The Locality of Concurrent Write Barriers

Laurence Hellyer
University of Kent
L.Hellyer@kent.ac.uk

Richard Jones
University of Kent
R.E.Jones@kent.ac.uk

Antony L. Hosking
Purdue University
hosking@cs.purdue.edu

Abstract

Concurrent and incremental collectors require barriers to ensure correct synchronisation between mutator and collector. The overheads imposed by particular barriers on particular systems have been widely studied. Somewhat fewer studies have also compared barriers in terms of their termination properties or the volume of floating garbage they generate. Until now, the consequences for locality of different barrier choices has not been studied, although locality will be of increasing importance for emerging architectures. This paper provides a study of the locality of concurrent write barriers, independent of the processor architecture, virtual machine, compiler or garbage collection algorithm.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); C.4 [*Performance of Systems*]: Measurement techniques

General Terms Measurement, Performance, Languages.

Keywords Language implementation, Memory management, Garbage collection, Java.

1. Introduction

Multicore and multiprocessor platforms are becoming ubiquitous, from large servers through desktops and laptops and even in embedded systems. In order to avoid becoming a bottleneck, it is essential that garbage collection algorithms take advantage of parallel hardware resources. There are a number of ways in which collectors can exploit parallelism. In a stop-the-world context, all mutator (user) threads may be stopped while a *parallel collection* is carried out by multiple garbage collector threads. This is likely to be the best solution if the goal is to achieve the highest throughput possible [35]. It is also the most straightforward way to harness the power of parallel hardware requiring no more synchronisation with mutator threads than a uniprocessor collector.

However, stop-the-world collection may lead to unacceptable pause times, especially if heaps are very large or worst case bounds are required. In order to reduce or bound pause times, either collector threads may be run *concurrently* with mutators, or each mutator thread may be required to perform some *incremental* garbage collection work. Both incremental and concurrent collection break the atomicity of garbage collection with respect to mutation of the object graph; object topology can change from one increment to the next or different threads' views of the topology may not be consistent. Unless care is taken to ensure that mutators and collectors share a coherent view of the heap, we risk mutators 'hiding' live objects from collector threads or, in the case of copying and compacting collectors, collectors moving objects behind the mutators' backs. We discuss this problem in more detail in Section 2. In either case, synchronisation between mutator and collector threads is essential.

Synchronisation is achieved through *barriers* that allow the mutator to communicate with the collector [22]. Barriers may be implemented by emitting code to instrument pointer loads or stores, or with operating system or hardware support [2, 11]. Moving collectors have typically intercepted pointer loads (with a *read barrier*) [4, 28]. Collectors that do not move objects use *write barriers* to intercept pointer stores [17, 33, 42]. Read barriers are generally considered to be more expensive than write barriers because of the prevalence of loads over stores [7]. The focus of this paper is software write barriers used by concurrent or incremental tracing collectors.

The role of a write barrier is to inform the collector of changes to the object graph topology. Let us consider how to barrier a write of a pointer p to the field f of an object o. Depending on policy, the barrier may notice either the *insertion* of the new pointer p into o or the *deletion* of the old pointer $o.f$. We give an outline of write barrier policies and mechanisms in Section 2. In each case, some object is *marked* (added to the collector's work list of objects to trace). Insertion barriers may advance the collector's wavefront of objects to trace by marking the new target p, or retreat it by marking the source o. Deletion barriers must mark the old target of $o.f$.

No collector can guarantee to trace only live objects (i.e. those that will be used in the future by the mutator); the problem is undecidable. Instead, tracing collectors approximate the set of live objects with the set of objects that are transitively reachable from the set of roots (local and global variables) by following pointers. If tracing is not atomic with respect to the mutators, the traced set will be larger than that obtained with a stop-the-world collector: concurrent/incremental tracing is *less precise* than stop-the-world tracing. The choice of barrier has profound consequences on the precision of the collector: deletion barriers are less precise than advancing insertion barriers which are in turn less precise than retreating insertion barriers [37]. The choice of barrier also affects how the collector must treat mutators' stacks and hence termination of the collector.

Barriers of different styles have different locality as they touch different objects, o, p or $o.f$. Locality has significant effects on performance, and designers of garbage collection algorithms take considerable care to optimise this [10, 14, 18]. Locality is likely to become more important as memory access becomes increasingly non-uniform. Blackburn and McKinley [7] observed that "the write barrier is a key to the efficiency of many modern garbage collectors.". Common folklore is that, despite their advantages in terms of ease of termination, deletion barriers lead to significantly

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

worse cache behaviour than installation barriers. In this paper, we seek to answer this question definitively by investigating the locality of different styles of concurrent write barriers. We seek to answer the question: what is the consequence of the choice of write barrier for the mutator? By abstracting from the details of any particular garbage collection algorithm, virtual machine or hardware platform, we provide empirical results and advice that is broadly applicable.

Structure of this paper Section 2 provides background on concurrent garbage collection and the challenges it faces in ensuring synchronisation between mutators and collectors. Section 3 provides an overview of existing write barriers and the subtle differences between them. Section 4 reviews related work and discusses the background to this research. Section 5 describes our methodology; results follow in Section 6. Further work and conclusions are presented in Section 7 and Section 8.

2. Barriers for concurrent GC

A *correct* collector must satisfy two properties.

- *Safety*: the collector retains *at least* all reachable objects;
- *Liveness*: the collector terminates, allowing it to free garbage.

Concurrent collectors are correct insofar as they are able to control mutator and collector interleavings. These are often most easily reasoned about by considering invariants that the collector and mutator must preserve based on the tricolour abstraction. All concurrent collectors preserve some realisation of these invariants, but they must retain at least all the reachable objects (safety) even as the mutator modifies objects.

