Block-Free Concurrent GC: Stack Scanning and Copying

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

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

Abstract

On-the-fly Garbage Collectors (GCs) are the state-of-the-art concurrent GC algorithms today. Everything is done concurrently, but phases are separated by blocking handshakes. Hence, progress relies on the scheduler to let application threads (mutators) run into GC checkpoints to reply to the handshakes. For a non-blocking GC, these blocking handshakes need to be addressed.

Therefore, we propose a new non-blocking handshake to replace previous blocking handshakes. It guarantees scheduling-independent operation level progress without blocking. It is scheduling independent but requires some other OS support. It allows bounded waiting for threads that are currently running on a processor, regardless of threads that are not running on a processor.

We discuss this non-blocking handshake in two GC algorithms for stack scanning and copying objects. They pave way for a future completely non-blocking GC by solving hard open theory problems when OS support is permitted.

The GC algorithms were integrated to the G1 GC of OpenJDK for Java. GC pause times were reduced to 12.5% compared to the original G1 on average in DaCapo. For a memory intense benchmark, latencies were reduced from 174 ms to 0.67 ms for the 99.99% percentile. The improved latency comes at a cost of 15% lower throughput.

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

Keywords non-blocking, block-free, compaction, stack scanning, 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 modify live objects and allocate new ones. Deallocating the garbage for later reuse leads to memory fragmentation, i.e., memory blocks consist of live objects and free gaps that are often too small for new objects. We distinguish logical objects from their physical memory locations referred to as cells. A fragmented heap memory region can be a from-space, part of the condemned set of potential objects for relocation. Its live objects have their from-space cells forwarded to cells in an unfragmented heap memory region called to-space.[1]

While mutators run some client application, moving garbage collector (GC) threads avoid fragmentation. GC finds live objects in the condemned set by scanning stacks and globals for references to root objects, and then tracing other live objects by computing the transitive closure of these roots through their references. GC compacts memory by relocating the 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 present paper focuses on two GC tasks, copying and stack scanning, that run concurrently to mutation. They are embedded in a host garbage collector responsible for tracing, condemned set computation, heap remapping, and memory reclamation. GC threads may run concurrently with the mutator thread(s). They need to synchronize copying of cells to prevent, e.g., that a mutator modifies a from-space cell while a GC thread has already copied it to to-space. Straight-forward synchronization approaches block access to objects being copied, or even stop the world. However, since the introduction of 64-bit processors, memory has quickly expanded and the latency of blocking GC is an issue for an ever increasing number of application contexts. Therefore, a concurrent non-blocking[2] solution is preferred.

The current state of the art, on-the-fly GCs, reduce blocking so that much can be done concurrently. However, they still fundamentally suffer from blocking because they rely on handshakes in order to, e.g., start a GC cycle and compute root objects. GC algorithms that are called "lock-free for mutators" still allow handshakes that can block the GC if mutators do not reply. This implies that memory reclamation and thus allocation requests of mutators can still get blocked. This is not good enough; for instance, most non-blocking data structures need memory allocations.

Our algorithms for copying and stack scanning are non-blocking, i.e., progress is guaranteed independently of the scheduler. We allow waiting for replies from *active* threads that are currently running on another processor if they are guaranteed to reply in a finite number of steps (CPU cycles). Conversely, we avoid waiting for replies from preempted or blocked threads that may not reply in a finite number of steps; they are handled separately. Details will be discussed in Section 2 where we introduce the notion of *block-freedom*.

ISMM'16, June 14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4317-6/16/06...$15.00
http://dx.doi.org/10.1145/2926697.2926701

[1] We inherited the terminology from Baker's algorithm (Baker 1978) 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] We use the same interpretation of non-blocking as Herlihy et al. (2003): Non-blocking synchronization is a family of progress conditions requiring that the failure or indefinite delay of a thread cannot prevent other threads from making progress.

For achieving concurrent, non-blocking copying (with memory bounds) and stack scanning, we use a special *block-free handshake*. Details will be discussed in Section 3.

The Field Pinning Protocol (FPP) (Österlund and Löwe 2015) was the first concurrent copying algorithm to be lock-free both for mutators and GC. This paper improves the solution with Deferred Field Pinning (DFP) using block-free handshakes, reducing fast-path barriers to only a conditional branch while improving progress.[3] Details will be discussed in Section 4.1.

Concurrent non-blocking stack scanning (for both mutator and GC) does not exist to date. Previous techniques could not retrieve references to objects from the stacks of other threads without waiting for them to reply. This was a fundamental progress bottleneck of GC solutions so far. We present a non-blocking stack scanning technique that solves this problem. The idea behind the solution is to handle preempted and active threads differently. A block-free handshake is performed that waits for threads that are actually running on the processor for a bounded number of steps. Preempted threads are taken care of separately by inspecting their preempted execution state from the OS. The top stack frame and registers are handled conservatively, the rest of the stack accurately. The conservatively handled top stack frame simply adds to a constant per thread memory overhead. Details will be discussed in Section 4.2.

Our copying and stack scanning are not intrusive for the GC: they do not require special memory layouts, run on stock hardware, and allow a native interface, which is crucial for real VMs. The algorithms do require new OS support that is not available today. However, we merely exposed information about threads that every OS kernel already has.

We demonstrate the practical feasibility of the approach by integrating the algorithms into the Garbage-First (G1) GC of OpenJDK (Detlefs et al. 2004). Latencies were reduced from 174 ms to 0.67 ms for the 99.99% percentile in a memory heavy DaCapo benchmark, on average down to 12.5%, at a cost of 15% throughput overhead on average for all benchmarks. Block-free handshakes, and other optimizations, are responsible for this low latency and high performance. They help avoid memory fences in the fast paths of memory access barriers, i.e., mutator loads and stores. Implementation and experiments are detailed in Sections 5 and 6, resp.

In short, there are three major contributions of the present paper. (1) It introduces block-free handshakes that exploit OS support to improve the progress guarantees of blocking handshakes. (2) It exploits block-free handshakes in concurrent, non-blocking copying of live objects, which considerably optimizes this GC algorithm while improving its progress guarantees. (3) It presents the first concurrent, non-blocking stack scanning approach enabled yet again by block-free handshakes. (2) and (3) were implemented in the G1 GC of OpenJDK where block-free handshake allowed other optimizations providing better overall performance. Measurements using the DaCapo benchmark suite confirm the practical feasibility of our approach.

Non-blocking copying and stack scanning are not enough for making a completely non-blocking GC. However, they were hard remaining fundamental problems that had not been solved before. The contributions of the paper get us close to a complete concurrent and non-blocking GC.

2. Block-Freedom

To avoid confusing the different overloaded definitions of lock- and wait-freedom, we define the notion of *block-freedom*. It is a non-blocking progress property similar to previous definitions of *lock-freedom*, e.g., the ones by Herlihy and Shavit (2011). It is within the bounds of the definition by Fraser (2004) but outside the one by Michael and Scott (1998).

[3] Conditional branch instructions have been aggressively optimized since the introduction of deep pipelines in CPUs.

Table 1: Comparison of possible synchronization issues.

Sync Property	**Blocking**	**Block-free**	**Lock-free**
Deadlocks	yes	no	no
Priority inversion	yes	no	no
Convoying	yes	no	no
Starvation	yes	**no**	**yes**
Kill issues	yes	no	no
Preemption issues	yes	no	no
Async signal issues	yes	no	no

Table 2: Comparison of synchronization prerequisites.

Help allowed by	**Blocking**	**Block-free**	**Lock-free**
Hardware	yes	yes	yes
Scheduler	yes	no	no
OS	yes	**yes**	**no**

The idea behind block-freedom is conceptually simple: a block-free algorithm guarantees progress of operations after finite number of *system-wide* steps, allowing OS dependency as long as the scheduler is not constrained. Its goal is also simple: remove problems with blocking like deadlocks, priority inversion, convoying etc., but without being unnecessarily restrictive. Instead of explicitly disallowing, it welcomes support from both OS and hardware to aid the progress of operations.

