Technical Disclosure Commons

Defensive Publications Series

March 12, 2019

Unified concurrent write barrier

Michael Lippautz

Ulan Degenbaev

Hannes Payer

Unified concurrent write barrier

ABSTRACT

In a programming language with support for garbage collection, a write barrier is a code snippet that maintains the key invariants of the garbage collector. The write barrier is typically executed after a write operation. The write barrier is computationally expensive and can impact program performance. This is true to a greater extent for languages where garbage collectors need to maintain multiple sets of invariants. For example, languages that employ garbage collection schemes with two collectors may maintain their invariants using multiple different write barriers. The techniques of this disclosure address the problem of maintaining multiple invariants by unifying the write barriers and by executing computationally expensive parts of the write barrier in a concurrent thread.

KEYWORDS

- write barrier
- garbage collection
- program invariants
- generational garbage collection
- concurrent write barrier
- unified write barrier

BACKGROUND

Write barriers are a well-known and well-studied mechanism for garbage collection [1, 2, 3]. *Marking write barriers* are defined based on how such barriers preserve the strong or weak tricolor invariants:

- Weak tricolor invariant: All white objects pointed to by black objects are grey protected, meaning that those objects are also reachable from grey or white objects, either directly or transitively.
- Strong tricolor invariant: There are no pointers from black to white objects.

The barriers are further classified into one or more of the following categories:

- Add to wavefront: Barriers that shade an object grey if it was white before.
- Advance wavefront: Barriers that scan an object to make it black.
- Retreating wavefront: Barriers that revert an object from black to grey.

The above set of existing marking write barriers is complete [2, 3], e.g., all possible variations of write barriers that maintain consistent state with the tricolor invariants are covered by the above write barriers. Example barriers are: Steele, Boehm, Dijkstra for grey mutator techniques; Baker, Appel, Abraham/Yuasa for black mutator techniques; etc.

There exist *hybrid* marking barriers in multiple variants. For example:

- Barriers that are combined, e.g., the Go programming language combines a Dijkstra-style write barrier [4] with a Yuasa deletion barrier. The resulting barrier executes two barriers sequentially.
- Hybrid barriers that statically compile [1] the right version of barrier after performing the write.

For generational garbage collection the heap is partitioned into generations. Inter-generational references are used to collect garbage in a single generation. Such references are generally treated as roots when collecting a single partition. A structure is defined that holds the information of the remembered inter-generational references set. The standard approach is to populate the remembered set using generational write barriers. When writing such an inter-generational reference, a barrier checks source and target, and emits an entry in a remembered set. The granularity of records in remembered sets varies and can range from being exact (address x points to object y) to being coarse-grained (some address on page x points into partition y) [3].

Another type of barrier is the compaction barrier. The compaction barrier maintains a set of object field addresses such that the field points to an object that resides in a page chosen for compaction [1].

Since write barriers can incur some cost, the barriers can be delayed by only recording necessary information into a fast intermediate data structure, often known as sequential store buffer (SSB) [5, 3]. SSBs can be used per thread if necessary [3].

