers conceptually form a distributed counter. Its value is the number of hazard pointers pointing to the address of a word (in from-space). This conceptual counter constitutes a preemptive readers-writer lock where pins use a non-preemptive read lock and copying uses a preemptive write lock.

[5] The fine grained copying granularity should reflect the largest size of a field that can be accessed by mutators. In Java it is long which is 64 bits and hence that is the granularity of the pinning discussed here.

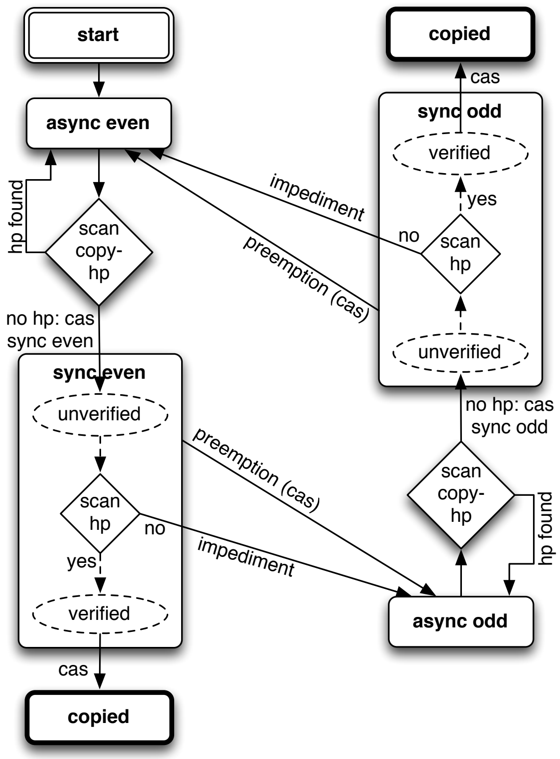

Figure 3. Copying a single word of memory

During copying, i.e. only in copy-gray objects, each value word goes through different *copy states*. These states must not to be confused with the copy-color of the object. They are depicted in Figure 3, starting in async even and eventually reaching the copied state. These states are stored in the status cell of the object: each value word in the object has a corresponding status byte in the status cell.

Each field mutation is first pinning the field address—either in from- or to-space—then mutating, then unpinning it again. As mentioned before, field-pinning and field-unpinning is implemented with hazard pointers and each mutator thread only holds one hazard pointer for pinning fields (as it can only access one field at a time). Hence, when it pins a new field for access, it implicitly unpins the previously accessed field by overwriting the hazard pointer. From the async even state until but excluding the copied state, it is always the from-space version of the field that is pinned.

For pinning a field (cf. Listing 1), the mutator first writes a hazard pointer to its address in from-space and then reads the copy color of the object.[6] If it is copy-gray, the copy state of the word is loaded, and maybe changed. The copy state is used for branching to different mutation and copying actions. The protocol supports any number of mutator and GC threads contending for pinning and copying. As discussed below, after pinning, the address of the field does not change

[6] We assume no compiler reorderings in the code listings and hardware reordering concerns are discussed in Section 5.4

and hence mutation is safe. The hazard pointer protects from copying the value of a field from from-space to to-space, which concurrently (or later) changes in from-space.

A mutator can preempt or impede an attempt to copy a value x and write a new value y to a field, which then gets eventually copied or copy-preempted by another mutator and so forth. As discussed below, we use CAS to transition from states before copying (consisting of sync and async states) to the successfully copied state. In order to avoid CAS ABA problems[7] while updating the states, we partition them into phases of different parity: odd and even. The async even and sync even states give equal guarantees as the corresponding async odd and sync odd states and their transitions are performed under equivalent conditions. However, all copy attempts of the even phase are finished before making a copy attempt with an odd parity, and vice versa, because the transitions from the async states to their respective sync states require that no thread pinned the word being copied.

Async states and their purpose: For copying, the GC thread goes through each cell, and each word of these cells. For each word, it writes a copy hazard pointer to it and sets the async state using CAS indicating an ongoing copy attempt. The purpose of the async state is to remember that copying eventually needs to be done. It does, however, not imply from-space is protected from mutations yet.

Sync states and their purpose: A sync state indicates ongoing copying and the from-space value word in this state must be protected from mutations, i.e. no mutator tries to mutate from-space, unaware of ongoing copying. If a mutator reads this state when pinning a field, then it tries to preempt the copy attempt and to transition to the async state (of the opposite parity), cf. line 21 of Listing 1. If it succeeds with the copy preemption, it uses from-space for mutations. Otherwise, the word was copied and it uses to-space.

Safely entering sync state: A transition to sync state is safe for a word if no other mutator pinned that word (for mutation or copying). Therefore, hazard pointers are scanned before and after this transition. The first scan, `scan1` (cf. line 7 of Listing 2) prevents an invalid sync state of a new parity while another copying thread is still in the copy attempt of previous parity. The first scan is therefore concerned with copy hazard pointers (cf. line 3 of Listing 2). After the first scan, the sync state is speculatively entered using a CAS of the status byte. Once the sync state is installed, any subsequent mutator pin will see that. However, mutators that pinned before the installation of the sync state could have seen the previous async state or even a copy-white object, and hence been missed. Therefore, another hazard pointer scan, `scan2` (cf. line 26 of Listing 2) is performed to verify that the sync state has been safely entered. This scan is concerned with field pinning hazard pointers (cf. line 3 of Listing 1). If a hazard pointer to the word is found in `scan2`, the copy attempt is impeded and a transition is made back to the async state (of opposite parity). Otherwise, it is guaranteed that all mutators are aware of the sync state. That means that any subsequent pin attempt tries preempting the copy attempt before trying to mutate from-space.

Attempt to enter copied state: Whoever brought the status byte to the sync state—any mutator or the GC thread—will copy the value word, *which is guaranteed not to have changed without preemption* since the sync state was verified. Once it has been copied, the copying thread attempts a transition to the final copied state using CAS. However, a mutator may try to preempt the copy attempt and install the async state (of the opposite parity) using CAS. In this race, one attempt wins, either copying or preemption. If copying wins, the mutator accesses the to-space. Otherwise, copying was preempted and the mutator accesses from-space.

