Barrier techniques for incremental tracing

Pekka P. Pirinen
Harlequin Limited *

May 15, 1998

Abstract

This paper presents a classification of barrier techniques for interleaving tracing with mutator operation during an incremental garbage collection. The two useful tricolour invariants are derived from more elementary considerations of graph traversal. Barrier techniques for maintaining these invariants are classified according to the action taken at the barrier (such as scanning an object or changing its colour), and it is shown that the algorithms described in the literature cover all the possibilities except one. Unfortunately, the new technique is impractical. Ways of combining barrier techniques are also discussed.

1 INTRODUCTION

Many garbage collection (GC) algorithms need to traverse, or *trace*, the graph of reachable objects. This paper surveys barrier techniques used to interleave tracing with the user program (incremental GC, not fully concurrent), from the point of view of the uses that the barriers are put into, rather than implementation techniques for raising barriers. Also, this paper does not concern itself with the various ways of using barriers to ensure the mutator's consistency in the face of changes made by the collector.

Chapter 2 establishes the concepts that are used to describe incremental GC algorithms; it might seem to be describing the obvious, but the argument does require precise definitions. Section 2.1 describes tracing in terms of tricolour marking and introduces other key concepts such as *the root* and *the condemned set*. Section 2.2 discusses the division into the mutator and the collector. Section 2.3 derives two invariants, that encapsulate the requirements on incremental tracing algorithms; these invariants are not new, but the analysis leads to an extension of Wilson's [Wil94] distinction of snapshot-at-the-beginning vs. incremental-update[1], which turns out to be very useful in the classification.

* Barrington Hall, Barrington, Cambridge CB2 5RG, UK; pekka@harlequin.co.uk

[1] although I prefer Jones' orthography [Jon96] for these concepts

Chapter 3 is the heart of the paper: after laying some background, it presents, in section 3.2, a classification of the barrier techniques for maintaining the invariants in the face of changes made by the mutator. This works by considering the different actions that an incremental GC algorithm might take when a barrier is hit. We find that there are only a handful of different barrier techniques; each of these is described and its properties examined. A key insight here is to take the colour of the mutator as the main classifier. The chapter concludes with a discussion on combining different barrier techniques in the same algorithm.

The final chapter explores the limitations of the analysis and the directions for future work in this area.

2 CONCEPTS

2.1 TRACING

Many garbage collection algorithms work by determining the set of objects reachable from the program's state (as an approximation of the set of live objects) and then recycling the unreachable nodes. To do this, they need to traverse the graph of pointers between program objects; this is called *tracing*.

2.1.1 ROOTS AND THE CONDEMNED SET

The starting point of the traversal or *the root* is the program's state, or the mutator's state to be precise, since the collector's state is not relevant to liveness (but this is no deep insight, it's only because we choose to divide the program into mutator and collector just so, see section 2.2).

GC algorithms often restrict their attention to some set of objects where they expect to find lots of unreachable objects; this we call *the condemned set*. They assume that everything else is reachable, and determine what part of the condemned set is reachable under that assumption. We try to arrange matters so that this is a good approximation of reachability from the root, but the techniques used are outside the scope of this paper.

Many collectors allow the client to register arbitrary "roots", such as static data segments, which will be part of the root of any collection thereafter. This can be regarded as an optimization, so that the collector doesn't have to find static data by scanning the program code or some other tedious method.

Note that everything outside the condemned set could be said to be part of the root, since the GC is not concerned with the reachability or liveness of these objects. However, this gets confusing, so we won't use that terminology.

2.1.2 TRICOLOUR MARKING

We will discuss tracing in terms of *tricolour marking* [DLM$^+$76]. All objects are partitioned into three sets:

Black objects have been noted reachable and the collector has finished with them and need not visit them again (for the purposes of tracing).

Grey objects have been noted reachable, but must still be visited by the collector in order to process their children.

White objects have been condemned, but not visited.

At the beginning, the condemned set is made white and everything else grey (although, if our algorithm is any good, we can easily deduce that much of it cannot refer to the white set, and can thus be immediately turned black). Tracing terminates when there are no grey objects left and the mutator is black (see section 2.2). At this point all reachable objects are black and all remaining white objects are unreachable (but the converses don't necessarily hold after incremental collections).

Note that the definitions imply that an object can never revert to white during a collection, once it has been coloured grey or black. This is important for most barrier algorithms.

