SCHOOL OF COMPUTING

Technical Report 03-10

The Locality of Concurrent Write Barriers

(Extended version)

Laurence Hellyer, University of Kent
Richard Jones, University of Kent
Antony Hosking, Purdue University

University of Kent | Computing

Technical Report 03-10: The Locality of Concurrent Write Barriers (extended version)

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

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

Antony L. Hosking
Department of Computer Science
Purdue University, West Lafayette, IN
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 report expands upon Hellyer et al. [2010] and provides a study of the locality of concurrent write barriers, independent of the processor architecture, virtual machine, compiler or garbage collection algorithm.

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 [Sun Microsystems, 2009]. 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 [Jones, 1996]. Barriers may be implemented by emitting code to instrument pointer loads or stores, or with operating system or hardware support [Appel et al., 1988; Boehm et al., 1991]. Moving collectors have typically intercepted pointer loads (with a *read barrier*) [Baker, 1978; Pizlo et al., 2008]. Collectors that do not move objects use *write barriers* to intercept pointer stores [Dijkstra et al., 1978; Steele, 1975; Yuasa, 1990]. Read barriers are generally considered to be more expensive than write barriers because of the prevalence of loads over stores [Blackburn and McKinley, 2003]. 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 [Vechev et al., 2005]. 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 [Boehm, 2000; Cher et al., 2004; Garner et al., 2007]. Locality is likely to become more important as memory access becomes increasingly non-uniform. Blackburn and McKinley [2003] 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 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 ask 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 report 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 [Dijkstra et al., 1978] 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.

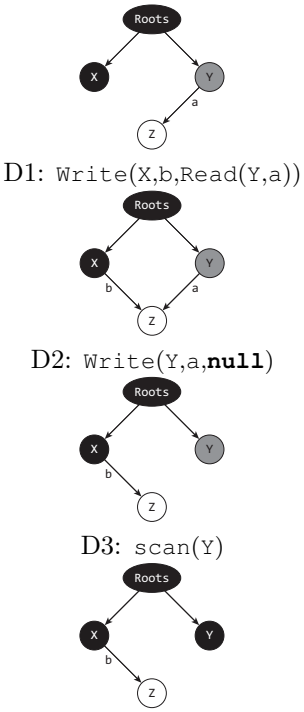
2.4 Mutator colour

In classifying algorithms it is also useful to talk about the colour of the mutator roots as if the mutator itself were an object. A grey mutator either has not yet been scanned by the collector so its roots are still to be traced, or its roots have been scanned but need to be rescanned. This means that the grey mutator roots may refer to objects that are white, grey or black. A black mutator has been scanned by the collector so its roots have been traced, and will not be scanned again. Under the strong invariant this 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.

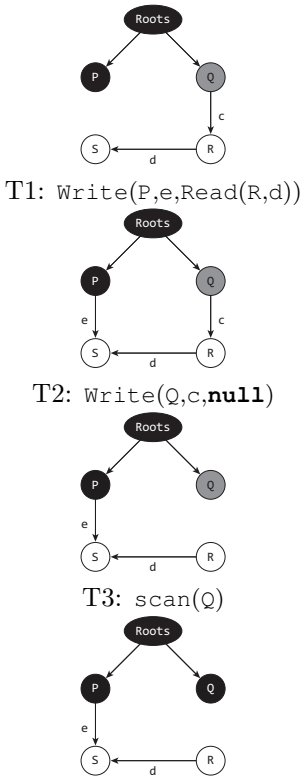
Our simplifying assumption for now is that there is only a single mutator. However, *on-the-fly* collectors distinguish among multiple mutator threads because they do not suspend them all at once to sample their roots. These collectors must operate with mutator threads of different colours, both grey (unscanned) and black (scanned). Moreover, some collectors may separate a single mutator thread's roots into scanned (black) and unscanned (grey) partitions. For example, the top frame of a thread's stack may be scanned to make it black, while the remaining stack frames are left unscanned (grey). Returning or unwinding into the grey portion of the stack forces the new top stack frame to be scanned.

2.5 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.



(a) Direct: hiding a reachable white object by dropping a direct link from grey.



(b) Transitive: hiding a transitively reachable white object by breaking an indirect chain from grey.

Figure 1: The lost object problem: a reachable white object is hidden from the collector by making it unreachable from grey.

2.6 The lost object problem

There are two scenarios under which a white pointer can be inserted behind the wavefront in Figure 1 [Vechev et al., 2005, 2006]. Figure 1a illustrates how the mutator can hide a white object *directly* reachable from a grey object by inserting its pointer behind the wavefront and then deleting its link from the grey object. The initial state of the heap shows a black object `X` and grey object `Y`, having been marked reachable from the roots. White object object `Z` is directly reachable from `Y`. In step D1 the mutator inserts pointer `b` from `X` to `Z` by copying pointer `a` from grey object `Y`. In step D2 the mutator deletes unscanned pointer `a` from the only unscanned object `Y` that refers to `Z`. In step D3 the collector scans the object `Y` to make it black, and terminates its marking phase. White object `Z` will be erroneously reclaimed, even though it is reachable via pointer `b`.

Figure 1b shows how the mutator can hide a white object *transitively* reachable via a chain of pointers from a grey object by inserts its pointer behind the wavefront and then deleting some *other* link in the chain. No pointer to the lost object itself is deleted, unlike the *direct* case which does delete a pointer to the lost object. The initial state of the heap shows a black object `P` and grey object `Q`, having been marked reachable from the roots. White object `R` is directly reachable from `Q`, while white object `S` is transitively reachable from `Q` via `R`. In step T1 the mutator inserts pointer `e` from `P` to `S` by copying pointer `d` from white object `R`. In step T2 the mutator deletes pointer `c` to `R`, destroying the path from the only unscanned object `Q` that leads to `S`. In step T3 the collector scans the object `Q` to make it black, and terminates its marking phase. White object `S` will be erroneously reclaimed, even though it is reachable via pointer `e`.

Objects can only become lost if both the weak and strong tricolour invariants are breached. In both scenarios, the mutator first writes a pointer to a white object into a black object (D1/T1), breaking the strong invariant. It then destroys all paths to that white object from grey objects (D2/T2), breaking the weak invariant. The result is that a (reachable) black object ends up pointing to a (garbage) white object, violating correctness. Solutions to the lost object problem operate at either of the steps that write the pointer to the white object (D1/T1) or delete the remaining paths to that object (D2/T2).

2.7 Incremental update solutions

Solutions that preserve the strong invariant are called *incremental update* techniques [Wilson, 1994] 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.