Listing 1. Field pin operation

```
1  pin_field_addr(fromspace_cell, offset, thread) {
2    // Fast path
3    thread.pin_hazard_pointer = fromspace_cell + offset
4    tospace_cell = fromspace_cell.forwarding_pointer()
5
6    if (tospace_cell == null)
7      return fromspace_cell + offset // copy−white object
8
9    // Medium fast path
10   status_cell = tospace_cell.forwarding_pointer()
11   if (status_cell == null)
12     return tospace_cell + offset // copy−black object
13
14   // Slow path for copy−gray objects
15   help_async_copy(thread, fromspace_cell, offset)
16   status_byte_addr = status_cell + (offset >> WordShift)
17   status_byte_sample = *status_byte_addr
18
19   if (status_byte_sample ∈ {async even/odd} ||
20       (status_byte_sample ∈ {sync even/odd} &&
21       preempt_copying_word(status_byte_addr)))
22     return fromspace_cell + offset
23   else
24     return tospace_cell + offset
25 }
```

Listing 2. Fine grained copy

```
1  try_copy_word(fromspace_word_addr, tospace_word_addr,
2        status_byte_addr, parity, thread) {
3    thread.copy_hazard_pointer = fromspace_word_addr
4    status_byte = *status_byte_addr
5
6    if (status_byte == COPIED) return true
7    scan1 = scan_threads_copy_hp(fromspace_word_addr)
8
9    if (scan1.copy_hazard_pointer_count() > 0) {
10     // pinned by other thread
11     scan1.blame_self()
12     return false
```

[7] CAS assumes a value A has not changed but it actually has changed in a sequence of A, B, A.

```
13      }
14
15      // transition to unverified sync state
16      result = cas_bool(
17        address: status_byte_addr,
18        expect: ASYNC | parity,
19        new: SYNC | parity)
20
21      if (!result) {
22        return false
23      }
24
25      // Verify sync state
26      scan2 = scan_threads_pin_hp(fromspace_word_addr)
27      if (scan2.pin_hazard_pointer_count() > 0) {
28        install_new_async(status_byte_addr, opposite_parity(parity))
29        scan2.blame_scanned_threads()
30        return false
31      }
32
33      *tospace_word_addr = *fromspace_word_addr;
34
35      return cas_bool(
36        address: status_byte_addr,
37        expect: SYNC | parity,
38        new: COPIED)
39    }
```

2.4 Asynchronous Copying

A mutator may *explicitly preempt* or *implicitly impede* copying, explicitly by preempting the sync state (cf. line 21 in Listing 1) or implicitly when a copying thread scans for hazard pointers (cf. line 26 in Listing 2) written when pinning a field (cf. line 3 in Listing 1). However, it is important that the cells get copied eventually. Therefore, each mutator attempts copying as many words as it has copy-preempted or copy-impeded. To implement this, both the GC threads and the mutator threads "blame" a mutator that has preempted or impeded a copy attempt. The responsibility of copying then goes over to the blamed mutator.

In an initial round the GC thread attempts to copy cells directly after shading their objects copy-gray with blaming disabled; a mutator cannot be held responsible for impeding this copy attempt. The mutator might not even know that the object is in the condemned set or copying is going on as it found all objects in copy-white.

After this initial round, the GC thread tracks the set of copy-gray objects that failed to become copy-black. The GC thread then tries to copy these objects one more time, word by word, but this time blaming any mutator impeding copying. After these two copy rounds, the GC thread is done for the current GC cycle. We will show in Section 4.1 that there are at most $O(t^2)$, t the number of threads, copy-gray objects left; the others were shaded copy-black. Note that cells that are still not copied by any mutator before the next GC cycle starts are automatically part of the live condemned set of that next GC cycle which reinitiates their copying.

Mutators explicitly preempting copying blame themselves. Mutators whose copy attempt is implicitly impeded blame the impeding mutator(s).

Only mutators receive blames (never the GC thread). Mutators become aware of blames in an asynchronous copy helper (cf. line 15 in Listing 1). In order to not incur performance overheads in the fast path, this asynchronous copy helper is placed in the slow path, which is invoked if and only if the mutator finds the object in copy-gray. So, if the GC thread had blamed a mutator in the initial copy round, the mutator might never have found out.

This blaming technique allows bounded copy progress and is a major improvement compared to prior work on concurrent copying algorithms that lack such a guarantee.

Listing 3. Asynchronous Copy Information

```
struct async_copy_desc_t {
  cell_ref_t fromspace_cell
  integer word_offset
}
```

When a mutator explicitly preempts a copy attempt of a word it blames itself, i.e., it remembers this word for asynchronous copying in a thread-local blame variable of type `async_copy_desc_t`. It contains all information (cell and word offset) needed for later copying this word.

Likewise, when a mutator implicitly impedes a copy attempt of a word, it receives a blame: it is forced to remember this word in a thread-local blame set of type `async_copy_desc_t` (cf. lines 11 and 29 in Listing 2) to take responsibility for asynchronous copying of that word. Its memory is managed with SMR and hazard pointers. It is a copy-on-write set: to add a word to this set, the old set is first loaded, then the new word is added (if it is not already in the set) by creating a new set consisting of the old set and the new word, then it is installed back with CAS.

There are multiple producers but a single consumer of blames (many-to-one). Blames are consumed atomically, i.e. all words are attempted to be copied at once and then the set is cleared, rather than incrementally decreased.

The asynchronous copy helper will select all instances of `async_copy_desc_t` to be copied. Before it starts copying, it samples the parity of the copy states and then scans all threads for copy hazard pointers (cf. line 7 in Listing 2) to see if any other thread is copying these words already (to prevent ABA). If any other thread is currently copying a word, then it remembers these `async_copy_desc_t` descriptors in a thread-local pending copy attempt set to be considered next time, called *pending preemptions*. They need to be remembered because the threads found to be copying (with a copy hazard pointer) could have been preempted by the current thread, in which case the current thread is to be blamed.

