Collecting Garbage on the Blockchain

Luc Bläser
luc.blaeser@dfinity.org
DFINITY Foundation
Switzerland

Claudio Russo
claudio.russo@dfinity.org
DFINITY Foundation
Switzerland

Ulan Degenbaev
ulan.degenbaev@dfinity.org
DFINITY Foundation
Switzerland

Ömer S. Ağacan
omersa@google.com
Google
Denmark

Gabor Greif
gabor.greif@dfinity.org
DFINITY Foundation
Switzerland

Jason Ibrahim
jason.ibrahim@dfinity.org
DFINITY Foundation
Switzerland

Abstract

We present a garbage collector that is specifically designed for a WebAssembly-based blockchain, such as the Internet Computer. Applications on the blockchain implement smart contracts that may have indefinitely long lifetime and may hold substantial monetary value. This imposes a different set of requirements for garbage collection compared to traditional platforms. In this paper, we explain the differences and show how our garbage collector optimizes towards these goals.

CCS Concepts: • **Software and its engineering → Garbage collection**.

Keywords: Garbage Collection; WebAssembly; Blockchain

ACM Reference Format:
Luc Bläser, Claudio Russo, Ulan Degenbaev, Ömer S. Ağacan, Gabor Greif, and Jason Ibrahim. 2023. Collecting Garbage on the Blockchain. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623507.3627672

1 Introduction

There is a recent trend towards compute blockchains that enable fully decentralized applications [2]. Such blockchains implement a virtual machine that run programs by secure distribution across untrusted computer nodes. This is achieved by replicating the program memory and execution and applying Byzantine fault-tolerant consensus on the computation [7].

To ease the consensus, program execution is typically represented as transactions that run a sequence of operations in a deterministic, isolated, and atomic way. Transactions have access to the program memory that is persisted on the blockchain and that they can read and modify. The final memory state of a transaction thereby serves as initial memory state for a subsequent program transaction.

In contrast to traditional applications, blockchain programs often implement smart contracts that hold monetary value and therefore require a particularly high assurance and reliability. They also tend to have much longer lifetime, with some applications staying on the blockchain indefinitely and continuously executing transactions. These concerns favor high-level programming languages that prioritize safety, by offering for example automatic memory management.

Memory reclamation is an important aspect on a blockchain as it allows recycling space in the program's memory, such that a transaction does not always need to grow the size of its memory for new allocations. Although a blockchain may store a history of memory images of previous transactions, garbage collection helps to reduce the program's *latest* heap and memory image. Modern blockchains can even securely truncate state history, such that old transactional memory snapshots including the state of garbage objects are discarded after some time [2]. That allows programs to have large memory in the order of gigabytes.

In contrast to classical garbage collection, a blockchain imposes stricter requirements on memory management:

- **Fragmentation**: Since an application can live perpetually on a blockchain, GC should avoid heap fragmentation. Simply restarting an application may not be an option.
- **Incrementality**: As each transaction on a blockchain needs to complete in a limited amount of time, a GC has to bound all of its pauses.

Many blockchains choose WebAssembly (Wasm) [14] as their virtual machine because it is open, formally specified, efficient, and secure. Moreover, Wasm is easily made deterministic and is targeted by a wide range of programming languages. However, currently, Wasm does not incorporate a garbage collector but only exposes linear memory. There is an upcoming Wasm GC proposal [15] to introduce host

VMIL '23, October 23, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0401-7/23/10.
https://doi.org/10.1145/3623507.3627672

garbage collection. It is not clear that blockchains can leverage this proposal: Blockchain nodes would need to reach consensus on the implementation specific, native state of the host GC rather than just the concrete Wasm state. It is also unlikely that any off-the-shelf Wasm engine will expose the private state of its GC to clients. Even if it did, the host's GC may still not meet our fragmentation and incrementality requirements.

Another approach is to reuse the existing GC of a managed language that compiles to Wasm. As the GCs of mainstream languages are typically designed for traditional computing, they do not suit our requirements. For example, Java GCs implement compaction but may sometimes perform unbounded stop-the-world operations in certain GC stages, see also Section 6. Also, these GCs use multi-threading which is not available in the deterministic context of blockchain execution.

Lacking an existing solution, we have designed a new GC that is tailored to a Wasm-based blockchain. Inspired by the Shenandoah GC [11], our GC also engages incremental snapshot-at-the-beginning marking, incremental partitioned evacuation-compaction, and incremental pointer updates based on the Brooks forwarding pointers [6]. It additionally avoids any unbound stop-the-world operations and is tuned to more conservative memory reclamation on the blockchain.

We implemented the GC for the programming language Motoko [12] and evaluated it on the Internet Computer blockchain [2].

In summary, we make the following contributions:

- The design of a Wasm-based garbage collector that is optimized for a blockchain.
- The open-source implementation of this GC.
- Performance evaluation of the GC in comparison to classical GCs.

The remainder of this paper is organized as follows: Section 2 provides some background information about the blockchain we build upon. Section 3 describes the GC design, giving an overview before discussing the specific aspects. Section 4 reports on the implementation aspects and tuning parameters of the GC. Section 5 shows experimental results when using the GC and comparing it to other GCs. Section 6 discusses related work. Section 7 concludes this paper.

2 Background

The GC presented in this paper manages the memory of the Motoko programming language that runs on the Internet Computer blockchain. We briefly provide some background information to support the subsequent explanation of the GC design.

The Internet Computer (IC) [2] is a compute blockchain whose software components, called canisters, follow the Actor Model [16] and are implemented in Wasm. The communication across canisters as well as external input/output with the canisters happens exclusively through message passing (not shared state). A message sent to a canister is asynchronously received by that canister which can then trigger Wasm code to run for that message. The code may change the canister state and send messages asynchronously, e.g. a result to the sending canister, messages to other canisters, or even messages to itself. The execution implemented on message reception constitutes a transaction that runs deterministically, atomically (may either succeed or roll back), in isolation (sequential message processing inside a canister), and with a maximum limit of computation steps. The IC measures costs in terms of both memory usage (Wasm allocated memory size) and synthetic counting of the executed instructions. According to the concept of orthogonal persistence [3], canisters run conceptually perpetually, meaning that their last memory state is automatically preserved on the blockchain. Canisters can also be upgraded to new program versions requiring state migration between upgrades.