2.8 Snapshot-at-the-beginning solutions

Solutions that preserve the weak invariant are called *snapshot-at-the-beginning* techniques [Wilson, 1994] 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 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 [1999], Figures 2 and 3 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 [Steele, 1975] (Figure 2a) 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. It defers deciding reachability of the target white object until the source object can be rescanned (the inserted pointer might be deleted before rescanning). This precision comes at the cost of progress, since the wavefront is regressed.
- Boehm et al. [1991] implemented a variant of the Steele [1975] barrier which ignores the colour of the inserted pointer (Figure 2b). 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).

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

(a) Steele [1975]

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

(b) Boehm et al. [1991]

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

(c) Dijkstra et al. [1978]

Figure 2: 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 [1978]

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

(b) Appel et al. [1988]

```
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 [1987]; Yuasa [1990]

Figure 3: Black mutator barriers.

- Dijkstra et al. [1978] designed a barrier (Figure 2c) 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. The original formulation of this barrier shaded the target without regard for the colour of the source, with a further loss of precision. Omitting this extra check permits relaxing atomicity, so long as the store and the shade operations are separately atomic. The store must still be performed ahead of the shade operation so as to avoid a subtle race when the collector transitions from one collector cycle to the next in the middle. If the operations are inverted then a collector cycle transition right after shading the stored `ref` grey can revert the stored `ref` white and scan the `src` to black before the store, which then creates a black to white pointer violating the strong invariant [Stenning, 1976].

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 [Baker, 1978] (Figure 3a) 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. [1988] implemented a coarse-grained (less precise) variant of Baker's read barrier (Figure 3b), 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 [1987] and Yuasa [1990] independently devised the deletion barrier of Figure 3c 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. At D2 it directly shades `Z` grey. At T2 it shades `R` grey so that `S` can eventually be shaded. With an insertion barrier at least we know that the mutator has had some interest in objects retained by the barrier (whether to acquire or store its reference), whereas the deletion barrier retains objects regardless of whether the mutator manipulated them.

3.3 Completeness of barrier techniques

Pirinen [1999] 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. The black mutator can safely acquire a white pointer from some grey source object since the target object will eventually be shaded grey when the grey source is scanned, or the write barrier will shade the target grey if the source field is modified. The read barrier makes sure that the mutator never acquires a white pointer from a white object. Thus, every reachable white object has a grey object directly keeping it alive throughout the collection cycle.

Variations on the listed techniques can be obtained in various ways by short-circuiting or coarsening some steps, including:

- Shading an object grey can be short-circuited by immediately scanning the object to make it black.
- A deletion barrier that shades the target of the deleted pointer grey can instead (and more coarsely) scan the source object containing the deleted pointer to black before the store.
- A read barrier that shades the target of the loaded pointer grey can instead (and more coarsely) scan the source object to black before the read. Thus, the read barrier of Appel et al. coarsens that of Baker.
- An insertion barrier that shades the target of the inserted pointer grey can instead revert the source to grey. This is how the barriers of Steele and Boehm et al. gain precision over that of Dijkstra et al..

Clearly, all strong invariant (incremental update) techniques must at least protect from a grey mutator inserting white pointers into black, or protect a black mutator from acquiring or using white pointers. The strong techniques all do one of these two things and need not do any more.

We have already argued that weak invariant (snapshot) techniques must operate with a black mutator. Under the weak invariant, a grey object does not merely capture a single path to reachable white objects. It may also be a placeholder for a pointer from a black object to some white object on that path. Thus, the snapshot barrier must preserve any white object directly pointed to from grey. The least it can do is to shade the white object when its pointer is deleted from grey.

To deal with white objects transitively reachable via a white path from a grey object (which may also be pointed to from black) we can either prevent the mutator from obtaining pointers to white objects on such paths so it can never modify the path [Pirinen, 1999], or make sure that deleting a pointer from a white object (which may be on such a path) at least makes the target of the pointer grey [Abraham and Patel, 1987; Yuasa, 1990].

All of the barrier techniques enumerated here cover the minimal requirements to maintain their invariants.

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 [Blackburn et al., 2002; Chambers, 1992; Detlefs et al., 2002; Hölzle, 1993; Hosking et al., 1992; Hosking and Moss, 1993; Sobalvarro, 1988; Stefanović et al., 1999; Ungar, 1984; Wilson and Moher, 1989; Zorn, 1990]. 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. [1992] 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 [2004] 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. [2005] start from an abstract specification of write barriers and derive Dijkstra's and Steele's insertion barrier [Dijkstra et al., 1978; Steele, 1975], the deletion barrier of Yuasa [1990] 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 [2000] 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. [2004] insert a FIFO queue in front of the mark-stack to better match the order that cache lines are fetched. Garner et al. [2007] restructure the usual tracing loop to enqueue edges rather than nodes [Jones, 1996]: 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 [Lam et al., 1992; Moon, 1984; Siegwart and Hirzel, 2006; Wilson et al., 1991]. The collector can also be proactively invoked to improve locality [Chen et al., 2006; Chilimbi et al., 1999]. Objects can also be reordered statically by type in order to colocate related objects or hot fields of objects [Lam et al., 1992; Novark et al., 2006; Shuf et al., 2002; Wilson et al., 1991].

5 Methodology

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

- Dijkstra's advancing insertion barrier Dijkstra et al. [1978];
- Steele's retreating insertion barrier Steele [1975]; and
- Yuasa's deletion barrier Yuasa [1990].

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 [Detlefs et al., 2002]. 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 parametrised 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 = read, W = write, $W?$ = conditional write).

	`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 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`.

Algorithm 1: Pseudocode for the write barrier Magenta text highlights differences between this Algorithm and Algorithm 2.