If no other thread was found in the scan, the mutator calls `try_copy_word` of Listing 2 word by word, but with `scan1` ignored since the copy hazard pointers were already scanned for all words to be copied by the copy helper. The copy hazard pointer still prevents ABA-conflicts when transitioning from a sync state of previous parity to the copied state. Any number of threads may compete for reaching sync state of a certain parity, but only one thread succeeds. A new copy attempt after it gets preempted needs a new scan for copy hazard pointers which will not succeed until the copy hazard pointer from the first attempt is released.

Recall that we do not need to carry blames from one copying round to another since each such round reinitiates copying of previously uncopied cells. Therefore, we reset the blame sets at the beginning of each round and give them a unique round ID which is either even or odd, depending on whether it is an initial round with blaming disabled, or the revisiting round with blaming enabled. This way, blaming does never see a blame set of a previous round or an initial copy round. However, it may see a blame set of a future round. Then blaming is cancelled as this word is already taken care of by that future round.

2.5 Object Comparisons

The normal implementation of object equality checks if the addresses of two references are the same. We wish to support the equality operation when the heap is being concurrently remapped. Under these circumstances, two references can refer to the same object, but one is perceived as copy-white and another as copy-black during this remapping. The code in Listing 4 describes the solution.

If addresses `o1==o2` we trivially finish. If not, and either `o1` or `o2` is null they are not equal by definition. Otherwise, we know that either `o1` and `o2` are different objects or one of the references has not been remapped yet after successful copying of a cell. The forwarding pointers are examined and compared to account for this case too.

Listing 4. Object Equality

```
1  equals(o1, o2, thread) {
2    thread.hazard_pointer = o1
3    if (o1 == o2) return true
4    if (o1 == null || o2 == null) return false
5    o1 = o1.forwarding_pointer() != null ?
6      o1.forwarding_pointer() : o1;
7    if (o1 == o2) return true;
8    o2 = o2.forwarding_pointer() != null ?
9      o2.forwarding_pointer() : o2;
10   return o1 == o2
11 }
```

The purpose of writing the `hazard_pointer` on line 2 of Listing 4 is to prevent this object from being relocated. `o2` can freely relocate. This is safe, because if they refer to different objects, we return false and it does not matter that one of the objects has a reference that can not be relied upon as long as it is not the same object as `o1`.

3. Host Garbage Collector Requirements

Our compaction algorithm is general and can be used in different host GCs. It was designed, however, for tracing GCs that split the heap into more than one heap region. Since a bounded number of cells are allowed not to finish copying, a finer region granularity limits the memory overhead of such copy failures preventing memory reclamation. Our choice of host GC for the evaluation is the G1 GC in OpenJDK, but it is possible to integrate to Pauseless [13], Metronome [2] and similar schemes.

A description of host GC activities like stack scanning, heap tracing, etc. is outside of our current scope. However, for the sake of completeness of the discussion of compaction, we describe possible remapping and reclaiming approaches of host GCs. As we will see, remapping still impedes completely lock-free GC. Conversely, if remapping (and all host GC activities) were lock-free, FPP would not impede their progress and the GC would remain lock-free.

3.1 Reclaiming Memory

A simple algorithm for reclaiming from-space cells in a condemned region is to add the region to a free list after all relocations are finished and there are no hazard pointers to the from-space region in a similar manner as SMR. Regions containing cells from copy-gray objects can not continue their life cycle. The to-space region may not be added to a new condemned set until all relocation to it has finished, and from-space regions may not be added to a freelist until all relocation from it has finished. The smaller the granularity of freeable memory in the GC, the smaller the impact is of objects failing to evacuate.

One alternative is segregated free lists like used in, e.g., Metronome [2]. This allows even finer granularity of memory reclamation where a whole region is not punished by a single cell not being copied; the surrounding memory may still be freed. Our own choice of algorithm for the evaluation is the simpler approach, integrated into G1 (which lacks segregated free lists at the moment).

3.2 Concurrent Remapping

Remapping involves updating all references in the heap and roots after copying. It could be done either lazily during tracing [13], after copying [2] or explicitly after copying using per-region remembered sets [15]. FPP is agnostic to whichever approach and supports remapping of the heap references (as opposed to thread roots) concurrent with mutation (e.g. during concurrent tracing). Whenever all references to an object are remapped, it is shaded copy-white (from copy-black).

The only assumption made so far is that at the instant we pin a field of a root object (actually when writing the hazard pointer at line 3 in Listing 1), this field reference is valid pointing to a field in either from-space or to-space. Any on-the-fly GC guarantees this invariant. But on-the-fly GC relies on a handshake with mutators to scan the stack for roots of the next GC cycle and to remap copied roots of the previous cycle, i.e., mutators give the permission to safely remap roots in a GC checkpoint before a from-space cell can be reclaimed. Then it can start a new GC cycle. Because of this handshake, remapping is not lock-free. As of writing the current paper, no on-the-fly nor any other tracing GC that we know of has lock-free progress guarantees.

However, our FPP should handle lock-free GC if it eventually existed. No handshakes in between the GC cycles means that a root can be valid before pinning, i.e., before making GC aware of a mutator's attempt to access a field by writing the hazard pointer. Then the mutator gets delayed; meanwhile GC and relocation moves and reclaims the from-space cell. The root reference is now invalid without GC nor mutator being aware of it.

Even if lock-free GC could relocate objects at any point in time, any number of times, FPP could handle this by recognizing and "healing" invalid references. If objects referenced from the thread root set may be relocated and reclaimed at any point in time, the problem boils down to writing a hazard pointer consistent with the root that could potentially become invalid in the field pin barrier (cf. line 3 in Listing 1) and when comparing objects (cf. line 2 in Listing 4). This can be achieved with the pseudo code in Listing 5, at the price of performance. The mutator installs the hazard pointer with a CAS and keeps track of the last hazard pointer written. Then, between every GC cycle, the hazard pointers of every mutator gets tagged by the GC thread using CAS. Any hazard pointer writes that are not aligned with the current GC cycle will get trapped by a failing CAS. Once an invalid root is trapped, the hazard pointer is untagged and then the thread roots are healed, e.g., using a lock-free shadow stack as described by Kliot et al. [24].