Motoko [12] is a relatively young programming language designed to optimally support the IC runtime model and thus ease development on this blockchain. Besides a blend of imperative, functional, object-oriented, and asynchronous programming concepts, the language features less mainstream concepts, such as structural typing, orthogonal persistence, and the Actor Model. The latter two are tailored to the Internet Computer blockchain. Motoko rigorously ensures type and memory safety, by relying on a garbage collector for automatic memory reclamation.

3 GC Design

Incremental and compacting collection are key design properties of our GC that is tailored to the blockchain.

The GC distributes its workload across multiple increments where the mutator is paused for only a limited amount of Wasm instructions. This is a hard limit, meaning it does not depend on the heap size, stack size, GC phase, and memory constellation. As a result, the GC runs concurrently to the mutator with deterministic interleaving, allowing scalable heap usage. Parallelism is not supported by the blockchain as it may imply non-deterministic differences between the replicated execution. Each GC work fits within the instruction-limited IC transactions.

Similar to the Shenandoah GC [11], our incremental GC organizes the heap in equally-sized partitions and selects high-garbage partitions for compaction by using incremental evacuation. We were able to adopt most of the Shenandoah design except that we eliminate remaining unbounded stop-the-world GC phases and insert an extra updating phase. We use the Brooks forwarding pointer technique [6], while Shenandoah later changed to conditional-jump load barriers [19]. The differences are discussed in Section 6.

In overall, the GC runs in three phases, all fully incremental:

1. **Marking**: The GC performs full-heap incremental marking with the snapshot-at-the-beginning algorithm.
2. **Evacuation**: The GC moves live objects in selected partitions to new empty partitions. Accesses to relocated objects are atomically redirected by the forwarding pointers.
3. **Updating**: The GC traverses the heap and updates all pointers to relocated objects, before freeing the memory of the evacuated partitions.

The GC phases and other design aspects are explained in more detail in the subsequent sections.

3.1 Marking Phase

When a new GC run is scheduled, the incremental marking phase is first initiated. This involves an incremental tri-color-marking [18] based on the snapshot-at-the-beginning (SATB) algorithm, operating on the entire heap (across partitions). Full-heap marking has the advantage that it can also deal with arbitrarily large cyclic garbage, even if it is spread over multiple partitions. Applying a different, more progressive marking scheme, such as incremental update, is difficult on Wasm, as is explained in Section 3.8.

To realize SATB consistency, write barriers intercept mutator pointer overwrites between the GC mark increments. The barrier marks the old target of an overwritten pointer (deletion barrier). Concurrent object allocations, i.e. new objects created during a GC run, are always conservatively marked. The marking phase is only initiated on an empty call stack, which is the case at the end of an IC transaction. We discuss this design choice in Section 3.8. Therefore, the marking phase can start from a fixed set of global root pointers.

Our GC uses partition-associated mark bitmaps that are temporarily allocated during a GC run. This proved to be significantly more efficient than inlining the mark bit in the object header, with a difference of 38% of the GC's runtime costs. We try to avoid running out of memory due to the mark bitmap allocation by lowering the critical memory limit for GC scheduling accordingly, cf. Section **??**. The marking phase additionally needs a mark stack that we implemented as a growable linked table list in the heap that can be recycled as garbage during the active GC run.

As a side activity, the marking phase also records the amount of live data per partition. These statistics later serve as a basis for deciding on selective evacuation.

3.2 Evacuation Phase

The GC prioritizes partitions with larger amounts of garbage for evacuation using the statistics gathered by the marking phase. Thereby, the GC also limits the amount of selected partitions based on the available overall free memory space. For efficiency, a partition is only elected for evacuation if it contains more than a defined minimum amount of garbage.

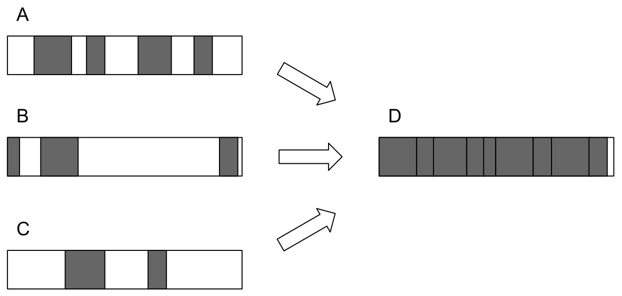

Figure 1. Compacting partitions through evacuation

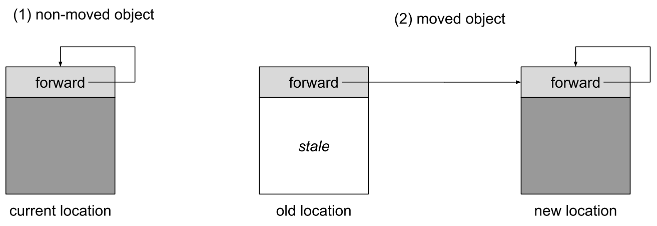

Figure 2. Brooks pointer

Once the selection step is done, the marked objects inside the selected partitions are evacuated to other free partitions and thereby compacted. Figure 1 visualizes how the live objects of partitions A, B, and C are compacted into partition D.

To allow incremental object moving and incremental updating of pointers, each object carries redirection information in its header, namely, a Brooks forwarding pointer [6]. Figure 2 illustrates the mechanism: For a non-moved object (1), the forwarding pointer refers to the object itself, while for moved objects (2), the forwarding pointer refers to the new object location. The runtime system ensures that each object access is redirected via this forwarding pointer. This constitutes a very efficient version of a load barrier, only involving an additional unconditional load instruction. Motoko does not support reference equality checks, which would otherwise also require forwarding pointer resolution.

The evacuated partitions are retained and whenever an object is evacuated, the original location is forwarded to the corresponding new object location. Therefore, the mutator can continue to use old incoming pointers to evacuated objects. To avoid additional write barriers during evacuation, an object is entirely copied without mutator interference. Section 3.6 elaborates on how this complies to strict increment limits.

3.3 Updating Phase

All pointers to any moved objects have to be updated before free space can be reclaimed. For this purpose, the GC performs the third phase that traverses all reachable objects in the entire heap and updates all pointers from live objects to their forwarded addresses.