The partition describes the order of traversal: the collector must always choose one of the grey objects to process next.

2.2 THE MUTATOR AND THE COLLECTOR

In order to analyse incremental GC, we notionally divide the program (following [DLM+76]) into two semi-independent parts: the mutator and the collector.

The mutator executes the user code, which allocates objects and "mutates" the objects. While mutating, it implicitly frees storage by modifying existing objects so that some objects become unreachable.

The collector executes the GC code, which discovers unreachable storage and reclaims it.

By analogy with object colour, we also say that the mutator is black if its state has been processed by the collector and does not need to be examined again (e.g., in Baker's GC algorithm [Bak78]), and that the mutator is grey, if its state still needs to processed by the collector (such as in Steele's algorithm [Ste75]). This analogy allows us to reason about the mutator state as if it were an object.

When we discuss mutator colour, we should take the mutator state to include all the program state that the mutator can act on *directly*. More precisely, anything that the mutator can access without regulation by the collector. In practice, this usually means it contains the registers of any processors that the mutator is running on, the stacks of mutator threads, and global static data. If a barrier can be placed between some object and the mutator, then we don't need to consider it part of the mutator state. This includes protectable static data and stacks protected by stack barriers. Such things might still be roots in the sense that they are always in the initial grey set.

2.3 INVARIANTS

The basic requirement on all tracing algorithms is to discover all the reachable objects. To achieve this, we only need to ensure that the collector's view of reachability stays correct, or at least over-conservative, so that it doesn't reclaim live objects. The mutator modifying grey or white objects is not a problem, since the collector will visit them some time in the future (if they're still reachable—if not, they're garbage, and didn't need to be visited). Modifying black objects by adding or deleting pointers to black or grey objects is not a problem, because it only involves objects that the collector has already decided are reachable. Adding white pointers to black objects might lead to a problem, if the collector doesn't ever see any other pointers to the white object. This would mean that the white object is reachable, but it will never be seen by the collector, because the collector doesn't revisit black objects. Removing white pointers from black objects doesn't add to the problem, since the collector would not have seen those pointers, anyway.

We conclude that to guarantee that the collector will not miss any reachable objects, we must ensure that all white objects pointed to by a black object are found by the collector, i.e., they are reachable from some grey object through a chain of white objects.

The weak tricolour invariant: All white objects pointed to by a black object are reachable from some grey object through a chain of white objects.

A copying GC algorithm will copy condemned objects as it first encounters them (when they become grey). To correctly update all the pointers to the moved objects, we need to make sure that the collector sees all such pointers (that will survive the collection), even when the mutator moves the pointers around. The mutator putting such pointers in grey or white objects is not a problem, since the collector will visit them some time in the future (if they're still reachable). Putting black or grey pointers in black objects is not a problem, because those pointers have already been updated. Putting white pointers into black objects cannot be allowed, because the collector doesn't revisit black objects, and hence will not have a chance to update those pointers.

We conclude that to guarantee that the collector does not miss any pointers between reachable objects, we must ensure there are no pointers from a black object to a white object.

The strong tricolour invariant: There are no pointers from a black object to a white object.

Obviously, the strong invariant implies the weak one.

If we're only maintaining the weak invariant, then instead of scanning something to turn it black, a grey copy of it can be made. Yuasa [Yua90] used this technique for the mutator. Under the strong invariant, this won't do.

In his analysis of write-barrier algorithms, Wilson [Wil94] notes that in order for the collector to miss a reachable object the following two conditions need to hold at some point during tracing:

- The mutator stores a pointer to a white object into a black object.

- All paths from any grey objects to that white object are destroyed.

He then divides write-barrier algorithms into two classes: incremental-update algorithms ensure that the first condition cannot happen; snapshot-at-the-beginning algorithms ensure the second condition cannot happen. snapshot-at-the-beginning algorithms are so called, because they work by ensuring the collector will process pointers that the mutator would overwrite (but note that this does not mean all modifications need to be seen by the collector); incremental-update algorithms, on the other hand, keep the collector updated on any changes to black objects.

Clearly, incremental-update algorithms preserve the strong invariant. We can also see that snapshot-at-the-beginning algorithms aim at maintaining the weak invariant only. We can now see that this distinction can be used to analyse read-barrier techniques as well as write-barrier ones, and even mixed read- and write-barriers (see technique 7 below); for example, Wilson and Johnstone's analysis of the advantages of incremental-update vs. snapshot-at-the-beginning algorithms in section 3 of [WJ93] applies more widely to a comparison of algorithms maintaining the strong invariant vs. those that don't (note in particular that local optimizations are much easier in incremental-update algorithms).