A *static operation* is an algorithm. A *dynamic operation instance* is an execution of a static operation. It takes *steps* that change the operation instance's execution state. Operation instances may be active or inactive. An *active operation instance* is currently executed by an active processor; an inactive operation instance is not. An *active processor* is a processor that is currently performing steps. The active processors take steps in parallel.

DEFINITION 1. *An* operation *is* block-free *iff any of its* active operation instances *will progress if it remains active for a finite number of system-wide steps taken by the active processors, independently of inactive operation instances. This holds even if any active operation instance can become inactive at any point.*

The statement that any active operation instance may become inactive at any point implies that preemption of operation instances must always be allowed. This gives the scheduler and operations the freedom to be mutually independent of each other. Hence, the property is scheduling independent as required by any non-blocking progress property. Because of this requirement, mechanisms such as `mprotect` are unfortunately not allowed for block-free operations, because it requires a non-preemptive lock in the kernel. This non-preemptive locking constrains the scheduler, which is not allowed for block-freedom. Any other hardware or OS feature necessary to guarantee the progress is welcome.

This definition makes it explicit that the progress of active operation instances is independent of inactive operation instances. However, unlike most definitions of lock-freedom, *bounded* waiting on other *active* operation instances is explicitly allowed. Block-freedom permits bounded waiting for an active thread to either reply or become inactive and then be handled separately.

Active operation instances concurrently executing on different processing units are *actively concurrent*; operation instances that are active and inactive due to preemption are *inactively concurrent*. It is useful for synchronization algorithms to know if a conflict being dealt with is due to active or inactive concurrency to open up opportunities for resolving these conflicts in different ways. For example, active concurrency has to deal with the state of conflicting operation instances being uncertain (invisible and loaded into remote processor) and continuously changing, with potentially delayed memory accesses requiring appropriate fencing for consistency. Inactive concurrency on the other hand deals with a potentially stale state of conflicting operation instances, which comes with a different set of problems, but does not have to handle, for example latent stores. Therefore, OS support to separate active and inactive concurrency ought to be exploited and is encouraged for block-free algorithms.

Finally, it is useful to remember the properties that motivated different progress guarantees and their prerequisites. A comparison of the progress guarantees of blocking, block-free, and lock-free (according to the definition in (Herlihy and Shavit 2011)) can be seen in Table 1. Note that any lock-free algorithm can be transformed into a wait-free algorithm which solves the starvation problem (Kogan and Petrank 2012). A comparison of the prerequisites is given in Table 2. Together the tables show that block-freedom has fewer requirements and stronger progress properties than lock-freedom.

3. Block-Free Handshakes

Remember that the block-free progress property allows bounded waiting for either a reply from another active thread or for that thread to become inactive. To aid such algorithms we created a general-purpose synchronization mechanism: the block-free handshake. It is an abstraction with similar properties as handshakes, but with non-blocking progress guarantees. Since many existing GC algorithms such as on-the-fly GCs rely on handshakes, much of their code can be adopted to use block-free handshakes instead, giving better progress guarantees.

A block-free handshake is performed by 1) requesting a handshake to responder threads, 2) getting their execution states, i.e., their register contents and references to the top stack frames, and 3) replying using these execution states. This execution state can be provided by an active responder in a GC checkpoint as for normal handshakes. However, in order to be block-free, also handshakes to inactive or uncooperative responders are finished by inspecting snapshots of their execution state. Such inspection is always paired with a barrier that, upon waking up, forces the responder to help finishing the handshake before continuing.

3.1 Requesting Block-Free Handshakes

The block-free handshake can be requested with one or more responder threads. To request a handshake with a snapshot of all mutator threads, the handshaking operation starts with iterating through a snapshot of all such threads from the VM thread list. For each thread, a handshake operation is added to a block-free operation queue.[4] The thread list is a block-free list that allows reading a snapshot of all threads.[5]

3.2 Getting Execution State

In order to reply to a handshake request, an execution state is needed. In a normal handshake, the requesting thread would wait until all threads have replied in checkpoints. Block-free handshakes require that the execution state of uncooperative responders can be inspected by other threads.

[4] We use a wait-free linked list managed with Safe Memory Reclamation (SMR) as described by Kogan and Petrank (2011), which is also block-free.

[5] We use a lock-free copy-on-write list managed with SMR accompanied with a helper as described in Kogan and Petrank (2012) to handle starvation, and hence achieve block-freedom.

Checkpoints are emitted by the compiler, e.g., in basic block back edges and returns, so that any location in managed code will have a finite number of steps to the next checkpoint. These checkpoints comprise thread-local yieldpoints using a conditional branch to check if a special action is required, such as handshaking. If handshaking is required, the mutator processes a snapshot of the list of handshake operations and clears the list before continuing. That way, a mutator can not be starved from running, even if the requesting threads keep on sending handshake requests while handshakes are being processed by the mutator.

The requesting thread either 1) waits a finite number of system-wide steps for a concurrently executing active mutator to get the execution state in a GC-checkpoint and finish the handshake operation, or 2) inspects the execution state and finishes the handshake operation itself. Either way, the operation is block-free.

3.2.1 Execution State Inspection

The thread requesting a block-free handshake can inspect a responder mutator thread to find out if it is both active (OS state) and managed (VM state).[6] We call this combination of VM state and OS state *managed-active mode*.

When trying to inspect the execution state of a responder mutator thread, the thread requesting a block-free handshake first checks for threads that are not in managed state using the VM state information. VM state transitions use fences, which is rather inexpensive today. Therefore, potentially blocking optimizations that require Asymmetric Dekker Synchronization (ADS) (Dice et al. 2010) for reading the VM state are not necessary. Our algorithm remains block-free.

If the requesting thread finds an uncooperative responder in non-managed VM state, then the responder thread has already published its execution state (frame pointer of top managed stack frame) before transitioning into that non-managed state. This idea was first introduced by Kliot et al. (2009). However, they had no way of dealing with threads that are inactive in managed VM state other than blockingly waiting for their reply, hoping for these uncooperative threads to be scheduled to run eventually. This blocking operation spoiled non-blocking guarantees of their solution. An important contribution of the present paper is a way to handle this situation in a block-free way. When an inactive responder thread is found in managed VM state, a new OS call is used to inspect the execution state.

3.2.2 OS Call to Inspect Execution State

To help algorithms handle active and inactive concurrency differently, we added a new general purpose OS call. In a block-free and non-intrusive manner, the call (1) finds out if a thread is active, (2) tries to inspect the thread state of inactive threads and (3) installs a barrier to be run should the thread become active again, provided the state was successfully inspected in (2). Algorithm 1 gives a possible implementation of this system call that was added to the XNU kernel.[7] The implementation does not perform any blocking nor does it use locks. Naturally, the implementation of the inspection system call would look different on a different OS with a different scheduler.

The call starts by sampling the running state of the thread and the number of context switches it has made. Then if it is assigned to a processor and is not in a blocked state, it goes on to check if it is the actively running thread of that processor. If so it fails. Otherwise it samples the registers from the Process Control Block (PCB) of

[6] This VM state describes whether the thread executes Java code (managed), native or VM runtime code, or if it is not running at all (blocked).