Listing 5. Stable Hazard Pointer Write

```
stable_hazard_pointer_write(root_ref, field_offset, thread) {
  while (true) {
    candidate_ref = load_root(root_ref)
    result = cas_bool(
      new_val: candidate_ref
      old_val: thread.old_hazard_pointer
      address: &thread.hazard_pointer
    )
    if (result) {
      thread.old_hazard_pointer = candidate_ref
      return candidate_ref
    }
    heal_thread_roots()
  }
}
```

Since no lock-free host GC exists, we will for the remainder of the paper assume a host GC that cannot arbitrarily relocate and reclaim cells without making sure it is safe (without the performance cost of our simple proof-of-concept algorithm in Listing 5), like an on-the-fly GC.

4. Progress Guarantees

It was an important design goal to theoretically bound the amount of memory that cannot be copied and to not impede lock-free progress guarantees of neither mutators or GC.

4.1 Copy Progress Guarantees

In order to guarantee *copy progress*, we need to show that if a set N of cells are selected for copying by the GC thread in a GC cycle, $|N| - \epsilon$ are guaranteed to be copied when the GC cycle is completed. The bound ϵ is the number of object unable to become copy-black. We will now show that this bound ϵ is $O(t^2)$, where t is the number of threads.

Recall that the GC thread revisits cells U_1 that could not be copied after trying to copy all cells in the initial round. In this initial round, the mutators may impede copying but not yet asynchronously help copying, cf. Section 2.4, since they may find the objects that are to be copied in copy-white. Hence, $|U_1| \leq |N|$.

For the revisit round over the cells in U_1, all mutators find the objects in U_1 in copy-gray and pin them using the slow path. Therefore, they help copying them when they preempt or impede copying. Still some of these copy-gray objects may remain copy-gray even after this revisit round. This set is denoted by U_2 and should be bounded by ϵ.

Given that the asynchronous copy helper will remember all blames, what remains to be shown is that the number of remembered blames are never more than ϵ. To do this, we must look closer at the mutator interactions during the revisit round.

Each thread i has at most one blame due to explicit preemption EP_i, a blame set of implicit impediments II_i and a set of pending preemptions PP_i.

Between every two pins, hence, between possible sources of blames, the II_i set is atomically cleared. At this linearization point [21], there is at most one blame for previous pinnings per thread other than i in execution (at most $t - 1$). No thread can have more than one such blame in execution since it cannot blame and be impeded at the same time.

However, since in order to get to the asynchronous copy helper, a copy-gray cell is first pinned, that one extra cell could cause an impediment before reaching the copy helper. Hence, the maximum size of the implicit impediment set is $|II_i| \leq t$ between any two pins of two copy-gray objects.

After the asynchronous copy helper is finished, all the asynchronous copy descriptors have been dealt with, except the set of pending copy attempts blocked by copy hazard pointers of other copying threads. The size of this pending preemption set $|PP_i| < t$ since there are only $t-1$ such copy hazard pointers to prevent copying.

Therefore, the worst case size of the ongoing asynchronous copying is $\epsilon = \sum_{i=1}^{t} |EP_i| + |II_i| + |PP_i| \leq t(1+t+t-1) = 2t^2 = O(t^2)$.

Now that there is a well defined maximum bound of cells that can not be copied, this allows us to, using the results of Bendersky and Petrank [6], calculate the maximum fragmentation bounds of the heap with a host GC either using bounded compaction, e.g., Detlefs et al. [15], or a segregated free-list approach, e.g., Bacon et al. [2].

Note that in case of never ending copying, there is no "live lock" – all threads, both mutators and GC, continue running. The algorithm is invariant of whether a bounded number of cells cannot be copied. Instead such never ending copying is merely observed as *bounded* memory overheads.

4.2 Mutator Progress Guarantees

Mutator progress concerns performing computations and heap memory accesses. Our copying algorithm lets mutators execute all the time and does not enforce any GC checkpoint. The field pinning barriers can always preempt any on-going copying and continue running. The copy helper never blocks and is also lock-free. Therefore, the mutator progress with respect to our copying algorithm is lock-free.

4.3 Garbage Collector Progress Guarantees

GC progress concerns the progress of reclaiming memory and depends on the host GC.

One part of the GC progress concerns copying objects during compaction. If the host GC adheres to the requirements of Section 3, then the host GC is never blocked by our copying protocol; it can start copying a cell whenever, without an initiating handshake. Instead of requiring the host GC to either reclaim the full condemned set or nothing, we allow it to reclaim most of it (with a bound) and then continue, leaving the rest of the task to be finished asynchronously.

Other aspects of GC progress concern root sampling, tracing transitive closure, and remapping. The progress guarantees of these tasks are outside the scope of the current paper. However, we would like to point out that had they been lock-free, then FPP would not impede their progress guarantees and hence the progress of the overall GC solution.

5. Implementation

Practical feasibility was an important concern in the design of FPP. To demonstrate this, FPP has been implemented for Java in the Hotspot JVM of OpenJDK 9 for the x86_64 architecture. This JVM is used for evaluation (cf. Section 6). It also offers a JNI interface which retains the expected data layout of objects.

5.1 The Host Garbage Collector

We integrated FPP with the Garbage First (G1) GC [15]. G1 splits the heap into many regions. Each region has a remembered set consisting of all incoming references. The condemned set consists of the young generation regions and the regions with the lowest liveness, calculated in a periodic concurrent marking traversal. Live cells in the condemned set are relocated by first relocating the roots consisting of the remembered sets, thread stacks and various globals, then relocating its transitive closure (in the condemned set). All this is done incrementally in safepoints.

We tried to transparently blend in with the approach, but with concurrent copying using FPP after the initial pause. We make no claims that the root scanning, tracing or remapping of the host GC are lock-free; in fact they are not.