3 BARRIERS

In incremental GC, barriers are used to intercept the mutator's accesses to the objects, and give the collector a chance to perform some action, before the access completes. *Read-barriers* intercept mutator loads and *write-barriers* mutator stores. These can be used to maintain invariants, as described below.

3.1 Scaleability

Hardware barriers on general-purpose platforms will have to work at a large granularity. Typically, pages containing many objects can be read- or write-protected, and the resulting exception handled by the collector. The cost of handling such exceptions is typically so large that we want to avoid repeats. So we're particularly interested in techniques that can be efficiently implemented at large granularity. We'll call them *scaleable*.

Locality suggests that repeated accesses to the same page are likely, so on the first access, we'll want process every object on the page to remove the condition that caused the barrier to be erected. This means that even if the exact address being accessed and the data being transferred could be determined (and on some platforms it's quite hard), we are not interested in them because they apply to one tiny part of the page only. Furthermore, all objects on the page should ideally be in the same condition, so that they can all be processed now, and the possibility of having to reprotect the page later is avoided. This is hard to achieve in non-copying collection.

Non-scaleable techniques are still interesting, as they're suitable for implementation in software.

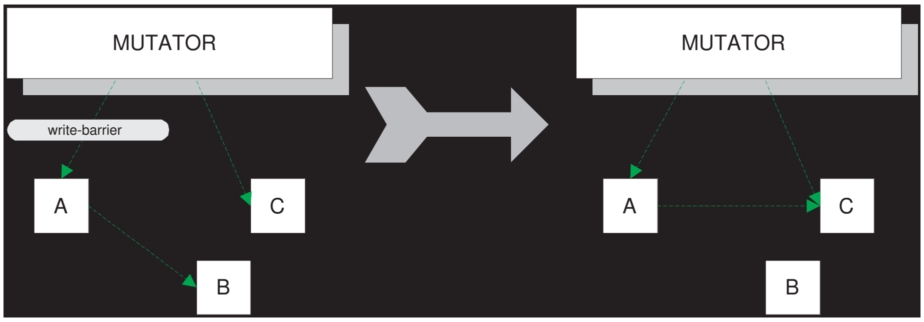

Figure 1: Write-barrier nomenclature

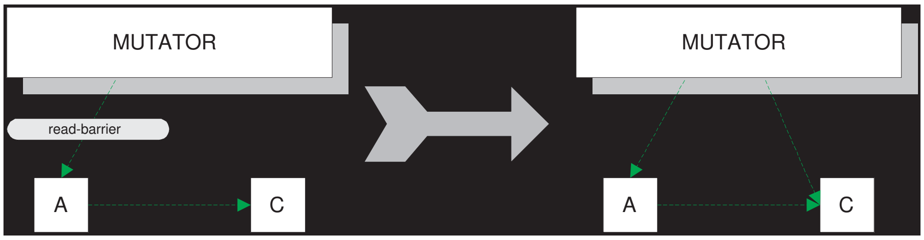

Figure 2: Read-barrier nomenclature

3.2 Invariant barrier techniques

In the following sections, we categorize the barrier techniques for maintaining the tricolour invariants on the basis of the action taken when a barrier is hit. Here's a list of the actions used in the algorithms:

- Scan (or copy, see section 2.3) one of the objects involved, turning it black.
- Turn an object grey, if it was white; we call this *shading*.
- Turn a black object back to grey.

That's all the useful simple actions, because turning an object white would break the invariants, as would turning an object black without scanning it. There might be some more complicated actions that could be used to maintain invariants, but this set covers the current state of the art.

In the write-barrier descriptions, there's a write-barrier on object A and the mutator attempts to change it from referring to object B to refer to object C, see figure 1.

In the read-barrier descriptions, there's a read-barrier on object A and the mutator attempts to read a pointer to object C from object A, see figure 2.

3.2.1 Invariant barriers for a grey mutator

Remember that a grey mutator might well contain pointers to white objects.

Technique 1 (Dijkstra et al. [DLM$^+$76]) *A write-barrier on black: when hit, shade the new referent, C.*

This maintains the strong invariant, so it's an incremental-update technique. It is not scaleable, because looking at all the referents (corresponding to object C in the diagram) of all objects on the page would be expensive.