[7] The algorithms presented should be read as using volatile (compiler) but acquire/release (hardware) memory ordering semantics

that thread. After the registers have been sampled, the running state and number of context switches are sampled again and if anything changed, failure is returned as a stable snapshot could not be inspected. Otherwise, the stable execution state snapshot is returned successfully.

```
def thread_try_get_state(thread)
  state = thread.running_state
  context_switches = thread.context_switches
  processor = thread.last_processor
  if state != RUNNING || processor == null
    return false # not in managed state
  elsif processor.active_thread == thread
    return false # active; no stable state
  end
  install_signal(thread, SIGPREEMPT)
  register_sample =
    sample_registers(thread.user_state)
  if thread.last_processor != processor ||
    processor.active_thread == thread ||
    thread.context_switches != context_switches
    return false # unstable snapshot
  else
    return register_sample # stable state
  end
end
```

Algorithm 1: New thread state inspection system call

Invalid accesses to PCB are prevented by handling threads with hazard pointers and Safe Memory Reclamation (SMR) (Michael 2004) in the JVM so that threads that are being accessed are not freed and have a valid PCB in the kernel.

Before the mutator thread state is inspected, a special signal, SIGPREEMPT, is set on that thread. Hence, in case the thread wakes up right after the thread state inspection returns from the kernel, the signal handler in the VM will be a barrier called when the application gets active and before it starts running again.

The corresponding pseudo code for scheduling a thread in XNU can be seen in Listing 2. Line 4 changes the OS state to active and line 5 invalidates the staleness of the execution state.

```
 1 def context_switch(processor, source_thread,
       target_thread)
 2   store_registers(source_thread->user_state)
 3   target_thread->last_processor = processor
 4   processor->active_thread = target_thread
 5   target_thread->context_switches++
 6   fence() # StoreLoad membar
 7   if target_thread->has_signal()
 8     run_signal_handlers(target_thread)
 9   end
10   load_registers_and_invoke(target_thread)
11 end
```

Algorithm 2: Context switch into thread

Threads being preempted unload their registers in PCB before changing any thread state. We make sure such thread state transitions are made with a releasing store to guarantee that once the state change is observable to other threads, the stores unloading registers in PCB are observable as well.

The membar on line 6 makes sure that preceding stores that change the OS thread state, serialize before checking for signals. If omitted, data races can occur due to concurrent inspecting threads still perceiving the target thread as inactive, causing the special PREEMPT signal to be skipped.

The system call was implemented for the XNU kernel but could be added to any OS kernel. The OS support required already partially exists. For example, checking the activeness of a thread is done by schedctl on Solaris and thread_get_info on Mach. Retrieving the execution state of a thread is similar to thread_get_state on Mach, and the barrier is like a normal signal or dtrace probe. However, we require these primitives to be block-free, and therefore made a custom system call instead.

The current XNU implementation relies on threads that get scheduled to run also becoming active, i.e., at least the first instruction of the user space execution state of the thread starts executing on an active processor. Therefore, even if mutators get scheduled to run but execute only a single instruction per scheduling quantum, it is enough to guarantee (slow) system-wide throughput; the handshake is finished by a requesting thread in a loop that either inspects the execution state successfully (inactive thread) or that thread is active and takes at least one step towards the next checkpoint, which will actively reply after a finite number of steps.

A hypothetical kernel might allow an arguably broken scheduler that can repeatedly schedule a thread to run but subsequently interrupt it before becoming active (i.e., before the first user-space instruction starts executing). Note however that even such a hypothetical kernel could still retrieve the stale execution state of such inactive threads from PCB in the kernel. It merely has to be less conservative in verifying the staleness of the execution state after loading it, compared to the current implementation that always invalidates staleness of the state upon scheduling, assuming that the thread will become active.

3.2.3 Managed-Active Barriers

Block-free handshakes come with the ability to install a barrier before a mutator thread becomes active (OS state) and managed (VM state). In the context of GC algorithms, this comes in handy because conflicts between a GC thread and (awaking) mutator threads can be handled. Especially, if a thread T_e other than the responder thread T_r, competes for finishing the handshake operation of T_r using an execution state inspected either from the VM or the OS, then T_r is guaranteed to run a managed-active barrier that competes for finishing the handshaking operation using the same execution state as sampled by T_e before continuing.

To make sure the managed-active barrier is invoked properly, code for transitioning into managed VM state also checks for pending handshake requests and runs any potential managed-active barriers required by them.

Similarly, threads that were preempted in managed VM state, have a special signal, SIGPREEMPT, sent to them when their execution state gets inspected using the thread_try_get_state() system call. It makes the thread run a non-reentrant signal handler before it becomes active. This signal handler publishes the execution state of the thread and changes the program counter to a handler calling the managed-active barrier. Subsequent state inspections may use the published state to finish the handshake.

We are careful with sending signals installing managed-active barriers to threads that are running interruptible system calls because there is signal-unsafe native code in library methods that do not check the status of system calls that were interrupted. We avoid this by making sure that the SIGPREEMPT signal is only checked, and hence its handler only run, when scheduling threads to run that are already in OS state *running*, i.e., not in a blocking system call. Threads that ran such system calls would not be in managed state.

3.3 Replying to Handshakes

A mutator thread can reply to a handshake directly from 1) a thread-local yieldpoint, or 2) from a managed-active barrier requested either by the OS or a VM state transition. Such replies are called *active*. The block-free handshaking operation remains in the handshake operation list of the replying thread until it is actively replied

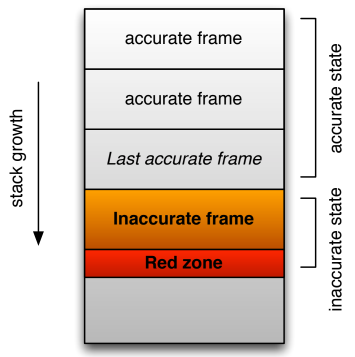

Figure 1: Stack of preempted thread being scanned.

to by the responder. It could involve some extra work, e.g., committing changes made by other threads.

Conversely, if another thread replies to the handshake instead of the intended responder thread, the reply is called *inactive*. This could happen if either the responder mutator thread is not in managed VM state or has been suspended by the OS because of, e.g., preemption. Then it is allowed for another thread, typically the requesting thread, to try finishing the operation.

3.3.1 Handling Inaccurate Replies in GC

A reply, active or inactive, will have an accompanying execution state of the responder thread. From this execution state, it is determined if the reply is *accurate* or *inaccurate* by looking up the managed code blob of the first managed stack frame (found from the execution state) and whether it was at an *accurate point*, i.e., had its program counter in a GC-checkpoint where reference maps are available, or not. If the execution state was in an accurate point, the reply is accurate, otherwise inaccurate.

In the case of an accurate reply (either from the requesting thread or the responder thread) there is only accurate state information, comprising all registers and stack frames. Even for inaccurate replies, the stack frames are mostly accurate and only some have inaccurate parts as shown in Figure 1.

Requesting threads may never receive an accurate reply from a responder that is, e.g., preempted indefinitely by the OS. This can be handled by the GC in two different ways: 1) dealing with inaccurate replies, or 2) making the reply accurate. We chose to handle inaccurate replies using a conservative GC approach.

Fortunately, the inaccurate portion of the state is bounded to the size of the top stack frame (which can be made constant) and registers. On some platforms, this also includes a “red zone” from the top stack frame that might have been used as a scratch area for leaf calls, allowed by the ABI. On AMD64 used in the implementation, this is 128 bytes.

The other stack frames are mostly accurate apart from their callee-saved registers that could still be residing in the conservative portion of the snapshot.

Handling conservative references is nothing new. It has previously been done, e.g., by Boehm (1993) and Demers et al. (1990). We can still guarantee GC-completeness but must handle that an additional constantly bounded amount of objects can not be reclaimed. This results in bounded temporary garbage, rather than progress impairment. Handling bounded number of objects that cannot be reclaimed is already handled by our GC. Boehm (2002) also noted that handling the top frame conservatively can, under extremely unlikely conditions, result in an unbounded amount of unrecognized garbage. This can be handled by repairing the program counter (PC) in managed code on-the-fly to get an accurate reply. This theoretical issue was left to future work.