Adding concurrent copying to G1 led to a few anomalies to the original algorithm. Since the mutator may now concurrently mutate the objects during tracing in the condemned set, a Yuasa snapshot-at-the-beginning (SATB) reference pre-write barrier [40], already emitted for concurrent marking, is activated during relocation of the condemned set to make sure all objects that were live at the snapshot when tracing started get relocated.

Another such anomaly is that the reference post-write G1 barrier used for maintaining the remembered sets which is normally disabled for the whole young generation must now be activated for parts of the young generation. The reason is that references in survivor spaces and the new concurrently allocated extension of the young generation (which was not in the condemned set but allocated since then) can receive references to cells in the condemned set while it is being evacuated. They must be eventually remapped. The disabling of this barrier is implemented by having a special card value for the young generation, making the post-write barrier skip ahead. In our implementation, this card value is not written for young regions allocated during relocation, but the optimization is still allowed for the young eden regions allocated when relocation is inactive.

While tracing, the GC tries to remap edges from predecessors whenever possible i.e., to objects that could be shaded copy-black. Meanwhile, the concurrent refinement threads responsible for maintaining the remembered sets, remap any references to copy-black objects before adding them to a remembered set if necessary. Finally, a safepoint is issued to flush the SATB buffers and remaining remapping of references in the remembered sets, i.e. the references that were added to copy-white or copy-gray objects. Then regions with no remaining copy-gray objects are reclaimed, while remaining regions that could not be reclaimed remain in the condemned set of the next GC cycle.

5.2 Coarse Grained Pinning and Copying

A coarse grained variant of FPP is supported that optimistically pins a whole cell if possible using a special cell pinning hazard pointer. If the object is determined to be copy-gray, then the operation fails. For an operation such as array copy, a slow path using fine grained FPP is invoked instead if it fails. Note that by pinning a whole cell, it is understood that this could come at a potential cost of having to help copying the whole cell asynchronously. There are actually multiple cell hazard pointers to speed up array copying and to keep multiple JNI external cells at the same time.

The GC thread does not have to copy cells word by word. The GC may copy cell by cell instead. The GC thread marks the whole cell as being copied by writing a copy hazard pointer to its header word. Then it performs the first scan and advances all the status bytes to sync state. As many status bytes as possible are changed at a time, limited by the size of the largest CAS available on the platform. On x86_64 16 status bytes (and hence 2 cache lines worth of value words) have their statuses updated at a time. If there was no intervention, the whole cell is then copied normally (using SSE acceleration and write combining if available). Similar to before, the status bytes are then advanced to the copied state 16 at a time.

To make this optimization possible, the status-cell size is aligned accordingly so that the object payload starts at a 16 byte aligned memory address, and the size of the status cell is aligned up to a 16 byte size. At a negative offset, memory is reserved for displaced cell headers during copying.

As a matter of fact, the GC does not even have to scan hazard pointers for every cell being copied. It can instead be done once every n cells as long as a copy hazard pointer is pointing at each cell being copied. This optimization becomes useful when copying many small objects.

The granularity at which memory is pinned can also be chosen arbitrarily (instead of words), but with the described optimizations, copying was no longer a performance bottleneck. Since the mutator may have to cooperate with the copying, it was intentional to make the copying granularity of FPP small and constant.

5.3 Java Native Interface

To allow JNI code having complete control over the raw data of objects (e.g. `GetPrimitiveArrayCritical`), the whole cell needs to be pinned. The mutator first attempts to use coarse grained pinning. If unsuccessful it copies every remaining word and eventually shades the object copy-black using FPP. This is a blocking operation as it could need to wait for some fields to become unpinned. Entering a critical JNI section is already expected to be a blocking operation and, hence, blocking copying here does not violate expected progress guarantees. However, such calls to JNI no longer impede the progress guarantees of the global system, only the mutator thread entering the critical JNI section.

The idea of pinning objects to support JNI is not new; it can be found in the Immix GC in Jikes RVM [7]. They use the sticky mark-bits algorithm [14] to support GC progress when cells can not be copied; we use asynchronous copy progress instead: the regions are remapped and reclaimed eventually when all cells have been relocated. The GC may continue to progress without reclaiming all regions.

As far as we know, this contribution is the only lock-free solution to fragmentation offering full JNI support. Previous approaches either changed the object layout or had no way of safely manipulating raw memory of objects without barriers for every mutator access.

5.4 Memory Ordering of Field Pin Barrier

The fast path of the mutator pin barrier uses hazard pointers for synchronization. Traditionally, this requires an expensive fence between the hazard pointer store (cf. line 3 in Listing 1) and forwarding pointer load (cf. line 4 in Listing 1), because they may be reordered by the hardware. This reordering could lead to races resulting in a mutator considering a cell copy-white before scanning begins, but the GC finds no hazard pointers to the word being copied as the store is delayed. It is fine to use fences, but our implementation optimizes this.

To handle the race efficiently, we employed a technique resembling Asymmetric Dekker Synchronization (ADS) [16], used for the implementation of fast locks. Instead of an expensive StoreLoad fence, the GC thread issues a system-wide store barrier before scanning hazard pointers, guaranteeing that if the store and load reorder, they *both* happen before the hazard pointer scan begins.

While there are multiple approaches for such a global store barrier (e.g. split cacheline locked instructions on x86, FlushProcessWriteBuffers on windows, cross calls, timed quiescence, etc.), we chose the arguably most portable and reliable variant: to exploit the guarantees of mprotect[8]. It requires the issue of a system-wide store serialization event using Inter-Processor Interrupt (IPI) messages, forcing a store buffer flush on remote processors. Note that we do not depend on memory protection per se, only on the store serialization properties of mprotect calls when write-protecting a dummy page owned by the GC thread.

Since the GC batches copying of n cells at a time, reusing the same hazard pointer scan for all n cells to be copied, the GC thread amortizes the synchronization overhead and, hence, its cost is negligible.

StoreLoad fences are only triggered when entering the slow path of our barrier in order to allow efficient asynchronous copying when the second round of copying starts with blaming enabled. It allows mutators to copy without batching, and without relying on ADS. In practice, the slow path is so rarely encountered that this cost is also negligible.