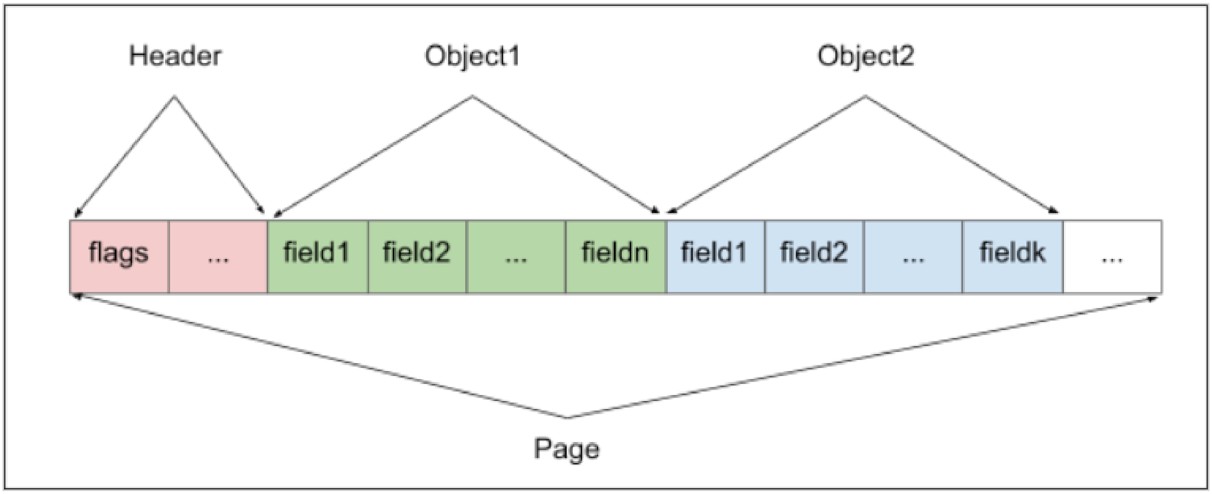

Fig. 1: Objects in a page

Fig. 1 illustrates objects in a page. Objects within programs comprise multiple words. The words in an object are known as the object fields. Objects are allocated in memory blocks, called pages. The address of a page is aligned on the page size granularity, which is a power of two (2^*N*). This enables an object to find its page quickly by masking out the lower *N* bits of the object address. This operation is referred to as `PageFromAddress(object)`. The header of a page contains a page flags word which is used to encode information relating to the write barrier.

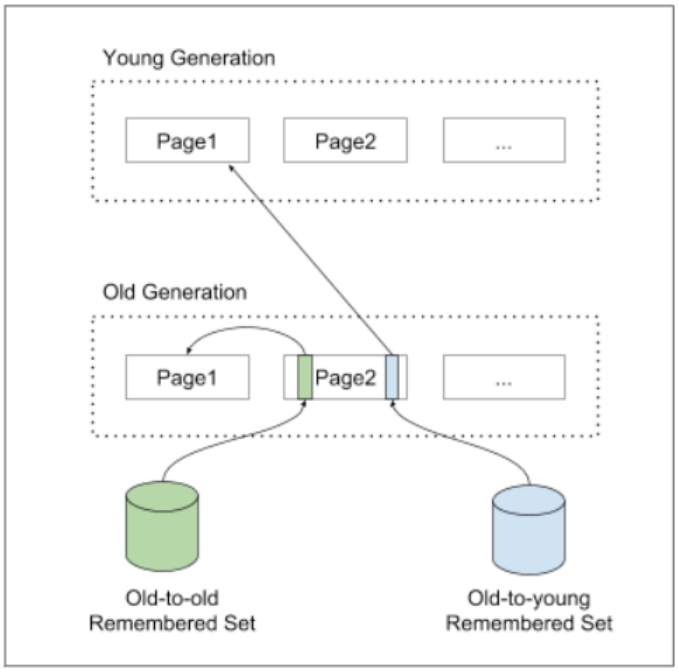

Fig. 2: Generations and remembered sets

The heap comprises of two generations: young and old. Each generation has a collection of pages that belong to that generation. The young generation garbage collector uses the generational barrier. The generational barrier maintains a set of object field addresses such that the field is in the old generation and it points to an object in the young generation. The set is called a *remembered set*, as illustrated in Fig. 2.

The mark-sweep-compact garbage collector uses the compaction barrier and the marking barrier. The compaction barrier maintains a set of object field addresses such that the field is in the old generation and it points to an object in the old generation that resides in a page chosen for compaction.

The marking barrier is a Dijkstra-style write barrier that maintains the strong tri-color invariant. The mark-sweep-compact garbage collector associates with each object two marking bits. The bits encode three colors: white, grey, and black. The strong tri-color invariant states that black objects do not point to white objects. The operation that computes the address of the two marking bits associated with the given object is denoted as `MarkbitsFromAddress(object)`.