2.1 The tricolour abstraction

The tricolour abstraction [17] is a useful characterisation of tracing collection that permits reasoning about collector correctness in terms of invariants that the collector must preserve. Using the tricolour abstraction, tracing collection partitions the object graph into black (live) and white (possibly dead) objects. Initially, every object is *white*; when an object is first encountered during tracing it is coloured *grey*; when it has been scanned and its children identified, it is shaded *black*. Conceptually, an object is black if the collector has finished processing it, and grey if the collector knows about it but has not yet finished processing it (or needs to process it again).

Tracing progress of the collector in the heap occurs by moving the collector *wavefront* (the grey objects) separating black objects from white objects until all reachable objects have been traced black. At the end of the trace there are no references from black to white (unreachable) objects, so they can safely be reclaimed. Concurrent mutation of the heap while the collector is working to advance the grey wavefront by scanning objects may destroy this invariant.

2.2 Tricolour invariants

To ensure that at the end of concurrent tracing there are no black objects that contain references to white objects, it is necessarily to preserve one of two invariants at all times.

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.

Of course, since problems can occur only when the mutator inserts a white pointer into a black object, it is sufficient simply to preserve:

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

Clearly, the strong invariant implies the weak invariant, but not the other way round. The strong invariant also means that a black mutator's roots can refer only to objects that are grey or black but not white. Under the weak invariant, a black mutator can hold white references so long as their targets are protected from deletion.

2.3 Precision

Different collector algorithms, which achieve safety and liveness in different ways, will have varying degrees of precision, efficiency, and atomicity. Precision is defined by the set of objects retained at the end of collection. A stop-the-world collector obtains maximal precision (all unreachable objects are collected) at the expense of any concurrency with the mutator. Finer grained atomicity permits increased concurrency with the mutator at the expense of possibly retaining more unreachable objects. Unreachable objects that are nevertheless retained at the end of the collection cycle are called *floating garbage*. It is important, though not necessary for correctness, that a concurrent collector also ensure *completeness* in collecting floating garbage at some later collection cycle.

2.4 Allocation colour

Mutator colour also influences the colour objects receive when they are allocated. Allocation results in the mutator holding the pointer to the newly allocated object, which must satisfy whichever invariant applies given the colour of the mutator. The allocation colour also affects how quickly a new object can be freed once it becomes unreachable. If an object is allocated black or grey then it will not be freed during the current collection cycle (since black and grey objects are considered to be live), even if the mutator drops its reference without storing it into the heap. A grey mutator can allocate objects white and so avoid unnecessarily retaining new objects. A black mutator cannot allocate white (whether the strong or weak invariant applies), unless (under the weak invariant) there is a guarantee that the white reference will be stored to a live object ahead of the wavefront so the collector will retain it. Otherwise, there is nothing to prevent the collector from reclaiming the object even though the black mutator retains a pointer to it. Initially, new objects contain no outgoing references so allocating black is always safe.

2.5 Incremental update solutions

Solutions that preserve the strong invariant are called *incremental update* techniques [39] since they inform the collector when the mutator tries to install a white pointer in a black object (behind the wavefront). Incremental update solutions conservatively treat an object as live (non-white) if a pointer to it is ever installed behind the wavefront, speculating that the mutator may yet delete all other paths to the object ahead of the wavefront. They use a mutator *write* barrier to protect against insertion of white pointers in black objects.

2.6 Snapshot-at-the-beginning solutions

Solutions that preserve the weak invariant are called *snapshot-at-the-beginning* techniques [39] since they inform the collector when the mutator deletes a white pointer from a grey or white object (ahead of the wavefront). Snapshot-at-the-beginning solutions conservatively treat an object as live (non-white) if a pointer to it ever existed ahead of the wavefront, speculating that the mutator may have also inserted that pointer behind the wavefront. Snapshot-at-the-beginning techniques use a mutator *write* barrier to protect against deletion of grey or white pointers from grey or white objects.

Under the weak invariant, deleting a grey or white reference from a black mutator is never a problem. However, deleting a white reference from a grey mutator might be, since that may remove the

```
1  atomic Write(src, fld, ref):
2     *fld ← ref
3     if is_black(src)
4        if is_white(ref)
5           revert(src)
```

(a) Steele [33]

```
1  atomic Write(src, fld, ref):
2     *fld ← ref
3     if is_black(src)
4        revert(src)
```

(b) Boehm et al. [11]

```
1  atomic Write(src, fld, ref):
2     *fld ← ref
3     if is_black(src)
4        shade(ref)
```

(c) Dijkstra et al. [17]

Figure 1: Grey mutator barriers.

```
1  atomic Read(src, fld):
2     ref ← *fld
3     if is_grey(src)
4        ref ← shade(ref)
5     return ref
```

(a) Baker [4]

```
1  atomic Read(src, fld):
2     if is_grey(src)
3        scan(src)
4     return *fld
```

(b) Appel et al. [2]

```
1  atomic Write(src, fld, ref):
2     if is_grey(src) or is_white(src)
3        shade(*fld)
4     *fld ← ref
```

(c) Abraham and Patel [1], Yuasa [42]

Figure 2: Black mutator barriers.

last link ahead of the wavefront to a chain of white objects that are otherwise reachable from black objects. However, we cannot prevent the mutator from deleting references it holds directly (that is, dropping objects). The only solution is to preempt the lost deletion by avoiding ever inserting a white reference in a black object, which degenerates to maintaining the strong invariant. Thus, snapshot collectors operate only with a black mutator.

3. Barrier techniques for concurrent collection

Barrier techniques that maintain one of the two tricolour invariants rely on a number of actions to cope with insertion or deletion of pointers. They can:

- Add to the wavefront by *shading* a white object grey. Shading an already grey or black object has no effect.
- Advance the wavefront by *scanning* an object to make it black.
- Regress the wavefront by *reverting* an object from black back to grey.

The only other actions — reverting an object to white or shading an object black without scanning — would break the weak or strong invariant.

Following Pirinen [27], Figures 1 and 2 enumerate the range of classical barrier techniques for concurrent collection.

3.1 Grey mutator techniques

We first consider approaches that operate with a grey mutator. All these techniques preserve the strong invariant by using an *insertion write barrier* to protect from storing white pointers into black objects. Because the mutator is grey they do not need a read barrier. They are incremental update techniques.