Technique 2 (Steele [Ste75]) *A write-barrier on black: when hit, turn the object under the barrier, A, grey only if the new referent, C, is white.*

This is also an incremental-update technique. Like other techniques involving the referents, it is not scaleable.

Technique 3 (Boehm et al. [BDS91]) *A write-barrier on black: when hit, turn the object under the barrier, A, grey.*

This technique can be regarded as a scaleable variant of technique 2; indeed, Boehm et al. implemented it page-wise, using dirty bits. It is an incremental-update technique as well. It regresses the collection a bit, so some attention to scheduling is required to guarantee timely termination.

3.2.2 Invariant barriers for a black mutator

Note that if we're not maintaining the strong invariant, even a black mutator can contain pointers to white objects. What makes it black is that its state has been scanned and will not be scanned again (see section 2.2).

Technique 4 (Baker [Bak78]) *A read-barrier on grey: when hit, shade the referent, C.*

This is an incremental-update technique. Like other techniques involving the referents, it is not scaleable.

Technique 5 (Appel, Ellis & Li [AEL88]) *A read-barrier on grey: when hit, scan the object under the barrier, A, to turn it black.*

Scanning object A ensures the new referent, C, is not white, hence preserving the strong invariant. This can be regarded as a scaleable variation of technique 4.

Technique 6 (Abraham & Patel [AP87]) *A write-barrier on grey and white: when hit, shade the old referent, B.*

This is a snapshot-at-the-beginning technique, so it maintains the weak invariant only. Like other techniques involving the referents, it is not scaleable. Actually, Abraham and Patel, like Yuasa [Yua90], effectively had a write-barrier on everything, but grey and white is enough, since we're only concerned about losing paths that the collector was going to traverse.

Technique 7 *A write-barrier on grey and a read-barrier on white. When the write-barrier is hit, we shade the old referent, B. When the read-barrier is hit, we shade the referent, C.*

This is also a snapshot-at-the-beginning technique. Like other techniques involving the referents, it is not scaleable. In practice, it's inferior to technique 6, as it's almost the same except for having a read-barrier on white instead of a write-barrier, and read-barriers tend to be more expensive.

This technique works because all black-to-white pointers must have a copy in some grey or white object. When the mutator reads a pointer out of a grey object, we rely on the write-barrier to ensure that it will be scanned eventually (as in technique 6); when reading a pointer out of a white object, we turn the referent grey, thereby making sure it won't be lost. Note that this technique actually maintains an invariant that is slightly stronger than the weak invariant, namely that every white object pointed to by a black object is also pointed to by a grey object.

3.2.3 Completeness of the classification

More techniques can be produced from the techniques described above by doing more than necessary. In fact, many techniques listed in the previous sections could have been omitted and just mentioned in this section as variations, but were discussed because of their historical or practical interest.

- Where the description calls for turning an object grey, the collector could always scan it directly—although this might have a negative impact on scaleability.
- Where the description calls for a write-barrier turning the old referent B grey, the collector could scan object A instead to turn it black before the store (some of which will be wasted work, but this might be more scalable under some conditions).
- Likewise where a read-barrier description calls for turning the referent C grey (indeed, this is how techniques 4 and 5 are related).
- Where a barrier description calls for turning the referent C grey, we could just shade object A instead (optionally, only if C is white).

Finally, one could sometimes put barriers on further objects and twiddle them as well (this can be more practical, as the implementors of technique 6 discovered). However, we shall class all these variants with the basic technique, and prove that all the classes have been described above.

Theorem 1 *The techniques 1 through 7, together with their variants, cover all barrier techniques that use the actions described in section 3.2.*

Lemma 1 *Technique 1 (and its variants 2 and 3) cover all incremental-update barrier techniques for a grey mutator.*

The strong invariant cannot be guarded from a grey mutator without a write-barrier on black, because otherwise the mutator might write a white pointer into black. The techniques mentioned use such a barrier, so clearly more is not needed. The minimal action needed to preserve the invariant is either to shade object C or to turn object A grey. These actions are included in the three techniques listed, so they do cover all the possibilities.

Lemma 2 *There are no snapshot-at-the-beginning barrier techniques for a grey mutator.*

A grey mutator might contain a pointer to a white object that is the only pointer to that object. Writing that into a black object and then dropping the pointer would break the weak invariant. We can't prevent the mutator from dropping pointers, so we need to put some kind of write-barrier on black—but this leads to a technique maintaining the strong invariant (covered by lemma 1). Therefore there are no other techniques for a grey mutator.