```
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   // Called by arraylength, getField, invokeInterface, invokeVirtual
26   // monitorEnter, monitorExit, Object.hashCode() and putField
27   if (ref = null) return
28   if (gcInProgress()) return
29   atomic
30     if (ref.noticed = NOTICED)
31       if (ref.used = UNUSED)
32         ref.used = now
33         log(now-ref.noticed, ref.id)
34     else
35       ref.used = now
36
37 New(): // allocate black
38   ref = allocate()
39   ref.noticed = now
40   ref.used = now
41
42 clearNotice(ref)
43   ref.noticed = UNNOTICED
44   ref.used = UNUSED
```

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 invoke 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 [Pirinen, 1999]. 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 [Kung and Song, 1977; Vechev et al., 2006]. Similarly, we choose also to allocate objects as noticed (`New` in Algorithm 1).

6 Results

We now describe our experimental platform, the benchmarks we used, how they were measured, and we 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 [Blackburn et al., 2004], 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 (RLE) 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 to reset the *noticed* state of heap objects at different allocation frequencies (`clearNotice` in Algorithm 1).

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 `arrayLoad`, `getField`, `invokeinterface`, `invokevirtual`, `monitorenter`, `monitorexit`, `Object.hashCode()` and `putField`. 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 [Blackburn et al., 2006], 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 (to remove the effects of class loading and compilation work). Multiple iterations of each benchmark were performed to improve the robustness of the measurements. Details of the benchmarks are given in Table 1.

Program	Threads	Max. live (MB)	Allocation (MB)
antlr	1	1.0	237.9
bloat	1	6.2	1217.9
chart	1	9.5	742.8
eclipse	1	27.6	3902.8
fop	1	6.9	100.3
hsqldb	20	71.9	142.7
jython	1	0.1	1183.4
luindex	1	1.0	448.4
lusearch	32	10.9	1780.9
pmd	1	13.7	779.7
xalan	8	25.5	60235.6

Table 1: The DaCapo benchmark suite, v. 2006-M2, baseline compiled. *Max. live* is the largest volume of data live at any point in the program, *Allocation* is total allocation [The DaCapo Group, 2006].

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.

6.3.1 Number of objects noticed

Figure 4 on page 15 shows the number of objects noticed by an insertion (blue line) or a deletion barrier (red line) with Java finalization turned off. Finalization is invoked before an object is reclaimed by the GC to free resources not automatically managed by the VM (i.e. network connections) [Gosling et al., 1997]. Each benchmark was invoked three times for each barrier for each collection frequency. The results presented are for the second iteration of the benchmark harness with System.GC() disabled. Each invocation is plotted on the same graph. For both barriers, the number of noticed objects tended to drop as the interval between collections increases. Less frequent collections decrease the number of times the mark-bits of long-lived objects are reset, leading to lower noticed counts. Perhaps intuitively obvious: with finalization turned off we found the insertion barrier always noticed more objects than the deletion barrier.