- Steele's barrier [33] (Figure 1a) yields the most precision of all the techniques because it simply notes the source object being modified. It does not change any decision about reachability of any object, but regresses the wavefront by changing the modified source object from black back to grey.
- Boehm et al. [11] implemented a variant of the Steele [33] barrier which ignores the colour of the inserted pointer (Figure 1b). Boehm et al. used virtual memory dirty bits to record pages modified by the mutator yielding a less precise barrier, and a stop-the-world phase to terminate collection (at which time the dirty pages are rescanned).
- Dijkstra et al. [17] designed a barrier (Figure 1c) that yields less precision than Steele's since it commits to shading the target of the inserted pointer reachable (non-white), even if the inserted pointer is subsequently deleted. This loss of precision aids progress by advancing the wavefront.

3.2 Black mutator techniques

The first two black mutator approaches apply incremental update techniques to maintain the strong invariant using an *insertion read barrier* to protect the mutator from acquiring white pointers. The third, a snapshot technique, uses a *deletion write barrier* to preserve the weak invariant. Under the weak invariant even a black mutator can contain white references. It is black only in that its roots have already been scanned by the collector — at the time they were scanned all the mutator roots became grey as their target objects were shaded (if they were not already black), but the mutator may have since loaded additional white pointers from the heap, adding them to its roots.

- Baker's read (mutator insertion) barrier [4] (Figure 2a) has less precision than Dijkstra et al., since it retains otherwise white objects whose references are loaded by the mutator at some time during the collection cycle, as opposed to those actually inserted behind the wavefront.
- Appel et al. [2] implemented a coarse-grained (less precise) variant of Baker's read barrier (Figure 2b), using virtual memory page protection primitives of the operating system to trap accesses by the mutator to grey pages of the heap without having to mediate those reads in software.
- Abraham and Patel [1] and Yuasa [42] independently devised the deletion barrier of Figure 2c which offers the least precision of all the techniques. Any unreachable object to which the last pointer was deleted during the collection cycle is retained.

3.3 Completeness of barrier techniques

Pirinen [27] argues that these barrier techniques cover the range of all possible approaches, with the exception a barrier that combines an insertion read barrier on a black mutator with a deletion read

barrier on the heap to preserve a weak invariant. All of the barrier techniques enumerated here cover the minimal requirements to maintain their invariants, but variations on these techniques can be obtained by short-circuiting or coarsening.

4. Related Work

The literature contains many studies of barriers for generational and concurrent/incremental collection and of the impact of particular barriers on the performance of particular systems [8, 12, 16, 19–21, 32, 34, 36, 40, 43]. However, few studies have compared different styles of barrier. Zorn counted proportions of loads and stores by large Franz Lisp programs in order to estimate the overhead of read and advancing write barriers. His work differs from ours in that he needed to trap all stores since Lisp is untyped. Note also that Franz Lisp was interpreted. Hosking et al. [20] compare barriers based on hash tables, sequential store buffers, card tables and virtual memory traps for generational collectors.

The work most closely related to ours is the study of Blackburn and Hosking [6] that measures the costs of a variety of read and write buffers for Jikes RVM. They compared the costs of barriers generated by an adaptive, optimising compiler for different Jikes RVM collection algorithms on different hardware platforms. In contrast, our work seeks to measure the locality of different write barriers independent of the platform, virtual machine or garbage collector used. Similarly to us, Blackburn and Hosking consider only the barrier itself and not the cost of manipulating work lists (sequential store buffers or card tables in their case).

Vechev et al. [37] start from an abstract specification of write barriers and derive Dijkstra's and Steele's insertion barrier [17, 33], the deletion barrier of Yuasa [42] and a hybrid of the two. They compare the space usage of the four barriers for a subset of the SPEC jvm98 benchmarks and conclude as expected that the Steele barrier is the most precise, followed by Dijkstra's and that the deletion barrier is significantly less precise than the others.

Locality is critically important to performance on modern architectures. In contrast to the work reported here, most studies have concentrated either on improving the locality of the tracing loop of mark-sweep collectors or on improving mutator locality by rearranging the layout of objects in the heap. Boehm [10] observes that marking dominates collection time (the cost of fetching the first pointer from an object accounted for a third of marking time on an Intel Pentium III), so he prefetches objects when a reference to them is added to the marking stack. Cher et al. [14] insert a FIFO queue in front of the mark-stack to better match the order that cache lines are fetched. Garner et al. [18] restructure the usual tracing loop to enqueue edges rather than nodes [22]: although this leads to more work as Java applications have about 40% more edges than nodes, this is outweighed by the reduction in cache misses. Several authors have considered how to use copying collection to improve the locality of mutators. Depth-first or pseudo depth-first copying can improve over breadth-first copying [24, 25, 31, 41]. The collector can also be proactively invoked to improve locality [13, 15]. Objects can also be reordered statically by type in order to colocate related objects or hot fields of objects [24, 26, 30, 41].

5. Methodology

We wish to explore the consequences for mutator locality of three different concurrent write barriers:

- Dijkstra's advancing insertion barrier [17];
- Steele's retreating insertion barrier [33]; and
- Yuasa's deletion barrier [42].

In each case we are interested in the number of cache misses a mutator running with a particular write barrier would incur that it would not have suffered if the mutator had run without the barrier. However, the number of cache misses incurred is specific to the environment in which a benchmark is run: the hardware, virtual machine and compiler as well as barrier and the benchmark used. Because we want to provide a platform-independent answer to the question of mutator/barrier locality, we abstract from these details by measuring those objects accessed by the write barrier that would not have been accessed by the benchmark otherwise.