5.5 Memory and Performance Overheads

What lock-free solutions of handling fragmentation have in common is that they come at a performance cost. Our algorithm is no different and requires both read and write barriers for mutator accesses.

[8] Even though implementations of mprotect may and do in practice use locks in the kernel, these are *non-preemptive* locks that can not be interrupted by scheduling. Therefore, perhaps counter-intuitively, calling mprotect is lock-free as one of the threads is guaranteed to progress.

To make this practically feasible, the following were the design goals of the fast path of the barriers: no fencing instructions like `lock cmpxchg`, as few instructions as possible, very unlikely slow-path, no cache interference and no special hardware or OS requirements.

The field pinning barrier constitutes 4 instructions (for x86) in the fast path which is taken whenever a cell is copy-white (cf. Listing 6). The first two instructions calculate the field address in the from-space cell and writes it to the hazard pointer, resp. The next instruction checks if the object is being copied (i.e. not copy-white) based on the parity of the bit pattern of the header status word. The last instruction branches to a medium fast path if that is the case. In the current implementation, this bit pattern is 00 for a locked object and 11 for an object being copied, both with even parity. Pattern 00 is a false positive branch to the medium fast path for locked objects.

Listing 6. x86 code for the fast path of a pin operation

```
lea $OFFSET(%r_obj_ref), %r_field
mov %r_field, $HP_OFFSET(%r_thread)
test $HEADER_OFFSET(%r_obj_ref), $FORWARD_MASK
jp SLOW_PATH_STUB
```

The medium fast path starts by verifying that the cell was indeed not copy-white with a single instruction (`je`), which will take the mutator back to normal execution if not.

Then the medium fast path then checks for copy-black objects. Only if the cell is partially copied, i.e. copy-gray, a (leaf) call to the runtime is made.

Contrary to Brooks' read barrier [10], there is no need for an extra cell header word. The normal header is displaced to the last cell in the chain, including the status cell.

5.6 Relaxed Loads

We currently use the same field pin barrier for both loads and stores. The read barrier could also optimistically assume objects are copy-white without writing the hazard pointer. This allows loading the value in from-space first and then verifying it by reading the header, reverting to pinning only if necessary (it was not copy-white) in a medium fast path placed in a code stub. This way, the field address calculation and hazard pointer store can be omitted. Consequently the fast path barrier reduces to only two instructions for loads.

5.7 Limitations

So far only the `-client` mode JIT-compiler c1 (and the interpreter) is supported with limited support for inlined intrinsics (e.g. `sun.misc.Unsafe`, arraycopy). Full support for the `-server` mode c2 compiler is a work in progress. Also, due to engineering issues, support for biased locking and fast locking is currently unavailable. This is not an inherent limitation, but an engineering issue due to the poor maintainability and documentation of the locking protocols used in hotspot.

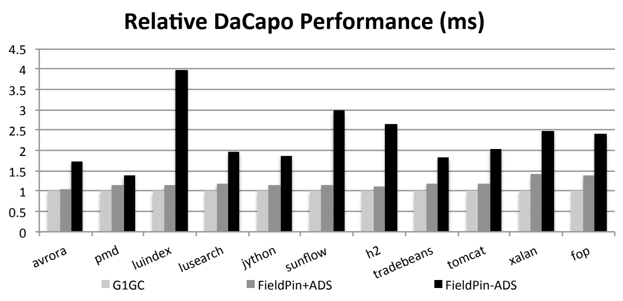

Figure 4. Normalized running times of DaCapo with Garbage-First GC with and without concurrent compaction

Table 1. DaCapo running times in milliseconds.

Bench	G1GC	FieldPin+ADS	FieldPin-ADS
avrora	3850	4081	6674
pmd	2601	2984	3581
luindex	893	1011	3549
lusearch	2236	2620	4376
jython	7541	8579	14175
sunflow	5233	5976	15630
h2	6380	7123	16962
tradebeans	7842	9263	14276
tomcat	4441	5259	8974
xalan	1767	2494	4405
fop	579	796	1385

6. Evaluation

The DaCapo benchmark suite [8] is a standard benchmark for measuring performance of real-world applications. In order to evaluate our solution, we used DaCapo 9.12. We could run all benchmarks that the original JVM could run, i.e., all except eclipse and batik. In addition, we excluded tradesoap because it exhibited unpredictable running times for the G1 system with or without our algorithm varying by a factor of 2 for every iteration.

The benchmarks were used for measuring performance and latency of the JVM and its execution environment. Experiments were run on a MacBook Pro with 2.7 GHz Intel Core i7, 4 cores, 16 GB 1600 MHz DDR3 and Mac OS X 10.10. It has 256 KB L2 cache and a shared 6 MB L3 cache.

6.1 Performance DaCapo

Figure 4 shows the normalized running times of the benchmarks compared to the original G1 solution, after 10 warmup iterations and with 512MB heap using the client c1 JIT compiler. We chose to compare our implementation to the original G1 GC of OpenJDK since it is our host GC. Both JVMs run with the same limitations: no fast locking nor biased locking or inlined intrinsics for array copying and `sun.misc.Unsafe`.

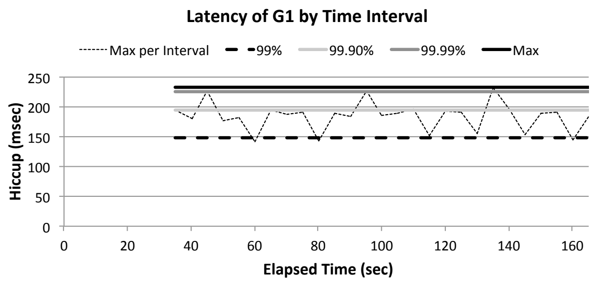

Figure 5. DaCapo h2 hiccups with -XX:+UseG1GC

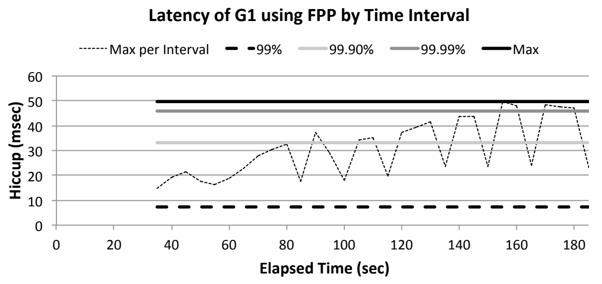

Figure 6. DaCapo h2 hiccups with our modified G1

The bars for `G1GC` show running time of the JVM using G1 defined as 1 and `FieldPin+ADS` shows the relative running time when running with our concurrent compaction algorithm and ADS enabled. It batches 100 kB or a maximum of 1024 objects to be copied at a time.