As the mutator may perform concurrent pointer writes behind the traversal frontier of the updating phase, a write barrier catches the written pointer values and resolves them to the corresponding forwarded locations. The same applies to new object allocations that may contain old pointer values in their initialized state, e.g. originating from the call stack.

The updating phase can only be completed when the call stack is empty, since the GC does not (and cannot) access the Wasm stack, similar to the start of the GC marking phase.

Once the updating phase is completed, all evacuated partitions are freed and can later be recycled for new object allocations. At the same time, the GC also frees the mark bitmaps stored in temporary partitions.

Unlike the Shenandoah GC, our GC does not postpone the pointer updates to the marking phase of the next GC run, to reduce the latency of memory reclamation. Otherwise, the evacuated partitions would need to be retained until the end of the next marking phase and there can be a longer pause until the next GC run is scheduled.

3.4 Object Allocation

Object allocation is realized by bump allocation inside a selected allocation partition. When the allocation partition's free space is insufficient for a new allocation, the free pointer is moved to a new free partition which then becomes the allocation partition. Large objects are handled specially, see Section 3.5. Because of the required determinism on the blockchain, there is no multi-threading, such that bump allocation with a single free pointer is an efficient choice.

As mentioned, we need to handle mutator allocations between the GC increments. We call these concurrent allocations as they are time-multiplexed with the GC run. An allocation barrier catches all newly created (and initialized) objects and does the following:

- If the GC is in the marking phase, the new object is marked (conservative incremental marking).
- If the GC is in the updating phase, the fields inside the initialized object are inspected and potential old pointers updated to the new forwarded location.
- The GC counts concurrent allocations to adapt its work intensity to the allocation rate, see Section 3.7.

3.5 Large Objects

An object larger than a partition is deemed *large* and requires special handling: A sufficient amount of contiguous free partitions is searched and reserved for such a large object. Large objects are not moved by our GC. Once they become garbage, having been left unmarked by the GC, their hosting partitions are immediately freed. Partitions storing large objects do not require a mark bitmap during the GC but use a single mark bit inside the metadata of a partition. Both external and internal fragmentation can occur for large objects which is an open design shortcoming of our GC. We therefore advise programmers to carefully allocate such special objects.

3.6 Increment Limit

The GC maintains a synthetic deterministic clock by counting work steps, such as marking an object, copying a word, or updating a pointer. The clock serves for limiting the duration of a GC increment. The GC increment is stopped whenever the limit is reached, ceding control to the mutator until the GC later resumes its work in a new increment. The clock is calibrated to linearly correlate to the IC execution costs.

To also respect the limit on large objects, large arrays are marked and updated in incremental slices. Other large objects, such as *blobs*, do not contain pointers in their payload. The maximum uninterrupted amount of work in the GC is moving an object, and since large objects stay in place, this is limited by the copying costs of the memory of an entire partition. With this design, the GC is incremental in all phases and work steps, assuming a minimum bound on the increment limit that depends on the partition size.

Currently, our GC does not provide a strict guarantee that it can always reclaim free space in time. However, we adapt GC work to the allocation rate, see Section 3.7, which according to our experimental tests serves for a timely reclamation.

3.7 Adaptive GC Work

To reduce the reclamation latency when the mutator allocates at a high rate during garbage collection, the GC imposes an additional small GC increment per concurrent allocation. However, instead of performing the increment immediately on each allocation, the allocation-attributed increment time is added to the next regularly scheduled GC increment. This GC increment is performed at the end of the mutator code of an executed message that constitutes a transaction.

In our runtime model, one cannot observe external effects during the execution of a transaction (or message). Hence postponing the allocation increments is equivalent to interleaving them in a more fine-grained manner inside the transaction. Therefore, even if the GC increment limit is extended at the execution level, it does not contravene the strict incrementality needed for the blockchain: The programmers have it under their control. If the extended GC increment exceeds the transaction limit, there is too high allocation work in that transaction, and the execution would equally have crossed the execution limit had the small GC allocation increments been performed at each allocation. In other words, the GC increment limit depends on the amount of mutator allocations performed since the last GC increment inside the same transaction, but remains independent of the heap size or memory structure.

3.8 Root Set

A GC run is only started and finished when the call stack is empty. This approach is chosen because of a Wasm security restriction that prevents the inspection of the native call stack for the root set collection. In our case, this limitation is not so decisive, since the call stack is empty at the end of each transaction. Not having to scan the call stack even benefits incrementality: Contrary to other GCs, e.g. the Shenandoah GC, we only have a constant set of root pointers, and avoid stop-the-world initialization and finalization in a simple way. The downside of this approach is that the marking phase must be conservative by retaining new allocations and using SATB. A more progressive marking scheme, such as incremental update, is not feasible if we want to switch GC phases during the same transaction. When the GC switches from the marking to the evacuation phase, we could have critical pointers on the stack that lead to unmarked objects and we could not detect those. An alternative would be to maintain a separate custom shadow stack [4] for local pointers which incurs performance overheads.

4 Implementation

Currently, the IC offers a 32-bit Wasm heap per canister as the current Wasm standard is also still based on 32 bit. There are plans to extend the heap to 64 bit in the future by using the Wasm64 extension. (Currently, programs can already use special 64-bit memory that is explicitly accessed and currently not relevant for the GC.) The Motoko language implementation consists of a compiler implemented in OCaml and a runtime system implemented in Rust, both targeting Wasm as binary code. The GC is part of the pre-compiled runtime system library that is linked into a Motoko-compiled user program. The compiled size of the incremental GC (excluding the remaining logic of the runtime system) is moderate with 2.3 KB.

4.1 Partitioned Memory

Our GC divides the memory address space in 32 MB logical partitions. The physical allocation of partitions (as sequences of 64 KB Wasm pages) happens lazily as more partitions are needed. We measured that the granularity of 32 MB is optimal in terms of both memory and execution costs, although the differences are small compared to using 64 MB or 16 MB. Smaller partition sizes cause more frequent handling of large objects, while larger partition sizes grow the heap in coarser granularity and also increase the lower bound for the configurable GC increment limit.