5.1 Notice and use

Any write barrier must intercept pointer stores in order to add work to a tracing collector's work list. In this case, we say that the barrier has *noticed* the object. The user program (excluding barrier code) may also access the object. We say that the object has been *used* if a value is stored in or loaded from a field of the object. We include any mutator access to an object's header fields (e.g. to get an array's length, for type queries, method dispatch, hashing or locking). In order to ensure termination (without stopping the world unnecessarily), we shall assume that the barrier is filtering and that the colour of the object is stored in the object header: it must access an object to test and set its colour rather than unconditionally setting a byte in a card table [16]. We would expect our results to be different if the object colour was stored separately from the object. Algorithm 1 shows how we emulate the barriers.

- `Write(src,fld,new)` stores `new` in field `fld` of object `src`. If this is a reference write, one of the objects is noticed, depending on the barrier used. The `src` is always used.
- `markNoticed` is parameterised by the barrier-style under test to record either the object modified (`src`), the new target (`new`) or the old target (`old`). An objects is stamped with the time (bytes allocated) that it is first noticed. If is already used, we log the last used to noticed distance. A pointer stored to a static is treated similarly except that we only notice the static's old or new target.
- The first time that an object is used after it has been noticed (marked by the barrier), `markUsed` stamps its `used` field with the current time, and again logs this reuse distance. Note that as statics are never 'noticed', they can never be 'used'. The logs are post-processed to identify the minimum use-notice or notice-use distance for each object.

5.2 Barrier abstractions

We now describe the object access behaviour of concurrent write barriers using the `Write` operation of Algorithm 1. As we saw in Section 2, Dijkstra's barrier modifies `src` (to write the reference), reads `src` (to check its colour), tests and may grey `new`. Steele's barrier also modifies `src`, reads `src`, tests `new`, tests and may grey `src`. Yuasa's barrier tests `src`, tests and may grey `old` and modifies `src`. We summarise these in the following table (R = reads, W = writes).

	src	old	new
Dijkstra	RW		RW?
Steele	RW		R
Yuasa	RW	RW?	

All three barriers access the `src` object: no extra cache misses on `src` will occur. Both Dijkstra and Steele access `new`, and Yuasa accesses `old`. In each of these cases, an extra cache miss will occur if the barrier causes the loading of `new` (`old`) which then becomes evicted from the cache before the mutator accesses it. We treat

Algorithm 1: Pseudocode for the write barrier.

```
1  Write(src, fld, new):
2    if (referenceType(new))
3      old = *fld
4      markNoticed(src, old, new)
5    markUsed(src)
6    ...// write the new value
7
8  Read(src, field):
9    markUsed(src)
10   ...//return the value of the field
11
12 markNoticed(src, old, new):
13   ref = ...// select src, old or new
14   if (ref = null) return
15   if (gcInProgress()) return
16   atomic
17     if (ref.noticed = UNNOTICED)
18       ref.noticed = now
19       notice_count++
20       if ref.used = USED
21         log(ref.used−now, ref.id)
22         ref.used = UNUSED
23
24 markUsed(ref):
25   if (ref = null) return
26   if (gcInProgress()) return
27   atomic
28     if (ref.noticed = NOTICED)
29       if (ref.used = UNUSED)
30         ref.used = now
31         log(now−ref.noticed, ref.id)
32     else
33       ref.used = now
34
35 New(): // allocate black
36   ref = allocate()
37   ref.noticed = now
38   ref.used = now
39
40 clearNotice(ref)
41   ref.noticed = UNNOTICED
42   ref.used = UNUSED
```

the locality behaviour of the two insertion barriers as equivalent. In the next section, we measure the locality performance of (a) Dijkstra/Steele by setting `ref=new` (line 13) and (b) Yuasa by setting `ref=old`.

5.3 Garbage collector independence

An important principle of this study is that it is not dependent on any hardware platform, virtual machine or garbage collection algorithm. However, we do want to study the effect that a collector would have. For example, by the end of a garbage collection cycle, all live objects will have been marked and their colours reset to white in preparation for the next cycle, during which they may be touched again by the write barrier.

We also measure the effect on write barrier locality of varying the period of garbage collection cycles. Concurrent and incremental algorithms are usually tuned or self-adjusting to ensure that they complete all tracing work before the mutators run out of space. Garbage collection threads or increments are scheduled more frequently, for longer, or to do more work. The corollary is that the mutator makes less progress between collection cycles. We emulate this by running our measurement framework on top of a standard stop-the-world, non-generational collector but invoking the collector more or less frequently by varying the amount of allocation allowed between each GC. Whenever the underlying collector's tracing loop marks an object, it calls `clearNotice` to reset the object's `noticed` and `used` fields. Thus, we vary the window in which a mutator can notice and use an object.

5.4 Allocation colour

Objects can be allocated in different colours depending on the colour of the mutator [27]. Black mutators usually allocate black, even under the weak tricolour invariant. Grey mutators offer more possibilities, though again a common policy is to allocate black during the marking phase in order to speed termination [23, 38]. Similarly, we choose also to allocate objects as noticed (`New` in Algorithm 1).

6. Results

In this section, we describe our experimental platform, the benchmarks used, and present and evaluate our results.

6.1 Platform and Benchmarks

To measure the locality of the barriers, we instrumented Jikes RVM, an open source Java virtual machine written in Java. Its well-defined memory management toolkit, MMTk [9], allows easy modification of existing garbage collectors (or implementation of new ones). The version used was 3.1.0+trunk15808. In order to measure the benchmarks' mutator behaviour platform-independently, and to improve consistency between runs, we choose to omit the adaptive compiler and use the baseline compiler to reduce VM meta-data. We do not believe that the validity of our results is compromised by the choice of compiler. First, a goal of this work is to understand the locality behaviour of barriers independent of any compiler used. Second, common compiler optimisations like redundant load elimination do not affect our results: by definition RLE removes a load only because the value has already been loaded. Finally, the compiler's ability to optimise code across safe-points is restricted in a managed environment because the collector may move objects. We ran the `BaseBaseMarkSweep` configuration using a mark-sweep collector.