The average loss in throughput for all benchmarks is 15.2%. It is expected that this gets worse on a faster JVM.

The `FieldPin-ADS` bars show the running times when the synchronization costs have been moved to the mutators instead. This implies a memory fence for each hazard pointer write. It is easy to see that this is expensive to do, which motivates why we avoid doing that. The average throughput loss with this technique is 53.0%

The absolute running times can be seen in Table 1.

6.2 Latency DaCapo using jHiccup

To record the latency of our JVM, we used jHiccup, a tool from Azul Systems designed to measure the latency of the execution environment in which a Java application runs, rather than the application itself. This includes GC hiccups, noise produced by OS scheduling, JIT compilers, etc. It records the response times of the environment in certain time intervals while running the application. For each interval, it plots one dot in the line chart corresponding to the worst response time in that interval.

The resulting charts from running 25 iterations of the h2 benchmark from DaCapo can be seen in Figure 5 for normal `-XX:+UseG1GC` execution and Figure 6 when using our concurrent compaction strategy in G1 instead. Both JVMs run with 8 GB heap, `System.gc` disabled and no fine tuning to make the comparison interesting. We chose h2 with a larger heap because it was the most memory intense benchmark, making latency issues more apparent.

Note that the tool automatically cuts off the beginning of the curve, due to the warmup phase of the JVM during which JIT-compilers cause extra noise which is not interesting when looking at the latency of a long running application. The horizontal lines show the worst case for different percentiles of response times. The top line shows the overall worst case response time recorded and the other lines the worst case considering 99% (99.9%, 99.99%, resp.) of the recorded response times, i.e. excluding 1% (0.1%, 0.01%, resp.) of the globally worst response times.

As can be seen, the latency has significantly improved. The hiccups are no longer due to copying but almost exclusively due to host GC activities like remembered set maintenance, as well as root sampling and remapping. Conclusively, by using concurrent compaction and FPP, the latency was greatly reduced for every percentile, most notably from 150 ms to 8 ms for 99%.

Similar improvements can be observed for the other less memory intense benchmarks. Table 2 shows the average GC evacuation pauses while running the other DaCapo benchmarks with 10 iterations and 512 MB heap (excluding explicit user triggered GC). The GC pauses were measured as the time from before a GC safepoint is acquired until after the safepoint is released.

Table 2. Average GC pauses of DaCapo benchmarks.

Bench	G1GC	FieldPin+ADS	Difference
avrora	6,88 ms	2,02 ms	-70,6 %
pmd	40,82 ms	5,02 ms	-87,7 %
luindex	7,98 ms	4,87 ms	-38,9 %
lusearch	2,73 ms	2,72 ms	-0.4 %
jython	2,50 ms	1,87 ms	-25.2 %
sunflow	6,33 ms	2,91 ms	-54.0 %
tradebeans	31.73 ms	11.81 ms	-62.8 %
tomcat	12,31 ms	5.48 ms	-55.5 %
xalan	4,35 ms	1,51 ms	-65.3 %
fop	37.39 ms	13.22 ms	-64.6 %

7. Related Work

Low latency GC: Improving the latency of GC systems has been an optimization goal for a long time. It started with the on-the-fly GCs by Steele Jr [38] and Dijkstra et al. [17], continued by Ben-Ari [5], Appel et al. [1], Nettles and O'Toole [28], Doligez and Leroy [19], Doligez and Gonthier [18], Blelloch and Cheng [9], Lim et al. [25], Domani et al. [20], Hudson and Moss [22].

In the Doligez-Leroy-Gonthier (DLG) GC, tracing is concurrent with mutation using handshakes instead of safepoints. However, since handshakes block GC, the term "quasi real-time" was used to describe its guarantees.

On-the-fly GCs often use derivatives of the Yuasa barrier [40] which is a SATB technique shading the old referent black [17] and counting all newly allocated memory as live. Mutator threads stop one by one at a handshake to sample their own roots, then the SATB barrier helps maintaining the transitive closure of the stack snapshot. Eventually a checkpoint between all mutators is reached, and then tracing starts.

To address the issue of real-time root sampling, stacklets were introduced [12], [11]. The idea is to split the stack into smaller fragments and scan them incrementally as the program continues executing. Ben-Ari [4] added support for moving GCs. Recent work [24] presents claimed "lock-free" root scanning. A concurrent GC thread can help the mutator scan its roots. However, the GC still waits for a handshake from mutators to scan their stacks, making it not lock-free.

In summary, on-the-fly GC can significantly reduce the response times of GC and offload most of the GC work to possibly parallel GC threads running concurrent with mutator threads. However, memory allocation is not lock-free because of the inherent dependency on handshakes for GC progress, which is not lock-free for the GC.

Fragmentation: To address fragmentation issues while retaining low latency, another level of complexity is added.

Three lock-free compaction algorithms, Clover, Stopless and Chicken, were compared in [31].

The Clover algorithm is only conditionally lock-free; it may fail being lock-free if an unlucky value is written to the heap. Clover also has a write barrier requiring expensive atomic instructions, making it slow and impractical due to challenges giving JNI safe access to raw memory.

Stopless [30] uses double-wide CAS to copy values first to an intermediate location and then to their destination. Stopless also requires atomic instructions in the write barriers.