Lemma 3 *Technique 4 (and its variant 5) cover all incremental-update barrier techniques for a black mutator.*

Obviously, the mutator couldn't stay black under the strong tricolour invariant, unless there's a read-barrier on grey. The techniques mentioned use such a barrier, so more is not needed. The minimal action required is to shade object C, and that is done in technique 4, so they do cover all the possibilities.

Lemma 4 *Techniques 6 and 7 cover all snapshot-at-the-beginning barrier techniques for a black mutator.*

With only the weak invariant, a grey object does not represent just the one path through it to reachable objects, it might also represent other paths from black directly to white. The collector cannot know which pointers from grey to white might be irreplaceable, so it must have a write-barrier protecting those. The minimal action to keep from losing such the white objects is to shade the old referent, B.

Additionally, we must do something about pointers from black to white objects that aren't directly pointed to from grey. Either we prevent them altogether or we make sure we don't lose the reachability from grey when white objects are modified.

The latter requires a write-barrier on white, and technique 6 fits this description. To conclude that it covers all the possibilities of this kind, we only have to observe, that among the actions listed in section 3.2, the only things that could possibly be of any use here is to shade or scan the old referent, B.

The former requires that we don't allow the mutator either to read pointers to such white objects or to write them into black. Such pointers can only be read from white objects, so we could put a read-barrier on white to catch that (technique 7 does that, in a minimal way), or we could make sure the mutator never sees white objects—but since the mutator is black that would mean maintaining the strong invariant. Preventing the writing of such pointers to black would imply a read-barrier on white, because the mutator itself is black, so that doesn't lead to any new techniques (attempts to handle the mutator separately from the black objects in the heap would make it grey).

So the techniques listed cover all the possibilities.

Lemmas 1 through 4 together cover the field of different mutator colors and barrier strategies, so we have found all the techniques. QED.

3.3 Switching and mixing barrier techniques

Since these techniques work by maintaining an invariant, the collector can switch from one barrier technique to another during the collection. However, if it switches between weak and strong techniques, then it can only maintain the weak invariant, since the strong invariant implies the weak one, but not vice versa.

We can also mix invariants, to get a guarantee that the collector will not miss a certain subset of pointers. This is useful in a mixed copying and non-moving collection, where the collector needs to find and fix all the pointers of the copied objects, but is allowed to miss pointers to the stationary objects. This could be regarded as a form of switching, where the technique to use is chosen on the basis of whether the pointer it protects could possibly refer to a moveable object (in the notation used in the barrier descriptions, if object C could be moveable).

4 Future barrier algorithms

The classification is only complete with respect to the preconditions set by the definition of the colours and the set of actions considered. On the other hand, all single-pass incremental GC algorithms known to the author can be analysed in terms of these techniques. For example, the replication-based algorithm of Nettles e.a. [NOPH92] can be described as a combination of a write-barrier for maintaining mutator consistency and technique 3.

The weak invariant was derived from general considerations, so it applies to any tracing algorithm, even ones that aren't stated in terms of tricolour marking. After all, the colours are merely a way to describe the progress of the traversal. However, algorithms can be constructed that make more passes over the heap (such as Bartlett's Mostly Copying [Bar88]); these circumvent the constraints established here, since a subsequent pass can revisit objects that were "black" in previous pass.

It is not obvious whether one could usefully extend the set of actions described in section 3.2. They'd have to be more complicated and thus probably less efficient. More than that, they would have to involve other objects, since the possibilities of A, B and C have been exhausted. Yet it's possible radically new algorithms remain to be discovered.

True concurrency sets much stricter requirements on tracing. If we can no longer assume that the mutator stops at the barrier, the analysis of this paper doesn't apply. This seems like an interesting area for further research.

5 Acknowledgements

[Jon96] was a valuable source of different GC algorithms. Thanks to my colleagues in the Adaptive Memory Management Group at Harlequin, particularly P. Tucker Withington and Gavin Matthews, for early comments on this work. Martin Simmons struggled through my first draft and helped make it clearer; if some parts of it are still obscure, it is probably because the author didn't take his advice.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[AP87] Santosh Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier-North Holland. Also technical report CSRD 620, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.

[BC92] Yves Bekkers and Jacques Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[BDS91] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[DLM+76] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[NOPH92] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92].

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.