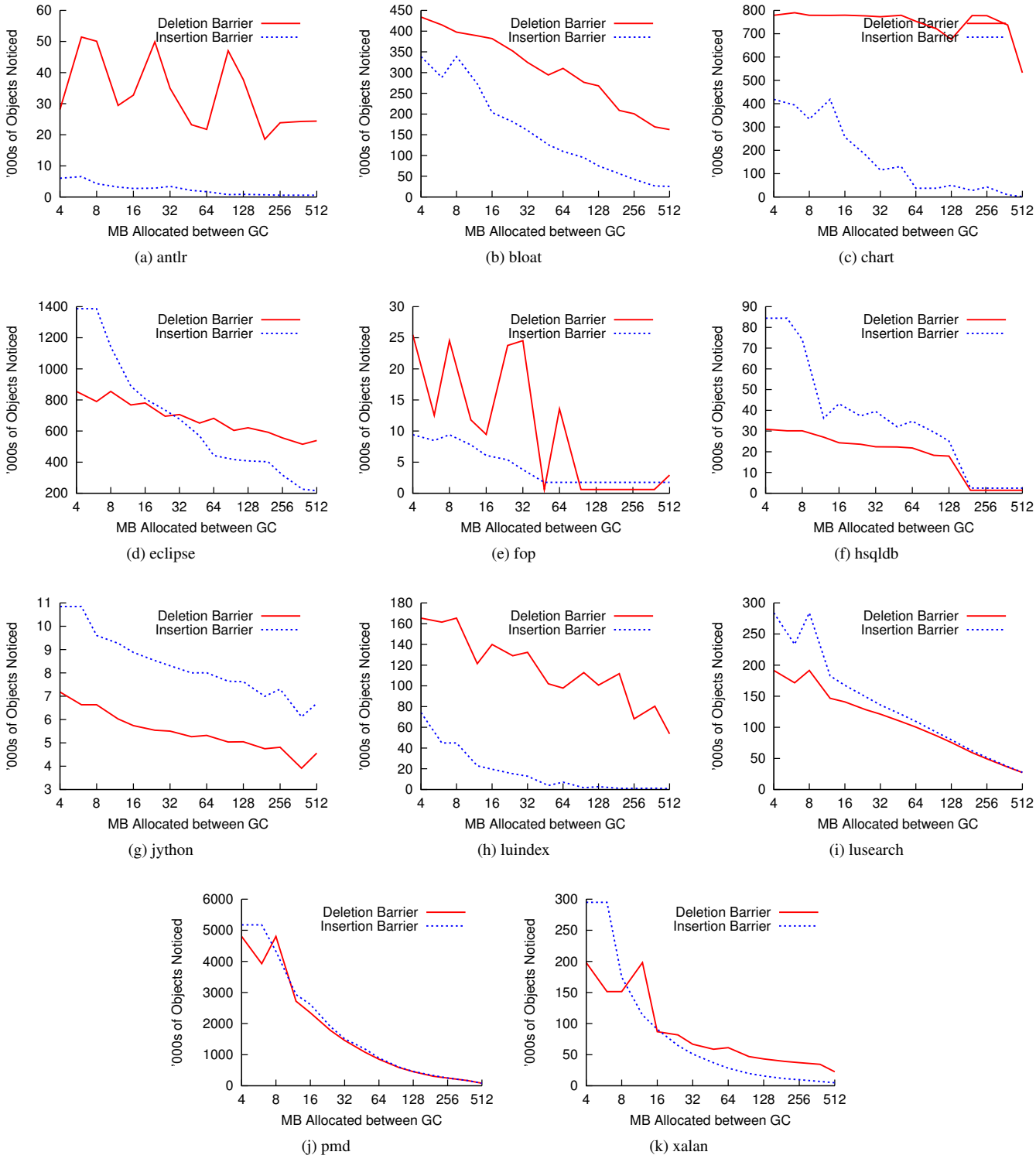
Object and array writes were intercepted using pre-existing support for write barriers within Jikes RVM and MMTk. Primitive writes marked the `src` object as used before updating the slot. Reference writes called `markNoticed` before marking the `src` object as used and updating the slot. The baseline compiler was modified to call `markUsed` on the `tgt` object via modified `getField`, `arrayLoad`, `monitorenter`, `invokevirtual`, `invokeinterface` and `Object.hashCode()` implementations. Marking objects as noticed or used during the underlying GC is disabled by each `markNoticed` and `markUsed` call checking a global flag.

6.2 Benchmarks

Our benchmarks were drawn from the DaCapo suite, 2006–10–MR2 [5], a set of real-world Java benchmarks with non-trivial and well known memory loads. In each case we use the default input and measure the objects noticed and used during the second iteration. Multiple iterations of each benchmark were performed to improve the robustness of the measurements.

6.3 Results

We investigated the number of objects noticed (and hence size of the work lists generated) by the barriers, the proportion of objects noticed by the barriers that were subsequently used by the mutator within the same collection cycle, and the reuse distance between noticing and using an object.



(a) antlr (b) bloat (c) chart

(d) eclipse (e) fop (f) hsqldb

(g) jython (h) luindex (i) lusearch

(j) pmd (k) xalan

Figure 3: Number of objects noticed by mutator write barriers.

Figure 3 shows the number of objects noticed by an insertion or a deletion barrier. For both barriers, the number of noticed objects tends to drop as the interval between collections increases. More frequent collections increase the number of times the mark-bits of long-lived objects are set and reset, leading to higher noticed counts. Although one might expect each barrier to touch the same number of objects (since `Write` inserts one pointer and deletes another), in most cases, insertion barriers touch fewer objects than deletion barriers. Any imbalance indicates that either (a) one barrier is applied to more pointers to shared objects than the other, (b) one barrier is more likely to try to notice new objects (which we allocated black) than the other, or (c) pointer writes cause work for one barrier but not the other, for example, by exchanging pointers to a young and old object (in either direction). For antlr, chart and luindex the difference is large. However, hsqldb, jython and lusearch notice more objects with the insertion barrier. The frequency of collection affects the predominance of either barrier in eclipse and xalan.