4. Garbage Collection Algorithms

The previous state of the art of non-blocking GC is limited to on-the-fly GC, accepting blocking handshakes by the GC. With the block-free handshake defined, these normal handshakes can now be replaced with block-free handshakes. In this paper we focus on two GC algorithms: block-free copying of objects and stack scanning.

4.1 Block-Free Copying Using Field Pinning

One of the difficulties with non-blocking copying is to maintain consistency when multiple mutator threads are concurrently accessing a field that GC is trying to copy while limiting the number of copy-attempts that can fail, without impacting performance or memory overheads.

Our solution is based on the previous work of Österlund and Löwe (2015) that solves these problems. The algorithm, referred to as the *Field Pinning Protocol* (FPP) builds on the idea of pinning the address of field accesses (writing a hazard pointer pointing at the field being accessed) without requiring the mutator nor GC to perform a blocking handshake while allowing a bounded number of copy failures ($\epsilon \in O(t^2)$ where t is the number of threads).

The FPP algorithm lets objects be in one of three colors: copy-white, copy-gray, and copy-black. Objects start in copy-white. When copying starts, objects get shaded copy-gray; the from-space cell is forwarded to a to-space cell that has an encoded pointer in its cell header, pointing to a special status cell containing copying metadata for the copying protocol. Eventually, GC shades the object copy-black after copying the object payload from from-space to to-space. The object is copy-black after removing the encoded pointer from the to-space cell to the status cell. Now copying is finished and the GC can perform remapping. Once all incoming references to the from-space cell have been remapped to to-space, the object is copy-white and may be relocated once again.

The copying itself uses hazard pointers to pin fields for consistent mutator memory accesses. These hazard pointers are scanned by copying threads that avoid copying fields that could be concurrently accessed by a mutator. Asynchronous copying with the help of the mutator threads themselves completes such impeded copy attempts. Note that FPP was already a non-blocking compaction algorithm requiring no handshakes. Actually it was the first one, and the first to guarantee a bound on copy failures.

4.1.1 Deferred Pinning Using Block-Free Handshakes

Unfortunately, writing hazard pointers usually comes at the cost of memory fences, which is a high price for block-freedom. Österlund and Löwe (2015) propose Asymmetric Dekker Synchronization (ADS) to elide the memory fence from the fast-path of the field pinning barrier using `mprotect` for serializing latent hazard pointer stores on other processors. However, that optimization is not block-free because the `mprotect` call breaks the requirement that operation instances can become inactive at any point with a non-preemptive kernel lock.

```
;; Field pinning of $OFFSET(%r_obj_ref)
;; Test for copy-color
testb $HEADER_OFFSET(%r_obj_ref), 3
;; Not clearly copy-white objects
jp CODE_STUB
;; Actual memory access
movq %r_value, $OFFSET(%r_obj_ref)
```

Algorithm 3: AMD64 code for the fast path pin barrier using DFP

We propose deferred field pinning, a block-free optimization that elides the memory fence using block-free handshakes instead of `mprotect`. Pinning fields is moved into the slow-path triggered only when the mutator accesses copy-gray objects. The fast-path (cf. Algorithm 3) of the barrier only has a single conditional branch to a code stub triggered on objects that may not be copy-white.

Copy-white objects might spuriously branch since the GC header word is shared with monitors. In such an event, an instruction (`je`) in the code stub branches back to the fast-path. For copy-black objects `r_obj_ref` will refer to to-space after the code stub. If the object is copy-gray, a runtime call is made to the slow-path, using pinning with fences.

A block-free handshake is used to safely publish shading of objects from copy-white to copy-gray. After this handshake, a slow-path will be triggered for new accesses on the copy-gray objects, to find out if from-space or to-space should be used by the mutator.

The mutators may reply to the handshake in three different ways. 1) An active reply from a thread-local yieldpoint, implying previous memory accesses (that happened before the handshake was issued) are observable. 2) The thread is in a non-managed state, in which case any heap access uses slower FPP with a memory fence. 3) A mutator was in managed state and inactive. In this case, it could have been preempted after the conditional branch but before the memory access in Algorithm 3. This is handled by lazily computing its hazard pointer had it been written from the execution state provided by the block-free handshake. This computed hazard pointer is then installed in a separate thread field using Compare-And-Swap (CAS) and moved back to the normal hazard pointer field in the managed-active barrier.

Metadata is tracked for all field pins in managed code so that at any given PC between the logical pin and unpin, information is maintained about the base register and offset so that a hazard pointer can be lazily computed for preempted threads.

4.1.2 GC Batches and Relaxed Synchronization

The GC batches copying. It is started by shading the contained objects of a batch copy-gray with a block-free handshake and is then copied asynchronously once the handshake has reached a synchronization point. While lazily synchronizing previous copying batches, new batches are started.

This block-free handshake has three urgency levels corresponding to different efforts required to reach a synchronization point. The first level of urgency is relaxed. Mutators do not have their thread-local yieldpoints active on this low urgency level. Instead, threads reply by chance when making calls to the runtime system. Upon entry to the runtime, mutators load a current global timestamp, increased for each handshake started, and store it (with fence) to a thread-local timestamp. This allows the GC to see if a thread has replied by comparing its timestamp to this thread-local timestamp. The second level of urgency activates the thread-local yieldpoint, forcing responders to reply at the next checkpoint and then similarly update the timestamp observed. The third level of urgency makes the GC inspect the OS thread states if necessary to end the handshake even in case of thread suspension by the OS and compute deferred field pins. When there are too many batches being synchronized, the urgency level is increased.

This technique allows amortizing the synchronization costs both for GC and mutators. Mutators can with an updated timestamp reply to many global synchronization requests at the same time, i.e., they reply to all previously issued batches at the same time. Similarly the GC may amortize the synchronization cost of copying many objects with a single asynchronous handshake that can be lazily finished. Once the most recent batch has synchronized, all batches before share the same synchronization point.

Remark: We used the same solution to elide the memory fences required by G1 to conditionally skip dirty cards. The concurrent refinement is batched, sorted, prefetched and lazily synchronized using block-free handshakes.

4.1.3 Progress Guarantees

The original FPP algorithm was already non-blocking (lock-free), but the reworked algorithm is even block-free. The synchronization mechanism to publish copy-gray objects using a block-free handshake is trivially block-free. The word level copying protocol is also block-free because it finishes in a finite number of steps. The lock-free copy-on-write sets used for asynchronous copying have been fortified with helper methods to avoid starvation. Therefore, the whole copying algorithm is block-free for both mutator and GC. To the best of our knowledge, this is the strongest progress guarantee of any concurrent copying algorithm to date.

4.2 Block-Free Stack Scanning

A thread holds private state in, e.g., registers and the execution stack, that needs to be accessed by GC to sample and remap roots of that thread. For a non-blocking stack scanning algorithm this must work even if the thread is inactive and never scheduled to run again.

The previous state of the art, on-the-fly GCs, scanned stacks in a round-robin fashion, stopping threads one-by-one in a handshake. Naturally, mutator threads not scheduled by the OS would stop GC from progressing and break non-blocking guarantees of the GC. Our approach handles this by using a block-free handshake instead.

4.2.1 Accessing Internal Thread State

Any thread state access is performed by requesting a block-free handshake operation to the responder thread. The operation describes the transactional operation and can be run by either thread. The requesting thread or the responder thread can compete for finishing the operation. As with any block-free handshake, the requesting thread does not need to compete if the responder thread is in managed-active mode because the responder thread will respond in a checkpoint after a finite number of steps. If it is not in managed-active mode, the requesting thread competes with the managed-active barrier of the responder to finish the operation.