Figure 3 depicts the structure of the partitioned memory. Some partitions store regular-sized objects (1, 2), while others are free (7, 8) or host part of a large object (3, 4). One partition serves as the current allocation target for the regular-sized objects (5), and during garbage collection, mark bitmaps are placed in dedicated partitions (6).

Each partition requires small metadata, such as a flag to denote whether it is free, the size of allocated and marked

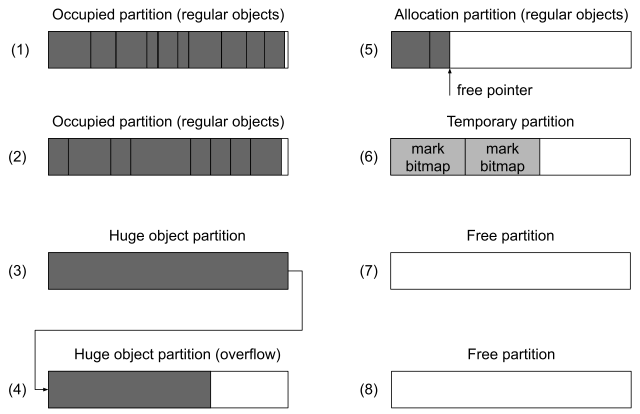
Occupied partition (regular objects)
(1)
Occupied partition (regular objects)
(2)
Huge object partition
(3)
Huge object partition (overflow)
(4)
Allocation partition (regular objects)
(5)
free pointer
Temporary partition
(6)
mark bitmap
mark bitmap
Free partition
(7)
Free partition
(8)

Figure 3. Partitioned memory

space, a pointer to a possible mark bitmap, and a flag specifying whether the partition belongs to a large object allocation. Currently, with the 32-bit address space, this metadata is simply stored as a static partition table. When porting the system to 64 bit, this metadata could be inlined to the partitions.

Allocating a new empty partition or finding sufficiently large contiguous space for a large object involves a simple linear search of the partition table. While this is efficient enough in 32-bit space, one could introduce a free list of partitions in the future 64-bit version.

4.2 Barrier Implementation

The incremental GC relies on three kinds of barriers:

- **Load barrier**: This resolves the forwarding pointer and only involves a single additional load instruction with an offset. Despite its lean implementation, we measured a still noticeable overhead of about 12% on average with regard to the instruction costs on the IC blockchain.
- **Write barrier**: This supports the SATB marking and updating of old pointers. The measured runtime overhead is relatively low with 2%.
- **Allocation barrier**: This serves for updating the pointers inside the initialized objects and for counting the concurrent allocations. The allocation barrier causes around 3% runtime overhead.

Fortunately, the costs of the barriers are again compensated by the main GC work, such that the incremental GC eventually outperforms other stop-the-world GCs, as discussed in Section 5.

4.3 Scheduling

We schedule a new GC run whenever the heap size has grown by more than 65% since the previous GC run or, if it is the first GC run, when the heap has grown beyond the partition size. When passing a critical memory limit of 3.25 GB on the 4 GB address space, we increase the GC frequency considerably, triggering a new GC run on every 1% growth. This critical limit is set low enough to consider the size of the mark bitmap that needs to be allocated during garbage collection. Allocations during an active GC run account for the growth threshold of the subsequent GC run.

A GC increment is bounded to 3,500,000 steps, as counted by the synthetic clock in the GC. This corresponds to about 600 to 700 million instructions on the IC, where the transaction limit is currently around 4 billion instructions. Each concurrent mutator allocation incurs a small GC increment of 20 steps. In our runtime model, the small increments can be simply added to the next regularly scheduled GC increment.

The compiler inserts GC scheduling calls at the end of a message execution (when the call stack is also guaranteed to be empty). These calls eventually perform a GC increment unless the heap growth is too little since the last completion of a GC run.

4.4 Evacuation Policy

Partitions with very little garbage, less than 15%, are never evacuated. Moreover, we precompute the amount of extra copy space that the evacuations need. If memory is scarce, we limit the number of evacuated partitions by prioritizing partitions with higher amounts of garbage. This prevents the GC from running out of memory in the case of a high load of evacuations.

5 Experimental Results

We measure the new GC on the IC blockchain. Cost metrics are thereby different to classical computer systems: On the IC, the dimensions of performance are the size of allocated Wasm memory and the number of executed Wasm instructions. Moreover, adherence to the transaction instruction limit is essential for scaling on the blockchain.

We compare our new GC, called the **incremental GC** in this section, with three other available garbage collectors that also have been implemented for Motoko and are available to the users:

- **Copying GC**: A two-space copying garbage collector.
- **Compacting GC**: A mark-and-compact garbage collector using the pointer threading technique [18].
- **Generational GC**: A two-generation compacting GC, performing frequent collection of young objects.

In contrast to the new incremental GC, these three GCs block the mutator for their entire work. Only the generational GC is able to reduce pauses when collecting the smaller young generation.

5.1 Performance Measurements

We assembled a performance benchmark of several Motoko programs, comprising user applications, a sample application set, as well as some data structure tests. The latter perform a series of insertions, reads, and deletions, including a scenario that clears and repopulates the structure. Table 1 lists the different programs of our GC performance benchmark, together with a brief description, and the total amount of allocated memory without garbage collection.

Table 1. Performance benchmark overview

Benchmark	Description	Allocations
asset-storage	Storing, listing and clearing 1 MB BLOBs, 70 rounds	281 MB
blobs	Managing 64 KB blobs in a list (insert, read, delete, repopulate)	2126 MB
btree-map	Storing numbers in a b-tree (insert, read, delete, repopulate)	397 MB
buffer	Storing numbers in an array list (insert, read, delete, repopulate)	123 MB
cancan	Video-sharing platform, storing videos that are chunked into 1MB blobs	805 MB
extendable-token	Generic token exchange library, measuring 200 repeated transfers	2 MB
game-of-life	Sample application, game of life simulation with 10 rounds using a 512 x 512 grid	318 MB
graph	Fully created graph (insert, read, delete, repopulate)	275 MB
imperative-rb-tree	Storing numbers in a mutable red-black tree (insert, read, delete, repopulate)	148 MB
linked-list	Storing numbers in a linear linked list (insert, read, delete, repopulate)	413 MB
qr-code	QR code generator, seeded pseudo-randomized, 3 rounds	920 MB
random-maze	Sample application, maze generation, 25 rounds with 10, 100 and 200 cells, seeded pseudo-randomized	27 MB
rb-tree	Storing numbers in a functional implementation of a red-black tree (insert, read, delete, repopulate)	920 MB
reversi	Sample application, reversi game, 30 iterations	2 MB
scalable-buffer	Multi-level array list used for storing numbers (insert, read, delete, repopulate)	145 MB
sha256	Computing sha256 hashes on 64KB data, 10 iterations	1290 MB
trie-map	Storing numbers in a functional implementation of a trie data structure (insert, read, delete, repopulate)	1026 MB

Some smaller-sized benchmark cases do not trigger the GC, e.g. 'extendable-token' and 'reversi'. We included these cases in the benchmark to also measure the runtime overheads of barriers and instrumented code in simple cases.