Figure 5 on page 16 is similar to Figure 4 except that it is for finalization turned on (the default in many JVMs). Finalization is non-deterministic [Gosling et al., 1997] and should occur in a separate thread [Boehm, 2003]. This non-determinism can be seen in the increased noise between invocations in Figure 5.

Comparing Figure 4 to Figure 5 the number of objects noticed by the insertion barrier does not particularly change between finalization on and off. However, the number of objects noticed by the deletion barrier dramatically increased for antlr, bloat, eclipse, fop, luindex and xalan with finalization on. In these cases (with the exception of eclipse) the deletion barrier noticed more objects than the insertion barrier with finalization turned on. For eclipse the frequency of collection affects the predominance of either barrier.

Although one might expect each barrier to touch the same number of objects (since `Write` inserts one pointer and deletes another), in appears that which barrier touches the most objects is benchmark and finalization dependent. 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).

Whilst we present the results for the chart benchmark we refrain from commenting on them. Chart is an unusual benchmark in that it performs much of it its work via JNI. It is not clear that the results from chart are representative of typical Java applications.

To summarise our key findings:

- With finalization turned off the insertion barrier always noticed more objects than the deletion barrier.
- The number of objects noticed by an insertion barrier is largely independent of finalization.

Finalization	Deletion Barrier	Insertion Barrier
Off	Noticed then Used	Noticed then Used
On	Used then Noticed	Noticed then Used

Table 2: The predominant ordering of Noticed and Used

- Finalization dramatically increases the number of objects noticed by a deletion barrier for antlr, bloat, eclipse, fop, luindex and xalan.
- The dominant barrier with finalization on is benchmark and collection scheduling dependant.

6.3.2 Number of objects noticed and reused

Figure 6 and Figure 7 on pages 17 and 18 show the fraction of objects that the mutator noticed *and* used in the same collection cycle. For example, the results show that with finalization on at least 70% of objects (excluding chart) that are noticed by a deletion barrier are also used within a 4MB window (the red line for jython). For the insertion barrier at least 75% of objects are noticed and used within a similar 4MB window (the blue line for jython).