Reads and writes of the internal thread state are handled via block-free read- and write-buffers. Assume a mutator and GC thread are competing for finishing a handshake operation. The mutator could finish the operation first and then continue running code that invalidates the reads of a concurrently executing GC thread. In this case, the GC thread will read values that should not be added to the block-free read-buffer. This is not a problem because the read- and write-buffers are accessed with CAS. Installing values based on invalidated reads will fail and the GC will find that the operation already finished by another thread. Reading potentially invalid states requires occasionally validating the transaction (by checking its status) to prevent, e.g., infinite loops based on inconsistent reads. This handling of inconsistent reads is analogous to the approach of STM systems, e.g., the ones of Harris and Fraser (2003) and Saha et al. (2006).

Write buffers are only physically committed by the thread the data belongs to and only in an active reply. Note that inaccurate and accurate state is read, but only accurate state can be written in transactions. After writes have been logically committed to a thread, the next transaction will set read-buffers with the values logically committed from the write-buffer rather than the physical values; even if a thread never actively replies, the values can be logically changed.

Using this approach, transactions on internal thread state of mutators can be performed in a block-free way.

4.2.2 Stack Sampling and Remapping

A stack scanning operation is transactionally performed using a block-free handshake. Both mutator and GC threads compete to finish it. Any potential object referenced by inaccurate roots is added to a *conservative root set* of a thread.

For a moving GC, an initiating stack scan is performed to sample roots to the heap. The block-free scanning operation marks all roots in the conservative root sets as live (even if they are in fact not). That way, accurate roots and conservative roots are treated the same during sampling.

In a second remap phase, accurate copy-black roots are shaded copy-white (transactionally using a block-free handshake) by replacing their from-space references with corresponding to-space references. Conservative roots (pointing to objects in the conservative root set) are not remapped until the mutator responds accurately. Similarly, roots pointing to copy-gray objects are not remapped. Consequently, some bounded amounts of objects pointed at from the top stack frame of a mutator could become temporary garbage that can not be reclaimed until an accurate reply is received. The from-space cells of such conservative roots can not be freed even though their payload might have been successfully copied to to-space, and the to-space cells can not be moved again until remapping has finished.

If a mutator is inactive for many GC cycles, objects referenced from the conservative root set are guaranteed to not be freed and any stale memory access will have lazily computed hazard pointers protecting it from inconsistencies.

Garbage that cannot be freed due to conservative handling of the top stack frame is bound to the number of objects that can be referenced from it and registers. Therefore, the total amount of floating garbage due to conservativeness is $O(t)$, where t is the number of threads. The total amount of garbage due to copy impediments is $O(t^2)$ (Österlund and Löwe 2015), and therefore the total bound of garbage of our solution due to conservative roots and copy impediments is $O(t^2)$. However, even one conservative reference can in theory lead to an unbounded number of objects being conservatively treated as live. A JIT compiler generating reference maps on the fly for the inspected execution states could solve this problem eventually.

This stack scanning algorithm is block-free both for mutator and GC making it the first completely block-free (and non-blocking) stack scanning algorithm to date.

5. Implementation and Optimizations

To demonstrate the practical feasibility, our algorithms have been implemented for Java in the HotSpot JVM of OpenJDK 9 for the AMD64 architecture and our own custom XNU kernel. This JVM is used for evaluation (cf. Section 6). This section describes the surrounding host GC algorithm.

We integrated our algorithms with the Garbage First (G1) GC (Detlefs et al. 2004). G1 splits the heap into many regions. Each region has a remembered set consisting of all incoming references from other regions. 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 thread roots including stacks, various globals including code roots and remembered sets, then relocating its transitive closure (in the condemned set). Regions that have been completely evacuated, i.e., contain no copy-gray objects or conservative roots, can then be reclaimed. Evacuations (including sampling condemned set roots, tracing through the condemned set and relocating its live objects) used to be done in incremental safepoints.

In our GC, evacuation is done concurrently. The different phases of the evacuation are separated by block-free handshakes. The only exception is reference processing which is still done in a safepoint. Reference processing is arguably not a mandatory GC feature. However, Ugawa et al. (2014) describe how to perform even reference processing concurrently (but not non-blocking).

Doing evacuation concurrently led to some anomalies of the original G1 algorithm, discussed in this section. For example, 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 (Yuasa 1990), already emitted for concurrent marking, is activated during evacuation of the condemned set. It makes sure all objects, that were live at the snapshot when evacuation started, get relocated. The SATB buffers are flushed in block-free handshakes that finish tracing.

5.1 Concurrent Young Cards

The original G1 algorithm uses a special young card value for all objects in the young generation. This allowed it to stop the post-write barrier early in the young generation based on the assumption that the whole young generation is evacuated in a safepoint. Our algorithm does this concurrent to the application execution. Therefore, as the condemned set is being evacuated, new references can be added to the condemned set from survivor regions and new concurrently allocated regions. To address this issue, special `conc_young_card` values are installed in concurrently allocated young regions, including new GC allocated survivor regions.

This `conc_young_card` value means that the card is in a young region concurrently allocated during evacuation, but has no references into the condemned set (being evacuated). The post-barrier is modified to dirty such cards on reference writes into the condemned set, without logging the card for concurrent refinement. All cards in the concurrently allocated young generation could have dirty cards and have their cards scanned for such references during remapping. The reasoning for doing this instead of concurrent refinement is that the scalability of enqueuing cards is not needed for the young generation, and it is cheaper to mark the cards as dirty than log and refine the cards into remembered sets.

5.2 Concurrent Remembered Set Scanning

In order to relax the remembered sets and scan them concurrently, the ability to iterate over objects in the heap (regions) must first be ensured. This is done using a block-free handshake that releases allocation buffers of mutators. Remembered sets and a snapshot of refinement buffers are then scanned for references concurrent to mutator execution.

A self-healing post write barrier was added for references: for any copy-black reference written to the heap, it is remapped in the post-write barrier. This healing could have been done in a read barrier, as well. However, since it is assumed that inter-regional reference stores are more infrequent than reference loads, and all references that need healing will be inter-regional, this is cheaper in G1 and has the same effect. Particularly, after a region has been evacuated, no more heap locations will be tainted with references into that region without being healed right after. Any new stores will therefore store references to the to-space cell.

The remapping phase ends with a special block-free handshake in which mutators compete for finishing the handshake operation of all mutators before continuing. This aligns all active mutators so that they have no pending self-healing to be done. Care must then only be taken to handle the threads not in managed-active mode correctly. They could have been preempted somewhere in the post-write barrier, awaiting self-heal. However, such stale mutators would have a (potentially deferred) hazard pointer marking which field is being accessed so it can be remapped.

This algorithm allows taking a snapshot of cards in the remembered set concurrent to mutator execution, walk through them and scan any stale from-space pointers and remap them concurrently. After this, thread roots can be flipped with a block-free handshake. The healing barrier in combination with the Yuasa-style SATB barrier allows lazy remembered set snapshot sampling and remapping.

By relaxing remembered sets, it is possible to scan snapshots of cards more lazily. At the same time, major latency bottlenecks are removed. Another feature is that stale references to to-space are not spread around in the heap, likely causing fast-path barriers to be missed. While this technique could be made completely block-free, the G1 remembered set internal data structures are not yet block-free. Therefore we make no claims of the remembered set implementation being block-free.

6. Evaluation

The DaCapo benchmark suite (Blackburn et al. 2006) is a standard benchmark for measuring performance of real-world applications. In order to evaluate our solution, we used DaCapo 9.12.[8] 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 G1 with or without our algorithms. We compare our implementation to the original G1 since it is our host GC.

The benchmarks were used for measuring performance, i.e., throughput and latency, of the JVM and its execution environment. Experiments were run on a MacBook Pro with 2.7 GHz Intel Core i7 Sandy Bridge, 4 cores, 16 GB 1600 MHz DDR3, 256 KB L2 cache and a shared 6 MB L3 cache. It runs Mac OS X 10.10.5 with a custom XNU kernel exposing the new system call and reduced scheduling quantum at 0.75 ms down from 10. This was changed because OS jitter outweighed GC latencies by an order of magnitude, making it difficult to plot the improvement. All benchmarks were run with 4 GC threads and 4 concurrent refinement threads.