Figure 4 shows the fraction of objects that the mutator noticed and used in the same collection cycle. The tendency for most objects to be uniquely referenced means that once a pointer has been deleted its target cannot be accessed again. We had therefore expected deletion barriers to lead to lower proportion of noticed objects also being used. Surprisingly, in all cases except jython and lusearch, use of a deletion barrier leads to a larger proportion, suggesting better temporal locality. We note that the target of an inserted pointer may be not be accessed subsequently for some time, for instance not until after a further collection cycle (at which point the mark-bits will have been reset). However, the proportions for both barriers are typically high (over 90%). The exception is the insertion barrier for chart, which performs badly until the collection frequencies are increased to 16MB or more. With jython, the proportion with both barriers tends to rise slowly.

Figure 5 shows the reuse distance of noticed objects. A point (x, y) indicates that $y\%$ of the noticed objects are used by the mutator within x bytes of allocation. We have plotted the curves for all the collection frequencies in order to give an impression of the distributions. The cache behaviour induced by a barrier will be better for curves above and to the left in the figure. For example, the reuse with the deletion barrier is much better in antlr than it is in jython. Deletion barrier typically tend to lead to shorter reuse distances. In most cases, these distance are very short (e.g. less than a kilobyte). However, varying the collection frequency does effect the reuse distance. This is particularly noticeable for the insertion barrier in hsqldb and both barriers in pmd. Chart stands out: its insertion barrier has much worse reuse behaviour than its deletion barrier. The reuse distances for both pmd barriers at the 512MB collection frequency is large: we are uncertain why this is. We were surprised that most of the benchmarks showed the deletion barrier to have better reuse than the insertion barrier. This suggests that, contrary to popular folklore, although deletion barriers may notice more objects, this will not necessarily lead to a degradation of cache performance for the barrier, given a suitable (i.e. cache-friendly) mechanism for remembering noticed objects. Furthermore, the deletion barriers appear to be less sensitive to collection frequency. Again, we suggest that this may be because the last reference to an objects is deleted only once but that a reference to an object may be inserted many times. Whether the subsequent insertions shade the object depends on its colour which is reset at each collection.

7. Further work

This paper raises several interesting questions that could be explored in the future.

- Allocating black and/or the use of concurrent barriers leads to an over-estimate of the live heap. By implementing a full concurrent tracing collector on top of our concurrent barrier, it would be possible to measure the amount of floating garbage that each barrier preserves and the time taken for correct termination.
- Our experiments only measured the locality behaviour of the barriers themselves, and not their payloads. A further study would investigate different implementations of remembered sets for the noticed objects.
- The allocation time between noticing and using an object serves as a proxy for cache behaviour. Our results could be validate by measuring the real cache behaviour of the barriers on a variety of hardware architectures.

8. Conclusion

We have compared the cache behaviour of mutator insertion and deletion write barriers, for concurrent/incremental collectors, making efforts to ensure our results are VM, GC and hardware agnostic.

Generally we confirm that deletion barriers generate more work for a concurrent GC than insertion barriers. For some benchmarks the amount of work generated largely depends on the selected barrier and the heap size. The ratio between `used` and `noticed` is greater than 0.9 for both barriers and most benchmarks. Generally the deletion barrier has a higher use ratio. The time between noticing and subsequently using an object can be much lower with a deletion barrier than an insertion barrier, suggesting that deletion barriers may lead to better cache performance than has hitherto been expected.

References

[1] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *Int. Conf. on Parallel Processing*, 243–246, 1987. Pennsylvania State University Press.

[2] A.W. Appel, J.R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] D.F. Bacon and A. Diwan, editors. *Int. Symp. on Memory Management*, 2004. ACM Press.

[4] H.G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[5] S.M. Blackburn, R. Garner, K.S. McKinley, A. Diwan, S.Z. Guyer, A.L. Hosking, J.E.B. Moss, and D. Stefanović. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[6] S.M. Blackburn and A.L. Hosking. Barriers: Friend or foe? In Bacon and Diwan [3], 143–151.

[7] S.M. Blackburn and K.S. McKinley. In or out? putting write barriers in their place. In *Int. Symp. on Memory Management*, 175–184, 2003.

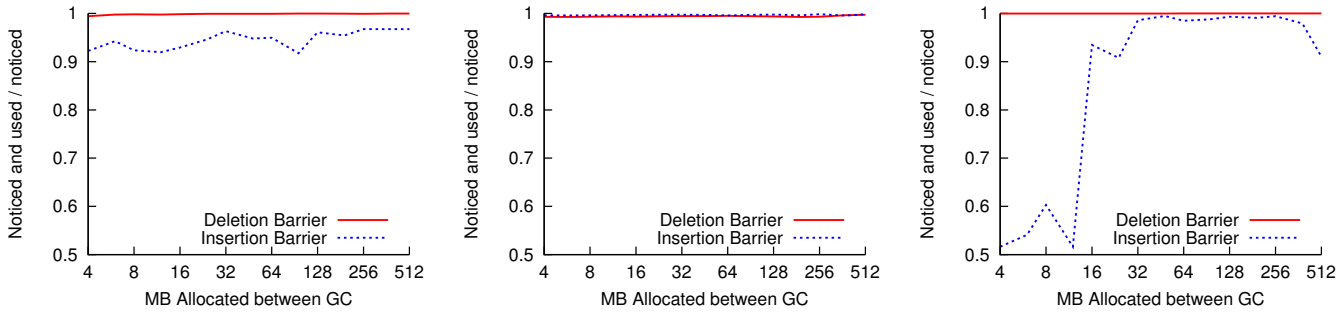
[8] S.M. Blackburn, R.E. Jones, K.S. McKinley, and J.E.B. Moss. Beltway: Getting around garbage collection gridlock. In *Programming Language Design and Implementation*, 153–164, 2002.

[9] S.M. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Int. Conf. on Software Engineering*, 137–146, 2004.