In almost all cases the proportion of noticed object that are also used rises as the interval between collections increases. This is as to be expected since the mutator is given longer to use a noticed object. The faction of noticed objects being used for both barriers is typically high (over 90%).

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 bloat, jython and lusearch, use of a deletion barrier leads to more noticed objects also being used, 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).

Our framework allowed us to determine if a use of an object occurred before or after the notice of the object. Whilst it is not immediately obvious how the pattern of access could be exploited we present the full results in Appendix A. Table 2 summarises the results and shows that the dominant ordering of noticing and using an object is noticed then used for both barriers with finalization off. With finalization on most objects are used then noticed by the deletion barrier.

To summarise our key findings:

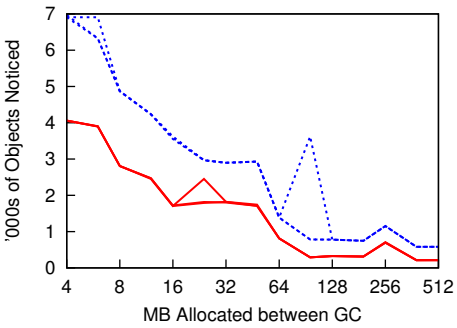
- The fraction of noticed objects that are also used in the same collection cycle is typically high (>90%).
- In all cases except bloat, jython and lusearch, use of a deletion barrier leads to more noticed objects also being used, suggesting better temporal locality.

6.3.3 Predicting locality

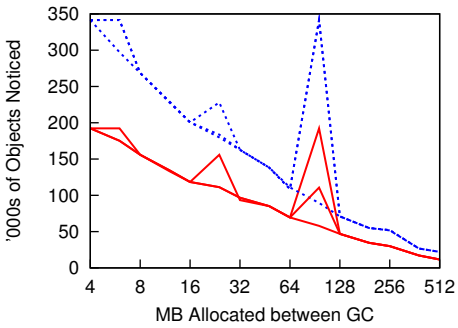
Figure 8 and Figure 9 on pages 20 and 21 show the reuse distance of noticed objects. A point (x, y) indicates that $y\%$ of the noticed objects are used by the mutator within $\pm x$ bytes of allocation. Where we recorded both a notice-then-use of an object and a use-then-notice of the same object in the same GC cycle we plot the smaller of the two reuse distances. We have plotted the curves for all the simulated collection frequencies and invocations in order to give an impression of the distributions. The cache behaviour induced by a barrier will be better for curves to the top and to the left in the figures. For example, the reuse with the deletion barrier is much better in antlr than it is in jython in Figure 8.

Deletion barriers typically tend to lead to shorter reuse distances in both Figure 8 and Figure 9. The only exception appears to be bloat where with finalization off the insertion barrier has a smaller allocation distance between noticing and using an object.

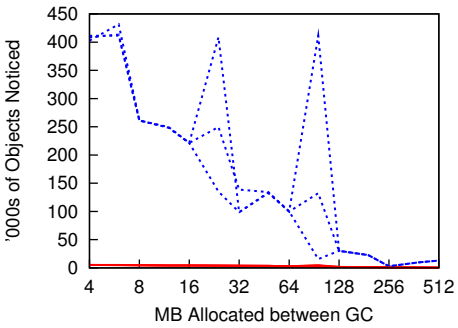
In most cases, the deletion reuse distances are very short (e.g. less than a kilobyte). However, varying the collection frequency does effect the reuse distance. This is particularly noticeable for the insertion barrier in hsqldb and both



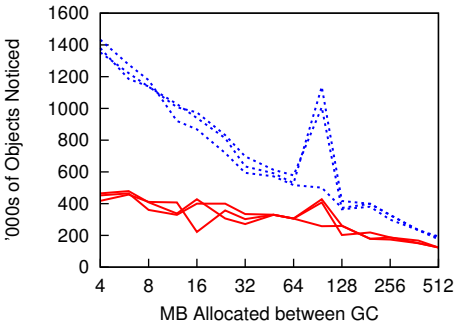
(a) antlr



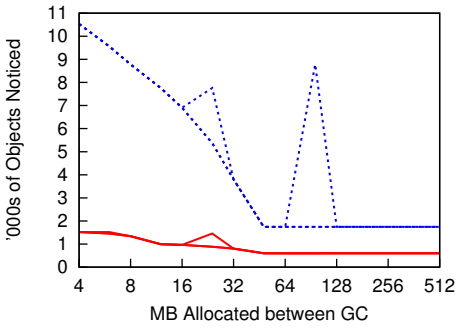
(b) bloat



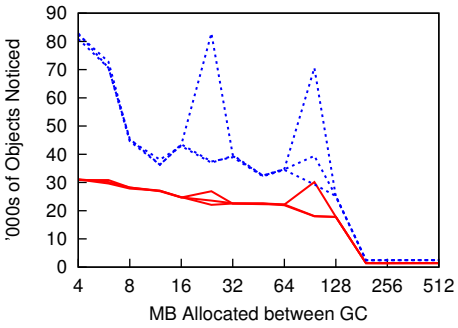
(c) chart



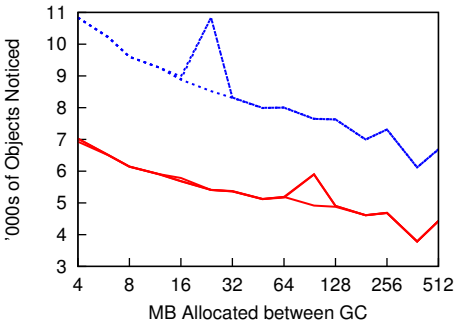
(d) eclipse



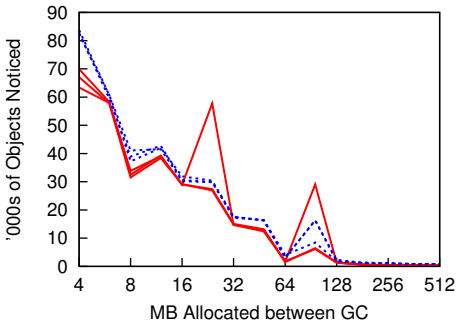
(e) fop



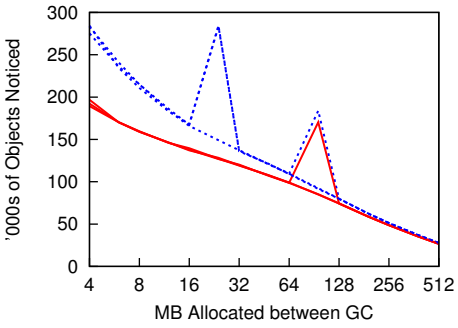
(f) hsqldb



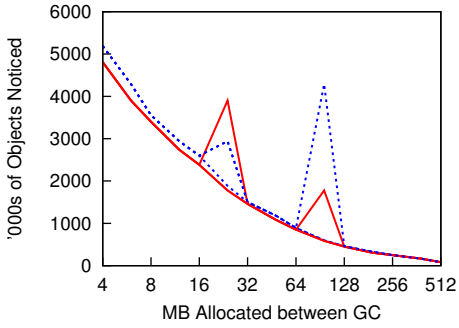