The benchmark is run on a local replicated runtime environment by using the IC-custom runtime tool called 'dfx', version 0.13.1. Due to the deterministic execution property of the blockchain, all runtime properties and measured results are identical to the production IC environment.

5.1.1 Maximum GC Pause. An important property of our GC is the incrementality, which is reflected by the maximum GC pause. Figure 4 shows the number of Wasm instructions of the longest GC work per transaction, measured for each benchmark case and for each of the four GCs. The horizontal line indicates the intended pause limit for the new incremental GC of 700 million Wasm instructions. While most cases are uncritical with regard to GC pauses, the simple blocking GCs (compacting GC and copying GC) suffer from considerably long pauses in certain scenarios, consuming up to 1.19 billion Wasm instructions. As for our new GC, pauses are only extended beyond the base limit if the mutator entails a

Table 2. Comparison of the GC pauses

GC	Maximum Pause (million Wasm instructions)	Average Pause (million Wasm instructions)
Incremental GC	712	15
Generational GC	1190	27
Compacting GC	8410	67
Copying GC	5900	60

Table 3. GC performance comparison

GC	Total runtime (billion Wasm instructions)	GC throughput (mutator utilization)	Allocated Wasm memory
Incremental GC	18.5	79.7%	296 MB
Generational GC	19.1	79.0%	191 MB
Compacting GC	22.0	75.0%	188 MB
Copying GC	20.5	77.9%	271 MB

lot of concurrent allocations within a blockchain transaction, cf. Section 3.7.

Table 2 summarizes the maximum and average duration of GC pauses in million Wasm instructions, as measured across all performance benchmark cases. In situations of small program heaps, the GC can run fast, such that the average is clearly shorter than the worst case of the longest pause. As can be seen, the new incremental GC imposes shorter pauses with regard to both the maximum and average duration.

5.1.2 Runtime Costs. The number of executed Wasm instructions represents the execution costs of the programs on the IC. The second column of Table 3 compares the average runtime costs (mutator and GC work) in billion Wasm instructions for the different GCs in the performance benchmark. Although performance is not a primary design goal of our new GC, it even offers the most economic execution on average. Compared to the copying GC, we save around 10% of the instructions. This is primarily because the new GC performs selective compaction of high-garbage partitions, while the copying GC moves the entire live set of the heap on each run. The runtime improvement is less significant when compared to the generational GC (a 3% saving), as the latter is optimized for faster execution by primarily concentrating on the small young generation.

Figure 5 details the runtime costs for the different benchmark cases, also in the number of Wasm instructions. The new GC's selective evacuation mechanism pays off in more coarse-grained cases of 'blob' and 'cancan' that deal with larger objects. Moreover, examples that fit well for the generational GC, also profit during the incremental GC, e.g. 'trie-map' and 'imperative-rb-tree'. However, there also exist other cases where the new incremental GC degrades the runtime performance, such as for 'scalable-buffer' and 'sha256', where the barrier overheads become noticeable.

5.1.3 GC Throughput. Another perspective on the GC efficiency can be obtained by measuring the GC throughput, also called the mutator utilization, being the fraction of the mutator execution costs to the total program execution costs. The results are listed in the third column of Table 3. With a slightly higher mutator utilization of 0.7% to 4.7%, our new incremental GC causes less disruptions on the mutator.

5.1.4 Allocated Memory. The IC uses the allocated Wasm memory size of a program (canister) as a metric of cost calculation. The last column of Table 3 summarizes the average allocated memory size of the GCs for the performance benchmark. We additionally measured the memory size without garbage collection, reflecting the artificial case where objects would never be freed.

The memory footprint of the incremental GC is clearly higher than for the other GCs. While the difference to the copying GC is only 9% on average, the memory allocation is 57% higher than the generational and compacting GC. This is due to the fact that the incremental and copying GC share the property of copying alive objects to free space during compaction. The incremental GC requires additional storage for the forwarding pointer in each object header and allocates Wasm memory in partition granularities. However, when memory space is becoming short, the incremental GC limits the required copy space, cf. Section 4.4. Moreover, incremental GC avoids evacuation of low-garbage partitions. In our incremental GC, we tuned the garbage threshold for the partition evacuation to 15% as this minimizes the Wasm allocated memory size.

5.2 Scalability Measurements

An important property of our new GC is to offer scalable memory usage, as the GC is able to limit the amount of its work per blockchain transaction, such that they can succeed.

To verify the scalability, we created a different benchmark set, with a subset of the programs of the previous benchmark that can scale in memory size. More specifically, we concentrated on the data structure scenarios, by issuing continuous insertions until hitting a limit, i.e. running out of memory or instruction limit of the blockchain transaction. The insertions do not only add live objects but also create garbage arising from temporary computational objects or dynamic data structure reorganization on growth.

Table 4 lists the programs in the GC scalability benchmark.

5.2.1 Allocation Limit. In a first measurement, we determine the maximum number of values that can be inserted into the different data structures. While the 'blob' program inserts 64 KB values into an array list, the other cases populate very small 32-bit numbers to the data structures. The average results are summarized in the second column of Table 5 with the detailed results shown on a logarithmic scale in Figure 6. We also include the artificial runtime configuration with disabled garbage collection ("No GC").