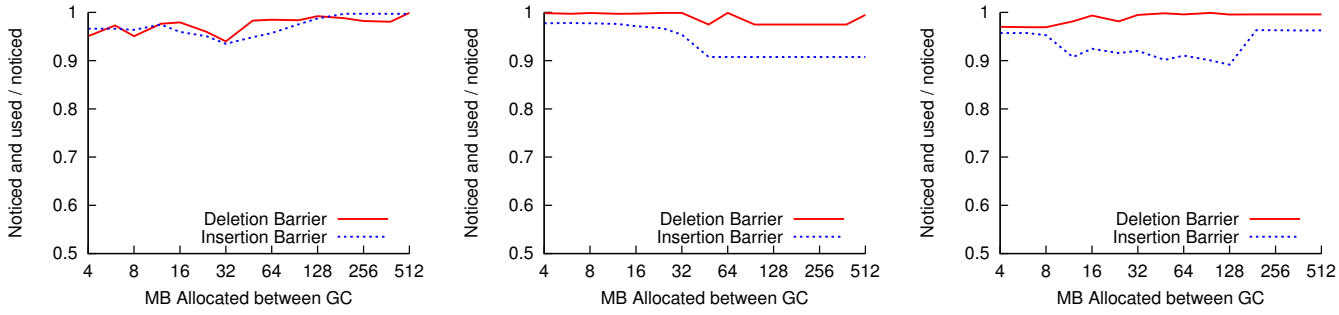
[10] H.-J. Boehm. Reducing garbage collector cache misses. In *Int. Symp. on Memory Management*, 59–64, 2000.

[11] H.-J. Boehm, A.J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

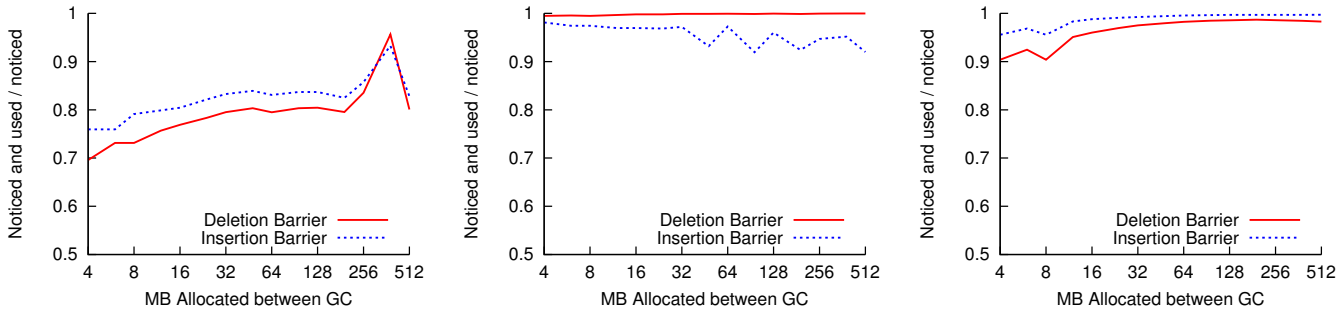
[12] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, 1992.



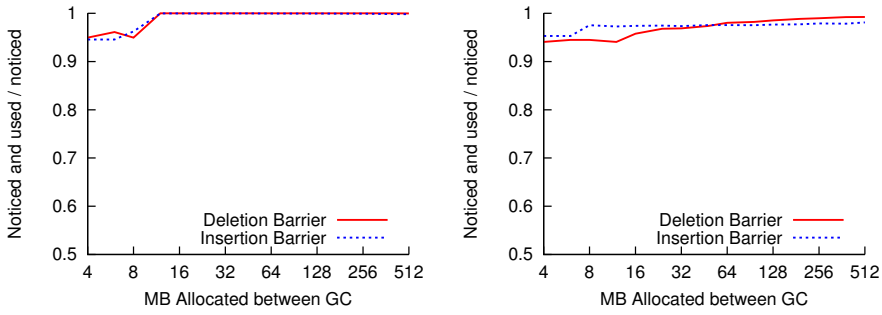
(a) antlr (b) bloat (c) chart



(d) eclipse (e) fop (f) hsqldb



(g) jython (h) luindex (i) lusearch



(j) pmd (k) xalan

Figure 4: Number of objects noticed & used by mutators / number of objects noticed by mutator write barrier.