(g) jython



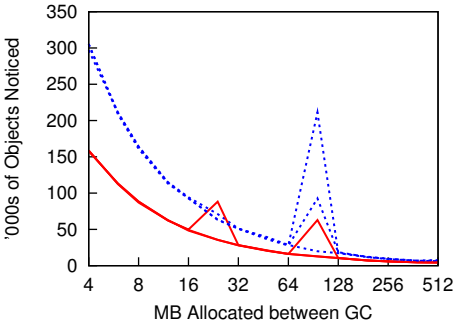
(h) luindex



(i) lusearch

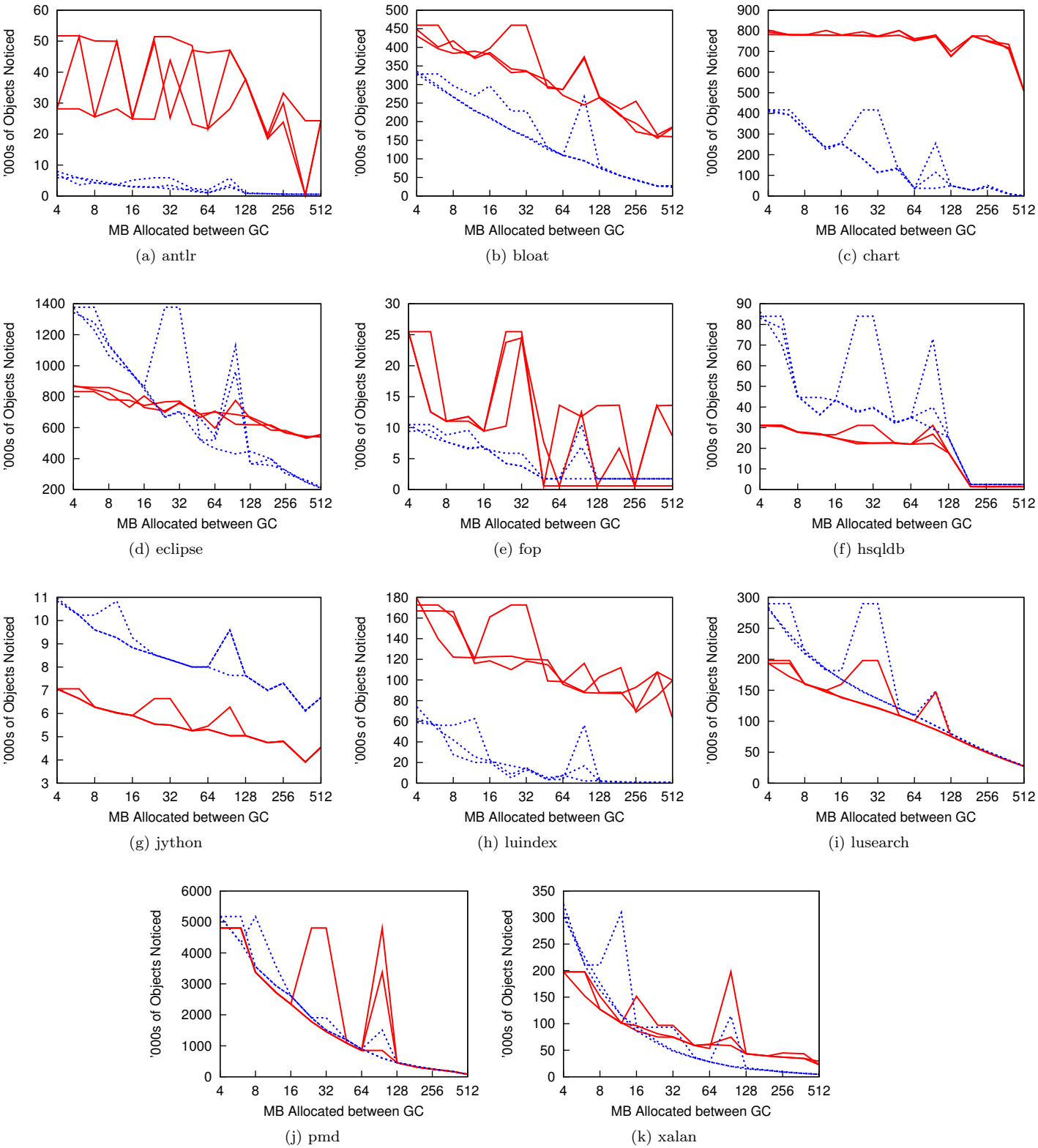


(j) pmd



(k) xalan

Figure 4: Number of objects noticed by mutator write barriers with finalization off.
__ Deletion Barrier ... Insertion Barrier



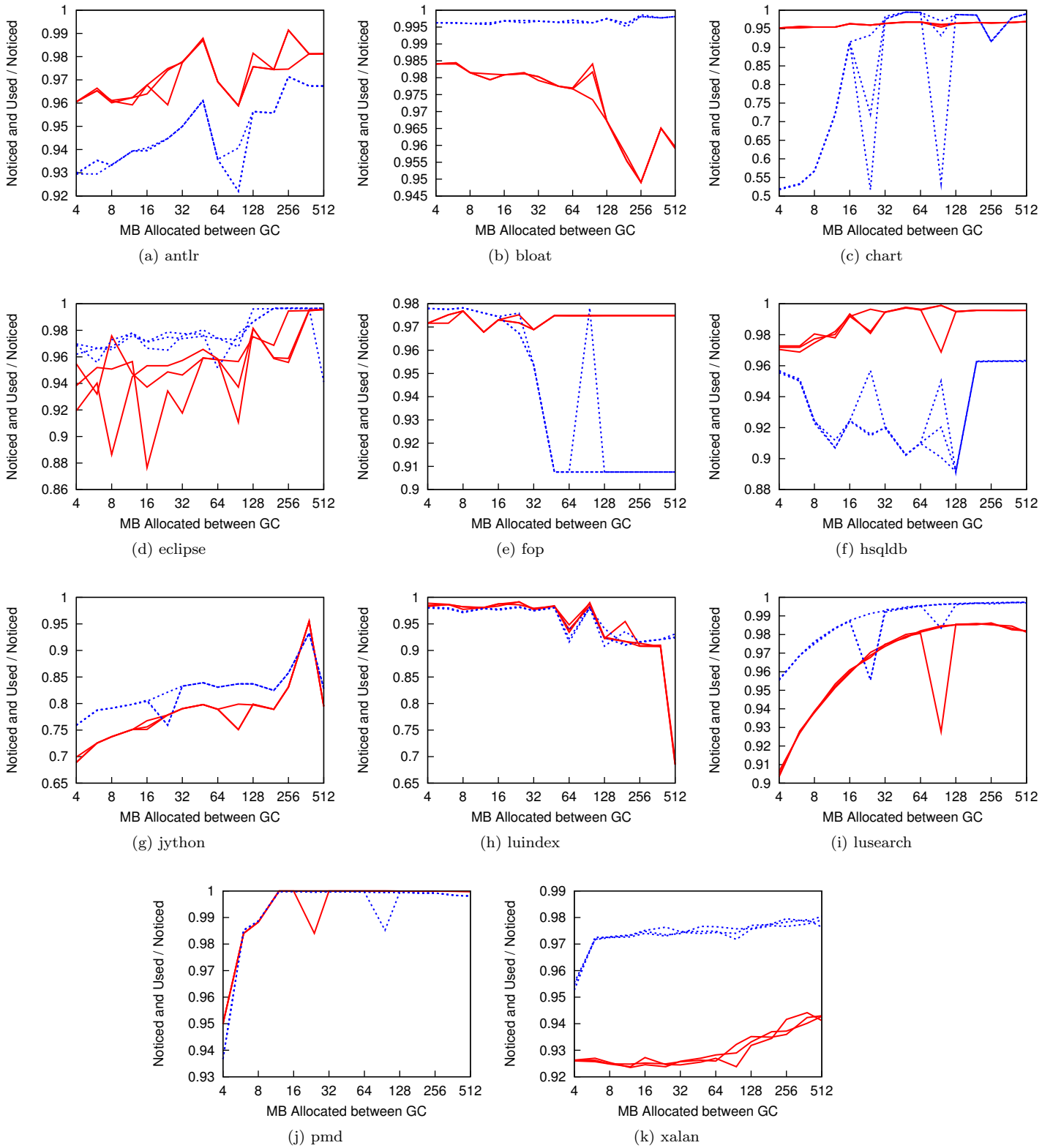
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 5: Number of objects noticed by mutator write barriers with finalization on.
__ Deletion Barrier ... Insertion Barrier



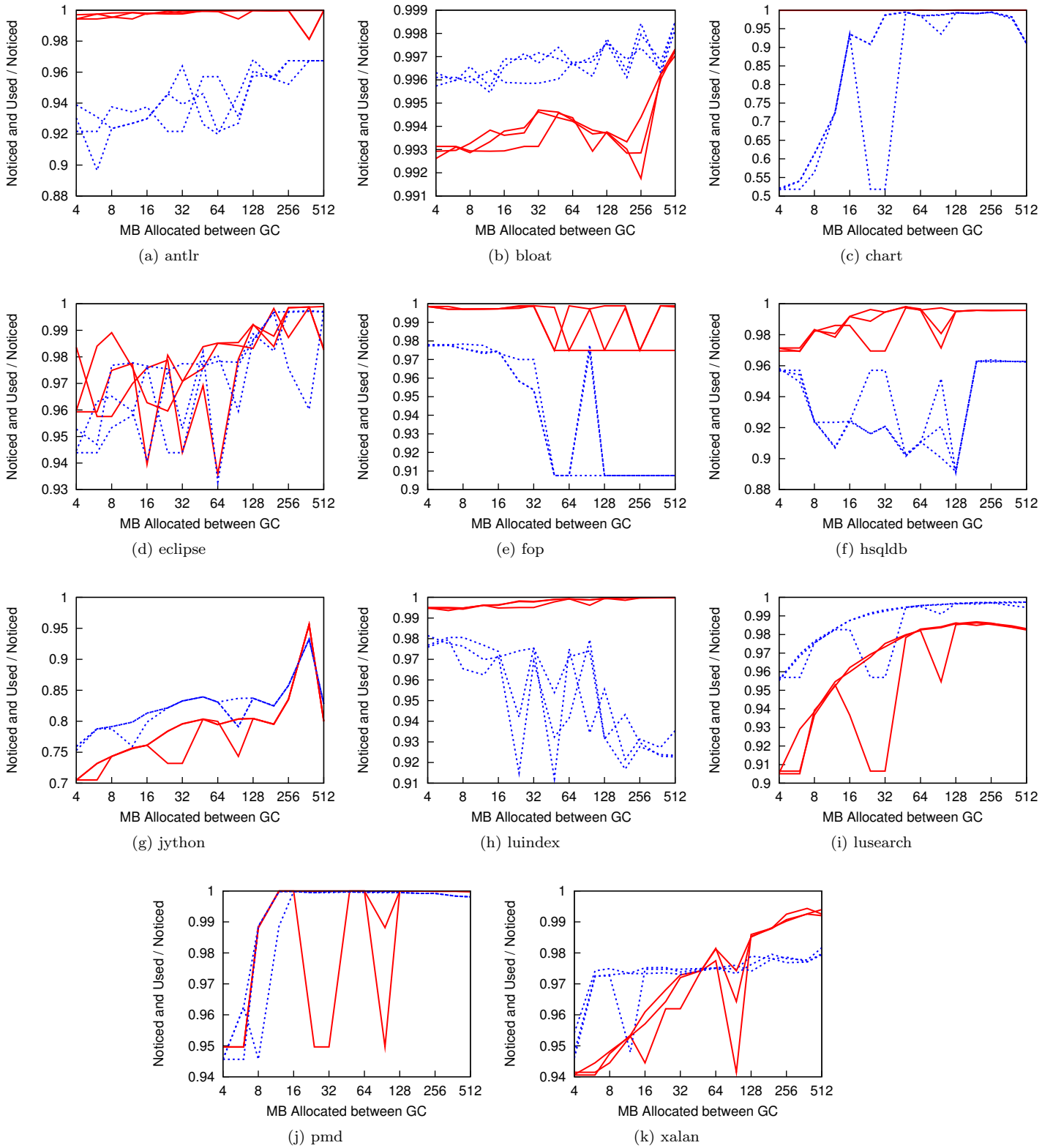
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 6: Number of objects noticed & used by mutators / number of objects noticed by mutator write barrier with finalization off. __ Deletion Barrier ... Insertion Barrier



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

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

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

(j) pmd (k) xalan

Figure 7: Number of objects noticed & used by mutators / number of objects noticed by mutator write barrier with finalization on. __ Deletion Barrier ... Insertion Barrier

barriers in pmd. 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. Of course, this discussion does not consider the transitive closure of the heap and its full impact on the locality of the deletion barrier. 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 object is deleted only once whereas a reference to an object may be inserted many times. Whether the subsequent insertions shade the object depends on its colour which is reset after each scheduled collection.

To be complete, Appendix B highlights the reuse distance between notice-then-use and use-then-notice.

To summarise our key findings:

- Deletion barriers tend to have very short (less than a KB) reuse distance between noticing an object and using the object.
- Varying the collection frequency does effect the reuse distance.
- The deletion barriers appear to be less sensitive to collection frequency.

6.3.4 Validation

Hardware performance counters were used to validate if reuse distance curves are a good indicator of likely cache behaviour. A clean checkout of Jikes RVM was made and a patch [Hellyer, 2010] developed to utilise the Libpfm4 performance event library [LibPfm4, 2010] and the Linux kernel perf_events subsystem. Using this framework it was possible to determine the number of cache misses incurred by the mutator.

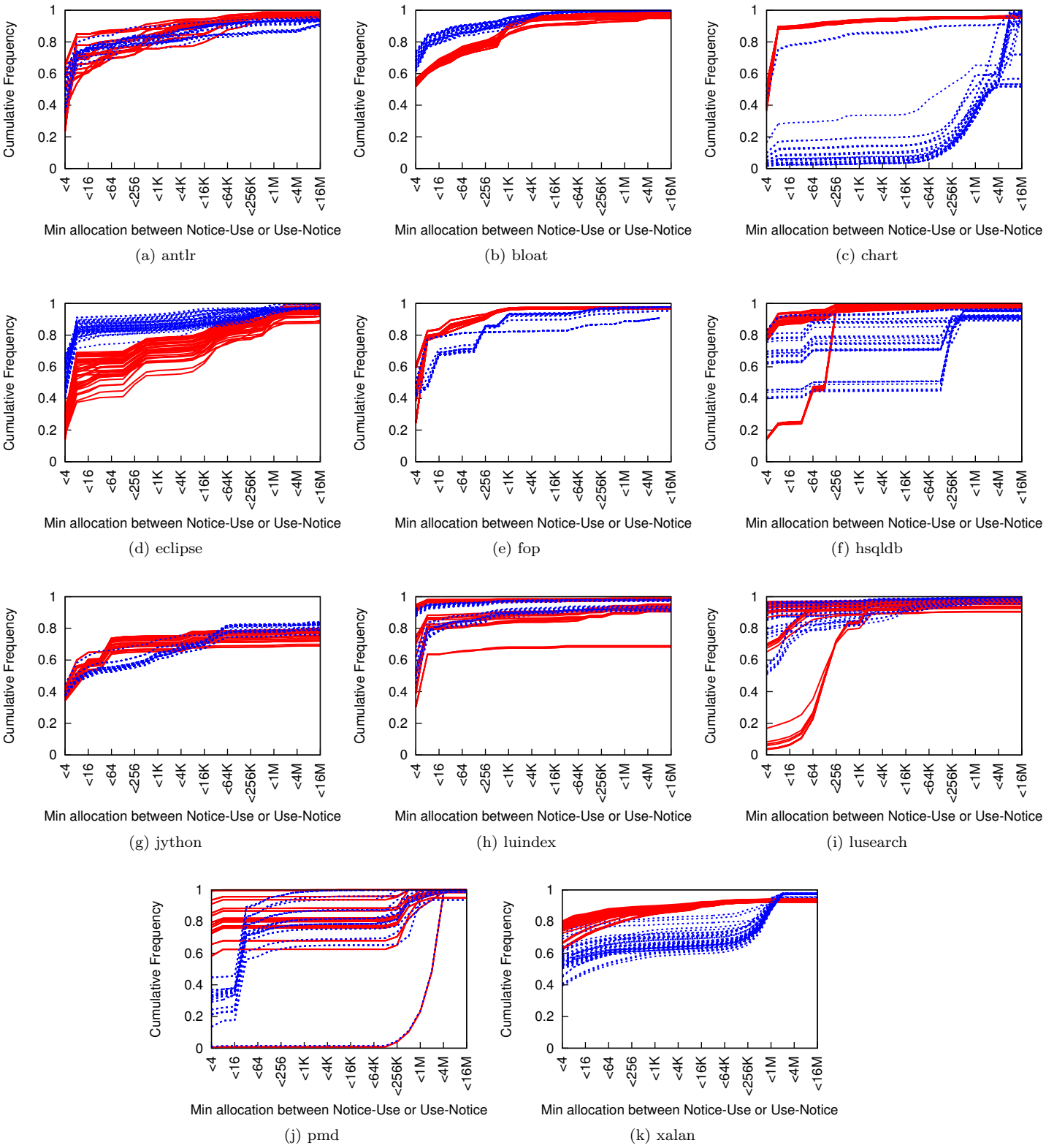
Algorithm 2 shows the mutator code used in the validation framework. Algorithm 2 is similar to the black code parts of Algorithm 1. Objects are implicitly used and loaded into the cache upon method dispatch, locking, hashing etc. The mutator write barrier affects the cache locality by requiring the additional load of either `old` or `new` and the conditional setting of a bit in the object header.