6.1 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 and locking). Full support for the `-server` mode c2 compiler is a work in progress.

6.2 Latency

The latency of a JVM is the time it takes before the execution environment responds to a request. The delays caused by compaction are typically one of the primary causes of bad latency, especially, in systems with large heap sizes.

We sampled 30 GC pauses (with -XX:+DisableExplicitGC to disable `System.gc()`) excluding the first 2 on all DaCapo benchmarks running with 512 MB heaps and 256 MB young generation, except luindex that had 32 MB young generation because it uses less memory and h2 had 1 GB heap because it uses more memory. Pause times were reduced down to 12.5% of the original G1 on average using DFP. Detailed results can be seen in Figure 2. The squares represent the baseline G1 solution, and the crosses represent our improved DFP solution. Both JVMs ran with the same limitations: no fast locking nor biased locking nor inlined intrinsics for array copying and `sun.misc.Unsafe`.

We observe that with G1 only the benchmark h2 had any real latency issues in the first place. The latency issues are due to

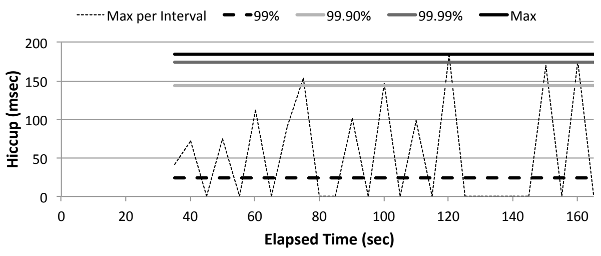

Figure 3: DaCapo h2 hiccups with normal G1

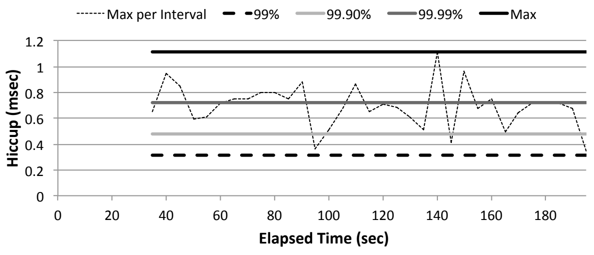

Figure 4: DaCapo h2 hiccups with our modified G1

using more memory, which fits the intended application domain. Therefore, we studied the h2 benchmark in more detail.

The latency of running h2 was recorded with jHiccup, a tool from Azul Systems measuring latency of the execution environment of an 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 execution environment in intervals of 5 seconds and 5000 samples per interval while running the application. Each interval corresponds to one dot in the line chart, representing the worst response time in that interval.

The resulting charts from running 25 iterations of the h2 DaCapo benchmark is shown in Figure 3 for normal G1 execution and Figure 4 when using our concurrent extensions instead. `System.gc` was disabled with `-XX:+DisableExplicitGC`. Both JVMs run with 8 GB heap. We ran h2 with a larger heap because it makes 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. This noise 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.

The latency has significantly improved and is around the scheduling quantum of the OS. The hiccups are no longer due to compaction but almost exclusively due to reference processing. Conclusively, by using concurrent compaction, the latency was greatly reduced for every percentile, most notably from 174 ms to 0.67 ms for 99.99%.

6.3 Throughput

Figure 5 shows the normalized running times of the benchmarks compared to the original G1 solution, after 10 warmup iterations, with 512 MB heap and the client c1 JIT compiler, matching configurations used for measuring the GC pauses. The absolute running

[8] There is a lack of benchmarks with real-world Java applications specifically targeting latency issues (Kalibera et al. 2009). Especially, the benchmark linked in that paper is disconnected. Now it points to a Japanese dating website instead.

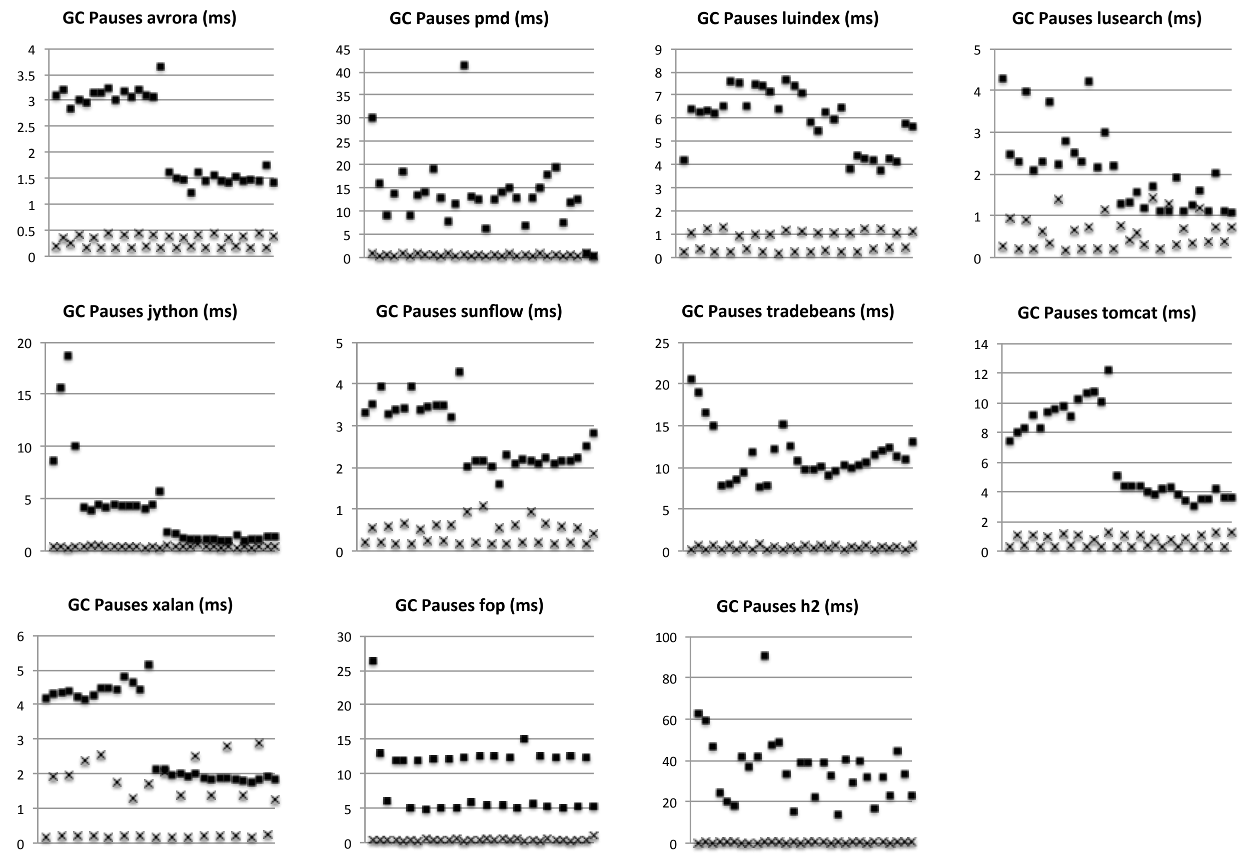

Figure 2: Comparison of GC pausetimes in DaCapo benchmarks. Squares represent G1, and crosses represent our DFP version.