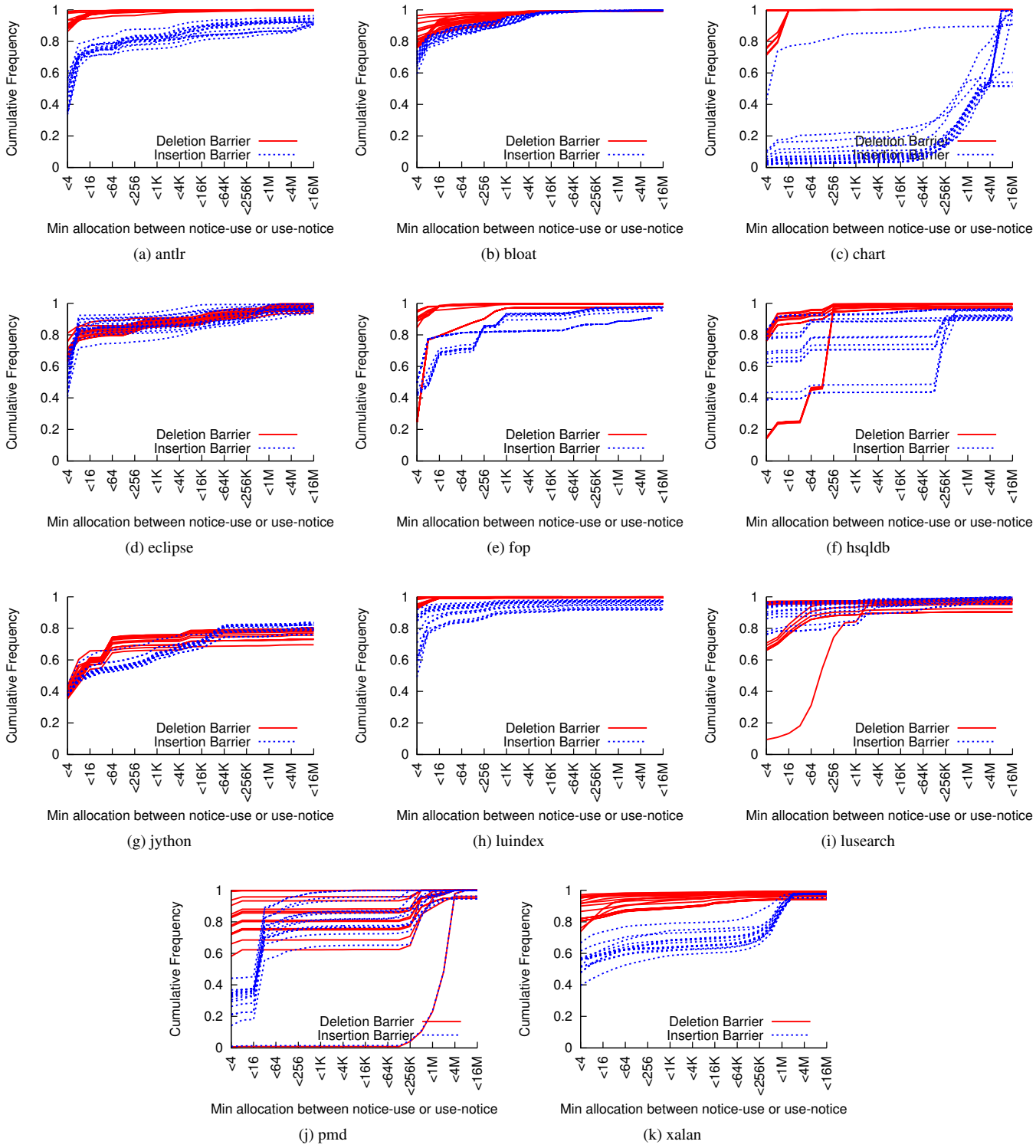


Figure 5: The distribution of minimum distance between noticing an object and using the object. A point (x, y) indicates that $y\%$ of the noticed objects are used by the mutator within $\pm x$ bytes of allocation. The figures show curves for each barrier and for each collection frequency.

[13] W.-K. Chen, S. Bhansali, T.M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In Schwartzbach and Ball [29], 332–340.

[14] C.-Y. Cher, A.L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *Architectural Support for Programming Languages and Operating Systems*, 199–210, 2004.

[15] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-conscious structure layout. In *Programming Language Design and Implementation*, 1–12, 1999.

[16] D. Detlefs, W.D. Clinger, M. Jacob, and R. Knippel. Concurrent remembered set refinement in generational garbage collection. In *Java Virtual Machine Research and Technology Symposium*, 2002.

[17] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, 1978.

[18] R. Garner, S.M. Blackburn, and D. Frampton. Effective prefetch for mark-sweep garbage collection. In *Int. Symp. on Memory Management*, 43–54, 2007.

[19] U. Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, 1993. URL `ftp://self.stanford.edu/pub/papers/write-barrier.ps.Z`.

[20] A.L. Hosking, J.E.B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *Object-Oriented Programming, Systems, Languages, and Applications*, 92–109, 1992.

[21] A.L. Hosking and J.E.B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Symp. on Operating Systems Principles*, 106–119, 1993.

[22] R.E. Jones, and R.Lins *Garbage Collection*. Wiley, 1996.

[23] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. In *Symp. on Foundations of Computer Science*, 120–131. IEEE Press, 1977.

[24] M.S. Lam, P.R. Wilson, and T.G. Moher. Object type directed garbage collection to improve locality. In *Int. Workshop on Memory Management*, Lecture Notes in Computer Science 637, 1992. Springer.

[25] D.A. Moon. Garbage collection in a large LISP system. In *Lisp and Functional Programming*, 235–245, 1984.

[26] G. Novark, T. Strohman, and E.D. Berger. Custom object layout for garbage-collected languages. Tech. Report, University of Massachusetts at Amherst, 2006.

[27] P.P. Pirinen. Barrier techniques for incremental tracing. In *Int. Symp. on Memory Management*, 20–25, 1999.

[28] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *Programming Language Design and Implementation*, 33–44, 2008.

[29] M.I. Schwartzbach and T. Ball, editors. *Programming Language Design and Implementation*, 2006.

[30] Y. Shuf, M. Gupta, R. Bordawekar, and J.P. Singh. Exploiting prolific types for memory management and optimizations. In *Principles of Programming Languages*, 2002.

[31] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *Int. Symp. on Memory Management*,52–63, 2006.

[32] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Tech. Report AITR-1417, MIT AI Lab, 1988.

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

[34] D. Stefanović, K.S. McKinley, and J.E.B. Moss. Age-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications*, 370–381, 1999.

[35] Sun Microsystems. Java SE 6 HotSpot virtual machine garbage collection tuning, 2009. URL `http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html`.

[36] D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5): 157–167, 1984.

[37] M. Vechev, D.F. Bacon, P. Cheng, and D. Grove. Derivation and evaluation of concurrent collectors. In *European Conf. on Object-Oriented Programming*, 2005. Springer-Verlag.

[38] M.T. Vechev, E. Yahav, and D.F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In Schwartzbach and Ball [29], 341–353.

[39] P.R. Wilson. Uniprocessor garbage collection techniques. Tech. Report, University of Texas, 1994.

[40] P.R. Wilson and T.G. Moher. Design of the opportunistic garbage collector. In *Object-Oriented Programming, Systems, Languages, and Applications*, 23–35, 1989.

[41] P.R. Wilson, M.S. Lam, and T.G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Programming Language Design and Implementation*, 177–191, 1991. doi: `10.1145/113445.113461`.

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

[43] B. Zorn. Barrier methods for garbage collection. Tech. Report CU-CS-494-90, University of Colorado, Boulder, 1990.