DESCRIPTION

Per the techniques of this disclosure, a write operation in a program is defined as an operation that stores a reference to a target object in a field of a source object:

```
    source->field = target
```

The techniques apply to a language that divides the heap into two generations: young and old, and uses two garbage collectors: a copying garbage collector for the young generation and a concurrent mark-sweep-compact garbage collector for the whole heap.

The write operation may break the invariants that the garbage collectors rely on. In order to maintain the invariants, the program runs a write barrier after each write operation. The write barrier performs the following actions.

- Maintain the old-to-young remembered set that tracks all references from the objects in the old generation to the objects in the young generation. This part of the write barrier is the generational barrier.
- Maintain the old-to-old remembered set that tracks references from the objects in the old generation to the objects located on the evacuation candidate pages (also in the old generation). This part of the write barrier is the compaction barrier.
- Maintain the tri-color invariant, such that no black object points to a white object (marking barrier).

As mentioned before, the write barrier is executed or emitted after each write operation. Therefore, for overall performance, it is important to optimize the number of instructions in the write barrier. The techniques described herein optimize the number of instructions in the write barrier, thereby reducing overhead and improving performance of the write barrier. The computationally expensive parts of the write barrier are offloaded to a concurrent thread and the number of instructions in the part of the write barrier inlined into the program are further reduced by unified processing of its three parts: generational, compaction, and marking.

To maintain invariants corresponding to the three parts, the techniques perform three corresponding checks and operations:

```
// Write operation:
source->field = target;
// Write barrier:
if (CheckGenerationalBarrier(source, target)) {
  ExecuteGenerationalBarrier(&source->field);
}
if (CheckCompactionBarrier(source, target)) {
  ExecuteCompactionBarrier(&source->field);
}
if (CheckMarkingBarrier(target)) {
  ExecuteMarkingBarrier(target);
}
```