Chicken is wait-free for mutators and fast, but has the worst copy progress guarantees. In theory, it is possible that not a single object gets relocated, i.e. no memory bound.

What these three solutions have in common is that they need a handshake to initiate copying, making them not lock-free for GC and provide no guarantees that copying finishes. They do however provide excellent latency.

The C4 [39] algorithm is a generational variant of the Pauseless GC [13]. They both rely on protecting from-space from mutations using page protection and are therefore not lock-free. However, they provide good latency in practice.

The Collie [23] requires special hardware (Hardware Transactional Memory) to move cells. It does not have any copy progress guarantees. Cells seen from roots or with too many inbound references cannot be moved. It relies on handshakes to start copying making it not lock-free for the GC thread. It also tracks a remembered set for each object, making it impractical in terms of memory overheads.

The Metronome GC [2] copies cells in incremental safepoints. Mutator progress is guaranteed in terms of minimum mutator utilisation (MMU), but is not lock-free as mutators cannot preempt compaction by GC. The same goes for the G1 GC which is slightly faster [15]. It is, however, less predictable: it needs to finish remapping before letting the mutators run, which depends on the size of remembered sets.

Replicating GC [27], [29] copies cells from from-space to to-space and maintains a mutation log produced by mutators and consumed by GC to copy cells that have been changed once more. Sapphire [22] uses a similar approach, but improves on being more incremental and has more scalable synchronization. Ritson et al [33] add transactional memory to the Sapphire approach. These solutions doe not provide any copy progress guarantees and are not lock-free.

All previously mentioned studies need checkpoints, i.e., safepoints or handshakes, for their compaction algorithms.

The Eventron [37] is a Java based real-time programming construct that can coexist with GC. They do not allow a real-time task to change references nor allocate memory. Our GC allows this without special programming constructs.

The realtime GC [36] in the JamaicaVM uses memory indirections for normal memory accesses, sometimes $O(log(n))$ indirections for objects of size n.

The Schism GC [32] limits fragmentation issues by splitting cells into fragments of constant size. An indirection index of variable size contains information where cell fragments can be found. These index cells are immutable and can therefore easily be replicated.

Approaches handling fragmentation with non-standard memory layouts bound memory consumption, but at high performance and memory overheads. Moreover, they make raw memory accesses for JNI impractical.

8. Conclusion and Future Work

A lock-free copying algorithm was described and implemented in G1 of OpenJDK to reduce latencies of compaction. Mutators pin the address of fields before accessing them. The algorithm has copy progress guarantees, i.e. the number of objects not being copied is bounded. It does not impede the progress guarantees of neither GC or mutator threads. The algorithm was designed to be practically feasible and runs on commodity hardware without any special OS support. Its performance overhead compared to G1 is 15% on average, the fast-path of the pinning barrier is only 4 instructions and contains no fencing instructions. For this performance cost, the latency of our compaction is considerably lower than the latency of the original G1 in our experiments.

Acknowledgments

We would like to thank Jesper Wilhelmsson for having Erik over at Oracle during an internship to implement this. Many thanks also go to Bjarne Steensgaard, Jeremy Singer, Christian Wimmer and the anonymous reviewers for excellent feedback that helped improving the paper. And of course thanks to the Swedish Research council for support under grant 2011-6185.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Notices*, volume 23, pages 11–20. ACM, 1988.

[2] D. F. Bacon, P. Cheng, and V. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In *On the Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 466–478. Springer, 2003.

[3] H. G. Baker Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] M. Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In *Automata, Languages and Programming*, pages 14–22. Springer, 1982.

[5] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(3):333–344, 1984.

[6] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):13, 2012.

[7] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, volume 43, pages 22–32. ACM, 2008.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[9] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. *ACM SIGPLAN Notices*, 34(5):104–117, 1999.

[10] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262. ACM, 1984.

[11] P. Cheng and G. E. Blelloch. *A parallel, real-time garbage collector*, volume 36. ACM, 2001.

[12] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. *ACM SIGPLAN Notices*, 33 (5):162–173, 1998.

[13] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56. ACM, 2005.

[14] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–269. ACM, 1989.

[15] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[16] D. Dice, H. Huang, and M. Yang. Asymmetric dekker synchronization. Technical report, Technical report, Sun Microsystems, 2001. http://home. comcast. net/pjbishop/Dave, 2001.

[17] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[18] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–83. ACM, 1994.

[19] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123. ACM, 1993.

[20] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *ACM SIGPLAN Notices*, volume 35, pages 274–284. ACM, 2000.

[21] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[22] R. L. Hudson and J. E. B. Moss. Sapphire: Copying GC without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57. ACM, 2001.

[23] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The collie: a wait-free compacting collector. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[24] G. Kliot, E. Petrank, and B. Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20. ACM, 2009.

[25] T. F. Lim, P. Pardyak, and B. N. Bershad. A memory-efficient real-time non-copying garbage collector. *ACM SIGPLAN Notices*, 34(3):118–129, 1999.

[26] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[27] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *ACM SIGPLAN Notices*, volume 28, pages 217–226. ACM, 1993.

[28] S. Nettles and J. O'Toole. Replication-based real-time garbage collection. In *Conference on Programming Language Design and Implementation. Association for Computing Machinery*, 1993.

[29] J. O'Toole and S. Nettles. Concurrent replicating garbage collection. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 34–42. ACM, 1994.

[30] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pages 159–172. ACM, 2007.

[31] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6): 33–44, 2008.

[32] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *ACM Sigplan Notices*, volume 45, pages 146–159. ACM, 2010.

[33] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with haswell hardware transactional memory. In *Proceedings of the 2014 international symposium on Memory management*, pages 105–115. ACM, 2014.

[34] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM (JACM)*, 18 (3):416–423, 1971.

[35] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM (JACM)*, 21(3):491–499, 1974.

[36] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103. ACM, 2007.

[37] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: A Safe Programming Construct for High-frequency Hard Real-time Applications. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 283–294, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

[38] G. L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, 1975.

[39] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46 (11):79–88, 2011.

[40] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.