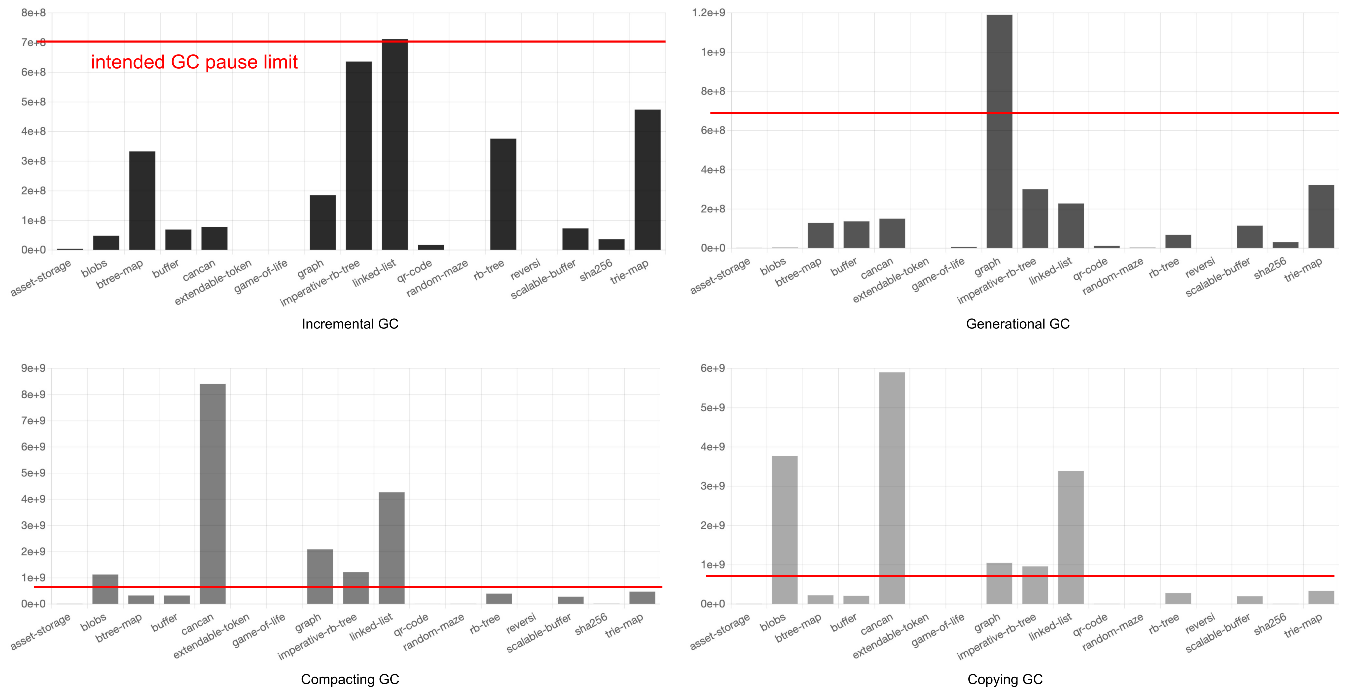

Figure 4. Maximum GC pause

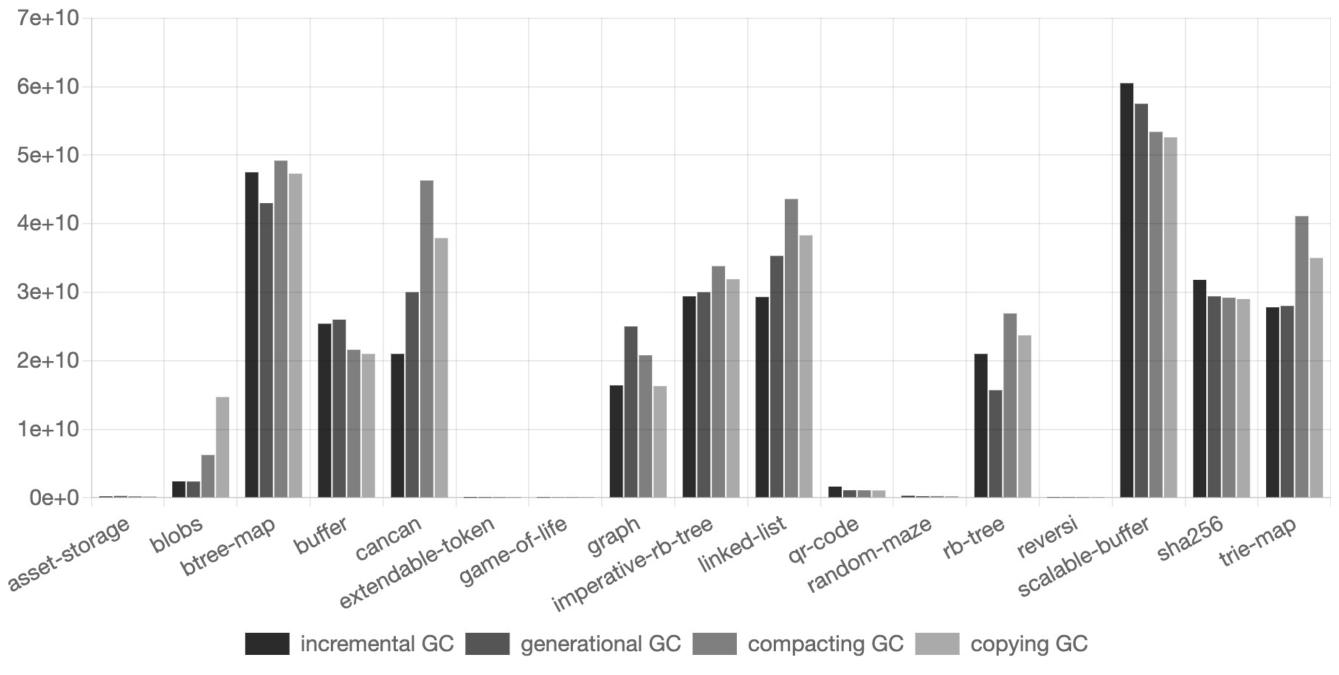

Figure 5. Total Wasm instructions executed per benchmark case

As intended by the GC design, the new incremental GC scales substantially higher, by 2.5x on average, when compared to the other GCs or having no collection at all. The other GCs all hit the instruction limit of the blockchain transaction, when trying to scale in memory size, except for the 'blob' case. Due to the two-space design, the copying GC can only scale up to 2 GB live data and therefore runs out of memory during the 'blob' case. Of course, when disabling the collector, garbage cannot be recycled, so memory is exhausted sooner than with the incremental GC.