By setting `ref` on line 11 of Algorithm 2 to `new` it was possible to measure the mutator cache locality effects of the additional load by an insertion barrier. Setting `ref` to `old` mimicked the locality of a deletion barrier. By omitting line 4 it was possible to measure the locality effect of having no mutator write barrier.

It is not possible to directly exclude the effects of class loading or compiling via our cache locality framework. To better measure the locality effects of each different mutator write barrier we ran each benchmark for 10 iterations in a single Jikes RVM invocation.

For example, for the fop benchmark on an Intel Core 2 Duo (T9550) with 32KB of level 1 data cache and 6MB of shared level 2 cache, our results show a trend towards: (i) a mutator with no write barrier (omitting line 4) has the lowest number of L1D and L2 cache misses, (ii) a deletion mutator barrier incurs slightly more cache misses than having no mutator write barrier and (iii) the insertion barrier incurring more cache misses than the deletion barrier. This trend supports our conclusions from Section 6.3.3 that the deletion barrier might exhibit better cache locality than an insertion barrier.

Cache miss results are inherently noisy but some of the measured results were statistically significant. Table 3 shows the cache misses results for the fop benchmark and the results of a running a one-way ANOVA with a Tukey post hoc test. The one-way ANOVA compares the distribution of cache misses for each barrier to determine if the distributions are drawn from independent samples (i.e. are the differences in cache misses due to chance, or do the differences in the data really reflect that each barrier has a different locality effect). Once the one-way ANOVA determines that the distributions are different, the Tukey post-hoc tests examines *where* the difference in distributions occur. Take the measured L1D misses for fop with a 512MB collection window: the mean L1D misses for a mutator with no write barrier was 238915677, the mean L1D misses for an insertion barrier was 244064819. The # sign next to these results is because the Tukey test showed the results to be statistically different. The associated 'p' value (0.043) is the probability (out of a total of 1) that the results are infact not statistically different. In this case there is a 4.3% chance that there really is no locality difference between having a mutator with no write barrier and an insertion



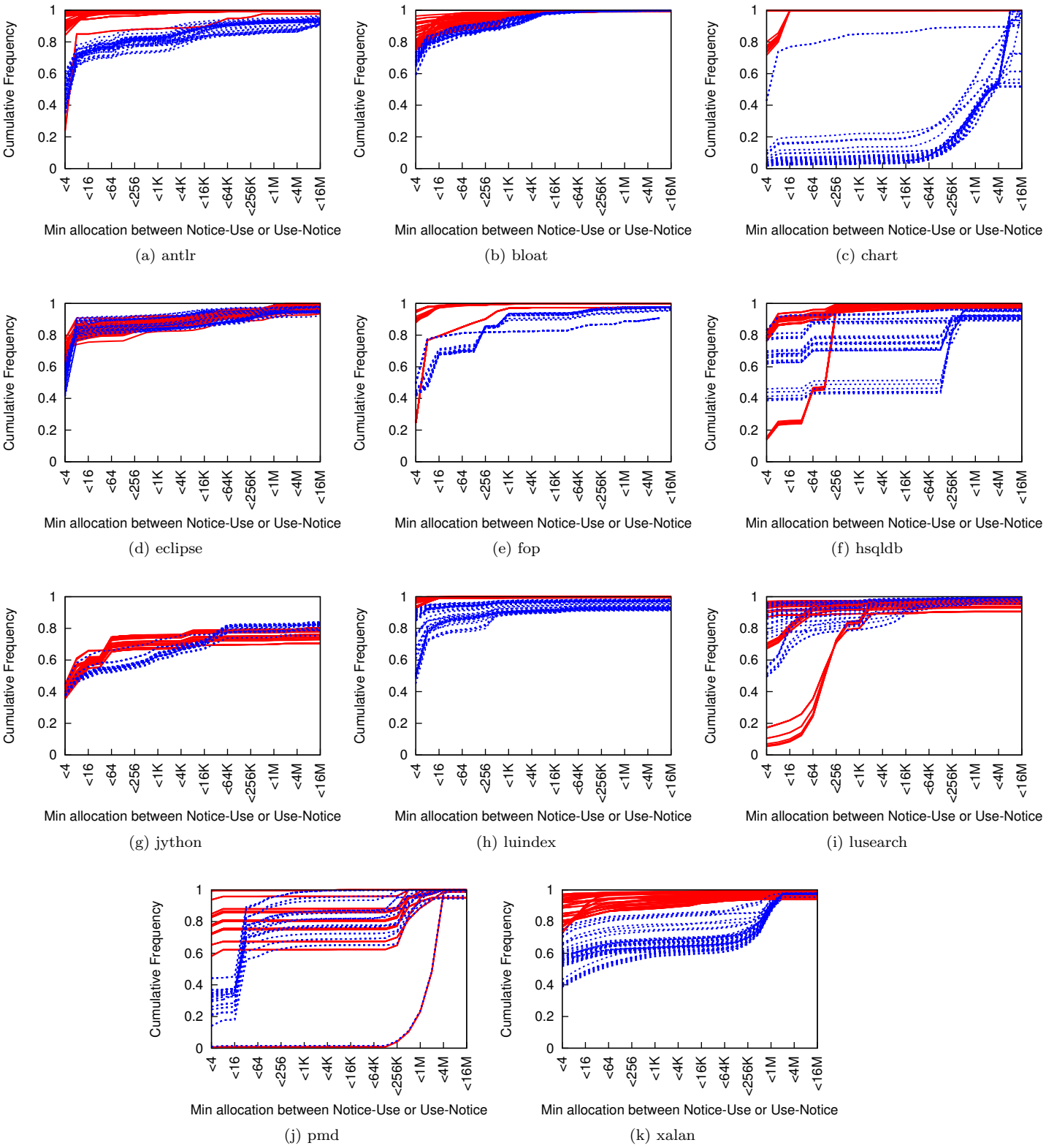
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 8: 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 with finalization off.
__ Deletion Barrier ... Insertion Barrier



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

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

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

(j) pmd (k) xalan

Figure 9: 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 with finalization on.
__ Deletion Barrier ... Insertion Barrier

Algorithm 2: Pseudocode for cache locality validation.