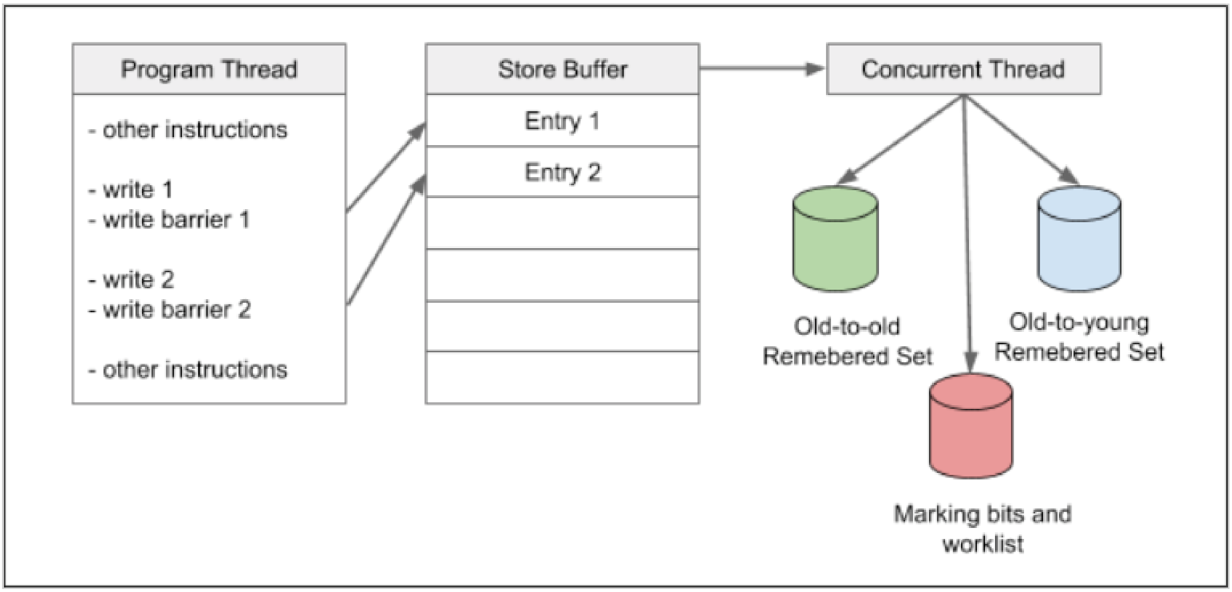

Fig. 3:. Concurrent execution of write barriers

The `Check` parts are unified, and the `Execute` parts are moved to a concurrent thread. As shown in Fig. 3, this is done by a special encoding of the page flags and by collecting the arguments of barriers in an intermediate buffer, instead of executing the `Execute` parts directly. The intermediate buffer is referred to as a store buffer. The store buffer is an array of word-sized entries. Each entry is either the address of a field (for the generational and compaction barriers) or the address of an object (for the marking barrier). To distinguish the three cases, the two alignment bits of an address that store a tag are used:

- Bits pattern `00` indicates that the entry is the address of an object (marking barrier).
- Bits pattern `01` indicates that the entry is the address of a field that contains a reference to an evacuation candidate (compaction barrier).
- Bits pattern `10` indicates that the entry is the address of a field that contains a reference to a young object (generational barrier).

Note that object fields and object start are aligned by at least four bytes such that there are at least two alignment bits. The page flags are chosen such that the same code for compaction and generational barriers can be used:

- Bit #0 indicates that the page is an evacuation candidate.
- Bit #1 indicates that the page is a young generation page.
- Bit #2 indicates that the page is an old generation page.
- Bit #3 indicates that marking is in progress.

Note that bits #0 and #1 are mutually exclusive such that a page is either an evacuation candidate or in the young generation, but not both. Bit #2 is a negation of bit #1. Also, bit #0 implies bit #2 such that a page can be an evacuation candidate only if marking is in progress.

By masking specific page flag bits, specific conditions can be checked for. The following masks are used (literals are encoded in binary base):

- `kRememberedSet = 0b0011`. This mask selects bits #0 and #1.
- `kPointersToHereAreInteresting = 0b1010`. This mask selects bits #1 and #3.
- `kPointersFromHereAreInteresting = 0b1100`. This mask selects bits #2 and #3.
- `kIsMarking = 0b0100`. This mask selects bit #2.

Instead of executing the barriers, quick checks on the page flag bits are performed using the above masks and entries are added to the store buffer. The store buffer has a fixed size and eventually becomes full. At that point, ownership of the store buffer is transferred to a concurrent thread for processing and to set up a new store buffer. The new store buffer is taken from the predefined pool of store buffers. If the pool is empty, then programs or threads dependent on the store buffer wait for the concurrent thread to process the store buffers.

The currently active store buffer is represented by three variables: `start` (address of the first entry in the store buffer), `top` (address of the first free entry), and `limit` (address of the end of the store buffer, e.g., the address of the entry after the last entry in the store buffer).

Pseudocode for the unified concurrent write barrier is given below.

```
// Write operation:
source->field = target;
// Write barrier:
source_flags = PageFromAddress(source)->flags;
if ((source_flags & kPointersFromHereAreInteresting) && IsHeapObject(target)) {
  target_flags = PageFromAddress(target)->flags;
  if (target_flags & kPointersToHereAreInteresting) {
    // Generational and compaction barriers.
    if (!(source_flags & kRememberedSet) && (target_flags & kRememberedSet)) {
      *top++ = &(source->field) | (target_flags & kRememberedSet);
    }
    // Marking barrier.
    if (target_flags & kIsMarking) {
      markbits = MarkbitsFromAddress(target);
      if (load(markbits) == kWhite) {
        if (compare_and_swap(markbits, kWhite, kGrey)) {
          *top++ = (target & ~0b11);
        }
      }
    }
    // Check if store buffer is full.
    if (top + 1 >= limit) {
      TransferStoreBufferToConcurrentThread();
    }
  }
}
```

A line-wise description of the pseudo-code is as follows:

1. Perform a check of whether any barrier is needed. The check is covered by the `kPointersFromHereAreInteresting`, `kPointersToHereAreInteresting` masks. Additionally a check is performed to determine if the target is a heap object or an unboxed integer.
2. If the check succeeds, then
   a. Add a store buffer entry for the generational or compaction barrier if the page flags indicate that this is needed.
   b. Add a store buffer entry for the marking barrier if marking is in progress and the target object is white. This also transitions the target object from white to grey.
   c. Check if the store buffer is full (e.g., insufficient space for two entries). In that case, the `TransferStoreBufferToConcurrentThread` function is invoked, which transfers the current store buffer to a concurrent thread and sets up a new empty store buffer. This ensures that the invariant has space for at least two entries in the store buffer.

Upon receiving a store buffer, the concurrent thread iterates over it, decodes each entry based on the tag bits, and executes the corresponding barrier:

```
void DrainStoreBufferConcurrently(start, limit) {
  for (Address* entry = start; entry != limit; entry++) {
    Address addr = (*entry) & ~0b11;
    int tag = (*entry) & 0b11;
    if (tag == 0b00) {
      ExecuteMarkingBarrier(addr | 0xb01);
    } else if (tag == 0b01) {
      ExecuteCompactionBarrier(addr);
    } else if (tag == 0b10) {
      ExecuteGenerationalBarrier(addr);
    }
  }
}
```

Once the store buffer process is complete, the concurrent thread returns the store buffer to the store buffer pool.

As mentioned before, an advantage of the described techniques is that the relatively slow parts of the write barrier are executed on a concurrent thread, thereby improving throughput and responsiveness of the program. To achieve this, the techniques can incur a slight increase in memory usage (e.g., for store buffers) and thread synchronization overhead.

Some alternative variations on the techniques are as follows.

- If the store buffer pool is empty, then the program thread may drain the current store buffer itself instead of transferring it to the concurrent thread. This ensures that the program thread can make progress independent from the concurrent thread.
- Instead of using a predefined store buffer pool, allocate and release store buffers.
- Use multiple concurrent threads for draining store buffers.
- Use lock-based or lock-free techniques for transferring a store buffer to the concurrent thread and to the store buffer pool.

The techniques are applicable in situations with generational/concurrent/incremental garbage collection, e.g., language runtimes, virtual machines, etc.

CONCLUSION

In a programming language with support for garbage collection, a write barrier is a code snippet that maintains the key invariants of the garbage collector. The write barrier is typically executed after a write operation. The write barrier is computationally expensive and can impact program performance. This is true to a greater extent for languages where garbage collectors need to maintain multiple sets of invariants. For example, languages that employ garbage collection schemes with two collectors may maintain their invariants using multiple different write barriers. The techniques of this disclosure address the problem of maintaining multiple invariants by unifying the write barriers and by executing computationally expensive parts of the write barrier in a concurrent thread.

REFERENCES

[1] Blackburn, Stephen M., and Antony L. Hosking. "Barriers: Friend or foe?." In *Proceedings of the 4th international symposium on memory management*, pp. 143-151. ACM, 2004. http://dx.doi.org/10.1145/1029873.1029891

[2] Pirinen, Pekka P. "Barrier techniques for incremental tracing." In *Proceedings of the 1st international symposium on memory management*, pp. 20-25. ACM, 1998. http://dx.doi.org/10.1145/286860.286863

[3] Jones, Hosking, and Moss. *The garbage collection handbook: the art of automatic memory management* (1st ed.). Chapman & Hall/CRC, 2011.

[4] Dijkstra, Edsger W., Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth FM Steffens. "On-the-fly garbage collection: an exercise in cooperation." *Communications of the ACM* 21, no. 11 (1978): 966-975. http://dx.doi.org/10.1145/359642.359655

[5] Blackburn, Stephen M., Perry Cheng, and Kathryn S. McKinley. "Oil and water? High performance garbage collection in Java with MMTk." In *Proceedings. 26th International Conference on Software Engineering*, pp. 137-146. IEEE, 2004.