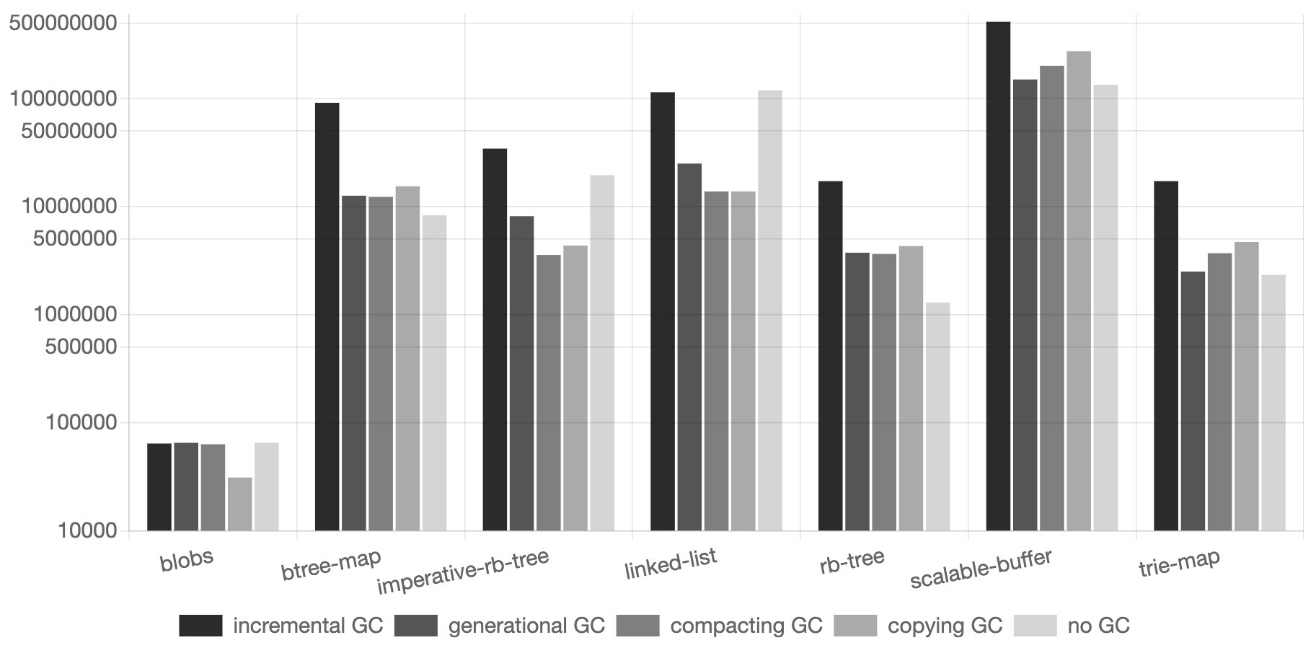

Figure 6. Maximum insertions per data structure, logarithmic scale

Table 4. Scalability benchmark overview

Benchmark	Description
blobs	Inserting 64 KB blobs in a list
btree-map	Inserting numbers in a b-tree
imperative-rb-tree	Inserting numbers in a mutable red-black tree
linked-list	Inserting numbers in a linear linked list
rb-tree	Inserting numbers in a functional implementation of a red-black tree
scalable-buffer	Inserting numbers in a multi-level array list
trie-map	Inserting numbers in a functional implementation of a trie data structure

Table 5. GC scalability comparison

GC	Allocation limit (million insertions)	Maximum usable heap size
Incremental GC	149	3.8 GB
Generational GC	33	0.90 GB
Compacting GC	37	0.87 GB
Copying GC	47	0.72 GB
No GC	47	4.0 GB

5.2.2 Maximum Heap Size. We also like to show that the new GC, unlike the other classical GCs, enables a program to use the available memory capacity. For this purpose, we remeasure the final heap size on the scalability benchmark, i.e. the program size when the benchmark case hits the limit (memory limit or instruction limits). The results are summarized in the last column of Table 5. As expected, the new incremental GC allows the heap to scale up close to the maximum available 4 GB in 32-bit space, with some reserve needed for the GC runtime structures, the mark stack and mark bitmaps.

6 Related Work

The Shenandoah GC [11] for OpenJDK served as an inspiration for our GC design. It is also a partitioned evacuation-compacting GC that is based on full-heap incremental SATB marking. We deliberately choose a different design in the following regards: Our GC guarantees a strict limit on the GC increments for a given rate of mutator allocations. There are no stop-the-world situations like in Shenandoah where the GC blocks for an unbounded amount of time when initiating or finalizing the marking phase, or when defragmenting large objects that span multiple partitions. For faster memory reclamation, we do not wait for the subsequent GC run when freeing partitions, but perform an additional incremental updating phase. More memory-economic configurations are preferred in our GC, such that partitions are already evacuated if they contain at least 15% garbage instead of 60% like in Shenandoah. Moreover, our GC targets Wasm for a blockchain runtime, while Shenandoah runs in the JVM and is optimized for classical computer architectures. In 2019, the Shenandoah GC implementation replaced the Brooks pointers by load barriers [19] with a conditional check. For our GC, we measured superior runtime performance of 27.5% with the Brooks pointer compared to a conditional load barrier.

ZGC [22] is another new high-scalable, mostly concurrent garbage collector in OpenJDK that also works with partitioned evacuation-compaction. It applies efficient hardware-supported load barriers by storing color bits in the pointers and mapping the heap to multiple virtual memory addresses, a different memory view for each color. Such hardware support is unfortunately not available in our blockchain virtual machine (where we also still work on 32-bit address spaces). Similar to Shenandoah, ZGC also contains stop-the-world intervals to transition between GC phases, at the beginning and the end of the marking phase, as well as after the selection of the evacuation candidates.

The current default GC of Java is still G1 [10] (also called "garbage first") which also employs evacuation-compaction by prioritizing high-garbage partitions like in our GC, Shenandoah, and ZGC. However, G1 does not implement forwarding pointers or any other technique for incremental evacuation. Instead, G1 performs stop-the-world evacuation and pointer updates. To reduce the pauses for pointer updates, G1 maintains a remembered set per partition.