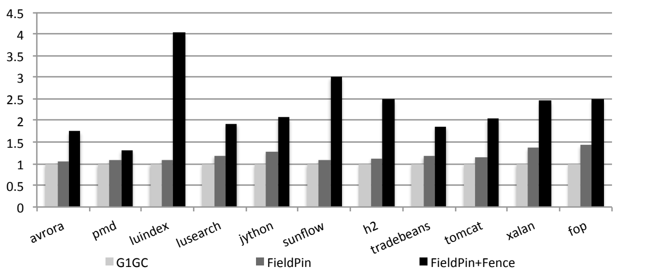

Figure 5: Normalized running times for DaCapo

Table 3: DaCapo absolute running times

Bench	G1GC	FieldPin	FieldPin+Fence
avrora	3861 ms	4087 ms	6747 ms
pmd	2632 ms	2871 ms	3479 ms
luindex	890 ms	979 ms	3594 ms
lusearch	2319 ms	2758 ms	4478 ms
jython	6822 ms	8790 ms	14099 ms
sunflow	5348 ms	5876 ms	16045 ms
h2	6374 ms	7171 ms	15838 ms
tradebeans	7787 ms	9150 ms	14531 ms
tomcat	4408 ms	5120 ms	9031 ms
xalan	1847 ms	2530 ms	4527 ms
fop	567 ms	808 ms	1414 ms

times can be seen in Table 3. Again, both JVMs run with the same limitations: no fast locking nor biased locking nor inlined intrinsics for array copying and `sun.misc.Unsafe`. The h2 benchmark was run with twice as much memory (because it used more memory) and jython ran twice as many warm up iterations (because it warmed up slowly).

The bars for **G1GC** show running times of the JVM using G1. The **FieldPin+Fence** bars show running times with the original field pinning protocol, where every field pin requires a memory fence. The average throughput loss with this technique is 53.2%. The **FieldPin** bars show the running times with DFP enabled. The average loss in throughput for all benchmarks is 15.4%. This could get worse on a faster JVM with a server JIT-compiler but, such a compiler also has more opportunities to elide barriers.

7. Related Work

On-the-fly GCs were designed to do more work concurrently for improved latency. They started with the work of Steele (1975) and Dijkstra et al. (1978), continued by Ben-Ari (1984), Appel et al. (1988), Nettles and O'Toole (1993), Doligez and Leroy (1993), Doligez and Gonthier (1994), Blelloch and Cheng (1999), Lim et al. (1998), Domani et al. (2000), Hudson and Moss (2001). This generation of GCs did not need safepoints, which was an important step in reducing GC latencies. Since, for instance, root scanning depends on all mutators eventually reaching a handshake, it could not be called non-blocking in terms of GC progress guarantees.

Stack Scanning: Stacklets were introduced to reduce latencies of stack scanning (Cheng et al. 1998), (Cheng and Blelloch 2001). The idea is to split the stack into smaller constant sized fragments and scan them incrementally as the program continues executing. While this reduced latencies, GC still depended on handshakes for progress making it not lock-free. Ben-Ari (1982) added support for moving GCs to stacklets. Recent work (Kliot et al. 2009) presents what they call lock-free root scanning. In this case, "lock-free" merely means that a concurrent GC thread can help the mutator scan its thread putting more of the work load on another thread, in a lock-free fashion. However, the GC still needs to wait for a blocking handshake from all mutator threads to finish stack scanning, making the GC in fact not lock-free or even non-blocking. This is good for low latency, but it is not a non-blocking solution due to the reliance on blocking handshakes.

Handling Fragmentation: There are GCs such as Schism (Pizlo et al. 2010) and JamaicaVM (Siebert 2007) that limit fragmentation without compaction. Instead they use a data layout of objects that is immune to fragmentation.

What they have in common is that they add memory and performance overheads due to indirections. It also becomes impractical to expose objects to already defined native interface that assumes more intuitive memory layouts.

Incremental Compaction: Metronome (Bacon et al. 2003) is an important low-latency GC. It uses a Brooks-style (Brooks 1984) read barrier. Cells are moved from pages with low occupancy to pages with high occupancy in incremental safepoints.

G1 in OpenJDK (Detlefs et al. 2004) tries to hold safepoints below a configurable pause time. A card marking post write barrier logs all interregional references. Regions are then evacuated in safepoints using remembered sets for each region.

Memory Protection Compaction: Baker's algorithm was the first algorithm to use a to-space invariant that dictated that mutators could only see cells in to-space. This only worked on single processor systems. The work of Appel et al. (1988) was first to enforce this invariant using `mprotect` to perform compaction (Baker 1978). It protects the whole from-space visible to mutators, and then helps moving objects over to to-space as mutators trap. This extended the approach to work on multiprocessor systems.

Similarly, the Pauseless GC (Click et al. 2005) uses page protection to protect from-space from accesses and makes sure mutations only happen in to-space after the cells have been copied. The advantage is that there is no global safepoint issued stopping all mutators from progressing. C4 (Tene et al. 2011) is a generational extension of Pauseless. The Compressor (Kermany and Petrank 2006) uses page protection to align mutators in a similar way. Unfortunately, using `mprotect` is not block-free because it requires non-preemptive kernel locks, violating scheduling independence.

Replicating Compaction: The replicating GC introduced by Nettles and O'Toole (1993) and O'Toole and Nettles (1994) 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 (Hudson and Moss 2001) uses a similar approach, but improves on being more incremental and more scalable (yet not non-blocking) in synchronization. Ritson et al. (2014) added transactional memory to the Sapphire approach, which resulted in improved performance.

Non-blocking Compaction: Chicken (Pizlo et al. 2008) and Staccato (McCloskey et al. 2008) install forwarding pointers concurrently to the to-space version of a cell. To enforce mutator alignment, their barriers immediately cancel any ongoing compaction before proceeding with mutations.

The Collie (Iyengar et al. 2012) relies on hardware transactional memory (HTM) to completely relocate objects one by one.

Clover (Pizlo et al. 2008) picks a randomly generated 64-bit value to mean that copying has finished, hoping it is not written. The algorithm does not guarantee that copying finishes.

Stopless (Pizlo et al. 2007) uses double-wide CAS to copy objects in a lock-free fashion for the mutator.

What these algorithms have in common is their reliance on blocking handshakes to finish before copying. This breaks non-blocking guarantees for the GC thread.

A low-latency GC was proposed by Österlund and Löwe (2015). The idea is to use a Field Pinning Protocol (FPP) for protecting field accesses while allowing both mutator and GC to progress with a known upper bound on copy failures. This was the first solution to be lock-free for both mutator and GC threads. While FPP using expensive fences is lock-free, the proposed high-throughput ADS optimization might not be, depending on the choice of the definition of "lock-free". The current paper introduces a block-free, low-latency and high-throughput optimization of FPP.

Manual Approaches: The Eventron (Spoonhower et al. 2006) is a Java based real-time programming construct that coexists with GC, allowing manually picked objects to be allocated outside of the heap. This allows parts of the application to run with very low latency independently of the rest of the GC solution.

Detlefs et al. (2001) provide a solution for lock-free reference counting GC when the DCAS hardware primitive is available; unfortunately it seldom is. Sundell (2005) presented a wait-free reference counting GC for commodity hardware.

8. Conclusion

The main contribution of this paper is a block-free handshake that allows previous on-the-fly GC algorithms using blocking handshakes to become block-free. Removing the blocking handshakes is necessary for making a completely non-blocking GC, both for mutator and GC threads. This is the first paper to address this fundamental problem and show how reliance on blocking handshakes can be removed.

We present block-freedom, a new non-blocking progress property with operation level progress guarantees. It is closely related to lock-freedom, but less restrictive: it allows finite waiting for threads that are running on other processors, but not threads that have been suspended by the OS. It also allows communication with the OS as long as the algorithm is scheduling independent.

Block-free handshakes were applied to two applications: block-free stack scanning and block-free object copying. Block-free stack scanning is the first stack scanning algorithm that has non-blocking guarantees both for mutators and GC. The copying algorithm builds on the field pinning protocol by Österlund and Löwe (2015), a concurrent lock-free copying algorithm that was non-blocking both for mutator and GC. Improved with block-free handshakes it gives even block-free progress guarantees while enabling lighter field pinning barriers using deferred field pinning.