The Mature Object Space (MOS) [17], also known as the train algorithm, is a real-time GC that is also based on partitioned evacuation compaction, however, with a specific evacuation scheme, that allows partitioned heap marking before reclaiming memory. The greatest advantage of MOS in comparison to our GC, Shenandoah, and ZGC, is that MOS does not need to mark the full heap and is still capable of reclaiming inter-partition garbage. Even if the marking phase is incremental, full heap marking will hit a scalability limit at some point for very large heaps, because the GC latency is not limited. At the same time, MOS imposes a specific order on the evacuations of partitions: Unlike G1, Shenandoah, ZGC, and our GC, MOS cannot optimize for selective evacuation of high-garbage partitions.

Another approach in real-time garbage collection [5, 8] is to avoid copying in most cases, using segregated free lists, and only defragmenting when memory is scarce. While an ideal real-time GC can guarantee even mutator utilization, the overheads are significantly higher than in our case, e.g. mutator utilization under 50% and space overheads of more than 2x [5].

The recent Wasm GC proposal [15] aims to offer automatic memory management directly by the Wasm engine, considering that several of the engines run in a browser which already features a powerful GC for JavaScript, such as V8 GC [9, 20], SpiderMonkey GC [13], and JavaScriptCore GC [21]. We cannot use Wasm GC for the following reasons: (1) Our blockchain uses the Wasmtime engine [1] which does not yet implement Wasm GC. (2) We require a deterministic and strictly incremental GC and currently, there exists no Wasm engine meeting these requirements. SpiderMonkey for example only offers a partly incremental GC which blocks the mutator for unbounded time in certain GC phases, e.g. during compaction. (3) Our blockchain would need to snapshot the host GC which could be expensive (if supported at all). (4) In addition to determinism and incrementality, the GC should ideally also compact the heap. This is for example not the case for JavaScriptCore.

AssemblyScript [4] implements an incremental mark-and-sweep garbage collector that runs in Wasm. While compaction is not addressed by this GC, it also contains an unbounded pause when scanning the root set, in particular the call stack. Moreover, AssemblyScript adds a space overhead of two words per object, while our GC only adds one (the forwarding pointer).

XLR [23] achieves high throughput and low latency by combining reference counting with tracing collection, and shifting from concurrent collection to incremental collection with short pauses. To improve the poor performance of reference counting, pointer updates are collected and aggregated over time to update the reference counters. Differently to our GC, XLR does not update pointers incrementally but pauses the mutator for evacuating a partition and updating the corresponding relevant incoming pointers by using remembered sets.

7 Conclusion

Garbage collection for Wasm-based blockchains is a relatively new area that poses different requirements compared to traditional computing. We have identified both fragmentation and incrementality as key aspects a GC should address in this context. As current mainstream GCs have a different focus, we have designed a new GC that is tailored to the Wasm-based blockchain. We implemented this GC for the Motoko programming language and evaluated it on the Internet Computer blockchain. Compared to classical GCs, our customized GC performs better in terms of memory scalability and runtime costs.

Availability

The presented GC is open source and available at: https://github.com/dfinity/motoko/pull/3837. The same applies to the Motoko programming language and the Internet Computer: https://internetcomputer.org. The GC benchmark suite can be found at: https://github.com/luc-blaeser/gcbench.

Acknowledgment

Matthew Hammer and Yan Chen provided valuable suggestions for improving this paper. We also thank the entire Motoko team for their support.

References

[1] Bytecode Alliance. 2023. *Wasmtime.* https://wasmtime.dev/

[2] Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, and Alexandra Zapuc. 2023. Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain. In *Proceedings of the 2023 USENIX Annual Technical Conference* (Boston, MA, USA) *(ATC '23)*.

[3] Malcolm P Atkinson, Peter J Bailey, Ken J Chisholm, W Paul Cockshott, and Ron Morrison. 1983. PS-Algol: A language for persistent

programming. In *Proc. 10th Australian National Computer Conference, Melbourne, Australia*. 70–79.

[4] The AssemblyScript Authors. 2023. *The AssemblyScript Project*. https://github.com/AssemblyScript

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 285–298.

[6] Rodney A. Brooks. 1984. 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* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 256–262.

[7] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2022. Internet Computer Consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (Salerno, Italy) *(PODC'22)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/3519270.3538430

[8] Perry Cheng and Guy E. Blelloch. 2001. A Parallel, Real-Time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 125–136.

[9] Ulan Degenbaev, Michael Lippautz, and Hannes Payer. 2019. Garbage Collection as a Joint Venture. *Commun. ACM* 62, 6 (may 2019), 36–41.

[10] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*.

[11] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages.

[12] DFINITY Foundation. 2023. *The Motoko Programming Language*. https://github.com/dfinity/motoko

[13] Mozilla Foundation. 2023. *SpiderMonkey Garbage Collector*. https://firefox-source-docs.mozilla.org/js/gc.html

[14] WebAssembly Community Group. 2022. *WebAssembly Specification, Version 2.0*. https://webassembly.org/

[15] WebAssembly Community Group. 2023. *GC Proposal for WebAssembly*. https://github.com/WebAssembly/gc

[16] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[17] Richard Hudson and J. Moss Eliot. 1992. Incremental Collection of Mature Objects. (07 1992).

[18] Richard Jones and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., USA.

[19] Roman Kennke. 2019. *Shenandoah GC in JDK 13, Part 2: Eliminating the Forward Pointer Word*. https://developers.redhat.com/blog/2019/06/28/shenandoah-gc-in-jdk-13-part-2-eliminating-the-forward-pointer-word?p=606477

[20] Georgiy Krylov, Maria Patrou, Gerhard W. Dueck, and Joran Siu. 2020. The Evolution of Garbage Collection in V8: Google's JavaScript Engine. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. 1–6.

[21] Haoran Xu. 2022. *Understanding Garbage Collection in JavaScriptCore from Scratch*. https://webkit.org/blog/12967/understanding-gc-in-jsc-from-scratch

[22] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (sep 2022), 34 pages.

[23] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-Latency, High-Throughput Garbage Collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 76–91.

Received 2023-07-23; accepted 2023-08-28