At a cost of 15% lower throughput, GC pause times were reduced down to 12.5% compared to the original G1 on average in DaCapo. Especially, in the memory intense h2 benchmark, latencies improved from 174 ms to 0.67 ms for the 99.99% percentile.

For future work, we would like to make reference processing concurrent and block-free, and make all internal data structures block-free. Then a completely block-free GC could be made.

Acknowledgments

We thank Dave Dice for fruitful discussions that inspired this work. He also advised us to introduce the notion of block-freedom instead of overloading previous notions of lock- and wait-freedom with yet another definition. We also thank the anonymous reviewers and Ben Titzer for the constructive feedback that helped improve the paper.

References

A. W. Appel, J. R. Ellis, and K. Li. Real-time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 11–20, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.53992. URL http://doi.acm.org/10.1145/53990.53992.

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.

H. G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Commun. ACM*, 21(4):280–294, Apr. 1978. ISSN 0001-0782. doi: 10.1145/359460.359470. URL http://doi.acm.org/10.1145/359460.359470.

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

M. Ben-Ari. Algorithms for On-the-fly Garbage Collection. *ACM Trans. Program. Lang. Syst.*, 6(3):333–344, July 1984. ISSN 0164-0925. doi: 10.1145/579.587. URL http://doi.acm.org/10.1145/579.587.

S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL http://doi.acm.org/10.1145/1167473.1167488.

G. E. Blelloch and P. Cheng. On Bounding Time and Space for Multiprocessor Garbage Collection. *SIGPLAN Not.*, 34(5):104–117, May 1999. ISSN 0362-1340. doi: 10.1145/301631.301648. URL http://doi.acm.org/10.1145/301631.301648.

H.-J. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 197–206, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155109. URL http://doi.acm.org/10.1145/155090.155109.

H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 93–100, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503282. URL http://doi.acm.org/10.1145/503272.503282.

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*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802042. URL http://doi.acm.org/10.1145/800055.802042.

P. Cheng and G. E. Blelloch. A Parallel, Real-time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378823. URL http://doi.acm.org/10.1145/378795.378823.

P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-driven Pretenuring. *SIGPLAN Not.*, 33(5):162–173, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277718. URL http://doi.acm.org/10.1145/277652.277718.

C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064988. URL http://doi.acm.org/10.1145/1064979.1064988.

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*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96735. URL http://doi.acm.org/10.1145/96709.96735.

D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029879. URL http://doi.acm.org/10.1145/1029873.1029879.

D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free Reference Counting. pages 190–199, 2001. doi: 10.1145/383962.384016. URL http://doi.acm.org/10.1145/383962.384016.

D. Dice, H. Huang, and M. Yang. Techniques for accessing a shared resource using an improved synchronization mechanism, Jan. 5 2010. URL http://www.google.com.ar/patents/US7644409. US Patent 7,644,409.

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM*, 21(11):966–975, Nov. 1978. ISSN 0001-0782. doi: 10.1145/359642.359655. URL http://doi.acm.org/10.1145/359642.359655.

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*, POPL '94, pages 70–83, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: 10.1145/174675.174673. URL http://doi.acm.org/10.1145/174675.174673.

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*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158611. URL http://doi.acm.org/10.1145/158511.158611.

T. Domani, E. K. Kolodner, and E. Petrank. A Generational On-the-fly Garbage Collector for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 274–284, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349336. URL http://doi.acm.org/10.1145/349299.349336.

K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: 10.1145/949305.949340. URL http://doi.acm.org/10.1145/949305.949340.

M. Herlihy and N. Shavit. On the nature of progress. In A. Fernàndez Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25872-5. doi: 10.1007/978-3-642-25873-2_22. URL http://dx.doi.org/10.1007/978-3-642-25873-2_22.

M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM. ISBN 1-58113-708-7. doi: 10.1145/872035.872048. URL http://doi.acm.org/10.1145/872035.872048.

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*, JGI '01, pages 48–57, New York, NY, USA, 2001. ACM. ISBN 1-58113-359-6. doi: 10.1145/376656.376810. URL http://doi.acm.org/10.1145/376656.376810.

B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: A Wait-free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259009. URL `http://doi.acm.org/10.1145/2258996.2259009`.

T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cdx: A family of real-time java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-732-5. doi: 10.1145/1620405.1620412. URL `http://doi.acm.org/10.1145/1620405.1620412`.

H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134023. URL `http://doi.acm.org/10.1145/1133981.1134023`.

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*, VEE '09, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508296. URL `http://doi.acm.org/10.1145/1508293.1508296`.

A. Kogan and E. Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. *SIGPLAN Not.*, 46(8):223–234, Feb. 2011. ISSN 0362-1340. doi: 10.1145/2038037.1941585. URL `http://doi.acm.org/10.1145/2038037.1941585`.

A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 141–150, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145835. URL `http://doi.acm.org/10.1145/2145816.2145835`.

T. F. Lim, P. Pardyak, and B. N. Bershad. A Memory-efficient Real-time Non-copying Garbage Collector. *SIGPLAN Not.*, 34(3):118–129, Oct. 1998. ISSN 0362-1340. doi: 10.1145/301589.286873. URL `http://doi.acm.org/10.1145/301589.286873`.

B. McCloskey, D. F. Bacon, P. Cheng, and D. Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Technical report, Technical Report RC24505, IBM Research, 2008.

M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6): 491–504, June 2004. ISSN 1045-9219. doi: 10.1109/TPDS.2004.8.

M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1 – 26, 1998. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1998.1446. URL `http://www.sciencedirect.com/science/article/pii/S0743731598914460`.

S. Nettles and J. O'Toole. Real-time Replication Garbage Collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 217–226, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155111. URL `http://doi.acm.org/10.1145/155090.155111`.

E. Österlund and W. Löwe. Concurrent Compaction Using a Field Pinning Protocol. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2015, pages 56–69, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754177. URL `http://doi.acm.org/10.1145/2754169.2754177`.

J. O'Toole and S. Nettles. Concurrent Replicating Garbage Collection. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 34–42, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182425. URL `http://doi.acm.org/10.1145/182409.182425`.

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*, ISMM '07, pages 159–172, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: 10.1145/1296907.1296927. URL `http://doi.acm.org/10.1145/1296907.1296927`.

F. Pizlo, E. Petrank, and B. Steensgaard. A Study of Concurrent Real-time Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375587. URL `http://doi.acm.org/10.1145/1375581.1375587`.

F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant Real-time Garbage Collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806615. URL `http://doi.acm.org/10.1145/1806596.1806615`.

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*, ISMM '14, pages 105–115, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2921-7. doi: 10.1145/2602988.2602992. URL `http://doi.acm.org/10.1145/2602988.2602992`.

B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1123001. URL `http://doi.acm.org/10.1145/1122971.1123001`.

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*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-813-8. doi: 10.1145/1288940.1288954. URL `http://doi.acm.org/10.1145/1288940.1288954`.

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.

G. L. Steele, Jr. Multiprocessing Compactifying Garbage Collection. *Commun. ACM*, 18(9):495–508, Sept. 1975. ISSN 0001-0782. doi: 10.1145/361002.361005. URL `http://doi.acm.org/10.1145/361002.361005`.

H. Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24b–24b, April 2005. doi: 10.1109/IPDPS.2005.451.

G. Tene, B. Iyengar, and M. Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993491. URL `http://doi.acm.org/10.1145/1993478.1993491`.

T. Ugawa, R. E. Jones, and C. G. Ritson. Reference Object Processing in On-the-fly Garbage Collection. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 59–69, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2921-7. doi: 10.1145/2602988.2602991. URL `http://doi.acm.org/10.1145/2602988.2602991`.

T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181 – 198, 1990. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/0164-1212(90)90084-Y. URL `http://www.sciencedirect.com/science/article/pii/016412129090084Y`.