```
1  Write(src, fld, new):
2    if (referenceType(new))
3      old = *fld
4      markNoticed(src, old, new) // omit for no barrier
5    ...// write the new value
6
7  Read(src, field):
8    ...//return the value of the field
9
10 markNoticed(src, old, new):
11   ref = ...// select src, old or new
12   if (ref = null) return
13   if (gcInProgress()) return
14   atomic
15     if (ref.noticed = FALSE)
16       ref.noticed = TRUE
17
18 New(): // allocate black
19   ref = allocate()
20   ref.noticed = TRUE
21
22 clearNotice(ref)
23   ref.noticed = FALSE
```

Collection Frequency	Barrier	Average L1D Misses	σ	Average L2 Misses	σ
512MB	None	238915677 #	2149580	828805 *	1074
	Deletion	242555429	2478560	827482 @	528
	Insertion	244064819 #	1001000	841135 @*	4089
4MB	None	284316968 &	1563000	3594498	15627
	Deletion	284150514 !	1369650	2560978	28326
	Insertion	288828633 !&	7306600	3607576	3683

Significance differences: # p = 0.043, @ p = 0.001, * p = 0.002, ! p = 0.010, & p = 0.011

Table 3: L1D and L2 cache misses for fop with finalization on

write barrier. The smaller the reported p value the less chance of stating the results are significantly different when infact they are not. A common statistical threshold for significant results is $p \leq 0.05$, giving only a 5% chance that a significant result is really insignificant.

The measured cache misses for the hsqldb benchmark show a similar trend to fop. However, the results were not statistically significant, showing larger variation in values.

To summarise our key findings:

- The number of cache misses is benchmark, finalization, mutator barrier and hardware specific.
- For our measured benchmarks (fop and hsqldb) a mutator with no write barrier has the best cache locality and a mutator with an insertion write barrier the worse.
- The measured cache locality appears to match predictions by our reuse distances graphs.

7 Further work

These results raise several questions that deserve future exploration:

- 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.

8 Conclusion

We have compared the object reuse and 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 with finalization on. With finalization off it is benchmark dependent as to which barrier `notices` more objects. For some benchmarks the amount of work generated largely depends on the selected barrier and the heap size. The ratio between objects that the application will load into its cache (`used`) and objects mutator write barriers touch (`noticed`) is greater than 0.9 for both barriers and most benchmarks. This suggests that many objects that mutator write barriers might need to write to will be loaded into cache by the application anyway. The time between noticing and using an object can be much lower with a deletion barrier than an insertion barrier, suggesting that deletion barriers may lead to better mutator cache performance than has hitherto been expected. We validate these results by measuring actual cache misses for one implementation of the barriers.

Acknowledgements

We are grateful for the support of the EPSRC through grants EP/H026975/1 and EP/F06523X/1, the National Science Foundation through grants CCF-0811691 and CCF-0702240, and grants from Microsoft, Intel, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors. We thank IBM Research for making the Jikes RVM system available.

References

Santosh Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *International Conference on Parallel Processing*, pages 243–246, University Park, Pennsylvania, USA, August 1987. Pennsylvania State University Press. Also technical report CSRD 620, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development.

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

Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. URL `http://home.pipeline.com/~hbaker1/RealTimeGC.ps.gz`. Also AI Laboratory Working Paper 139, 1977.

Stephen Blackburn, Robin Garner, Kathryn S. McKinley, Amer Diwan, Samuel Z. Guyer, Antony Hosking, J. Eliot B. Moss, and Darko Stefanović. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Twenty-First ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 41(10), Portland, OR, USA, October 2006.

Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In David F. Bacon and Amer Diwan, editors, *Proceedings of the Fourth International Symposium on Memory Management*, pages 143–151, Vancouver, Canada, October 2004. ACM Press. URL `http://www.research.ibm.com/ismm04/program.html`.

Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In Hans-J. Boehm and David Detlefs, editors, *Proceedings of the Third International Symposium on Memory Management (June, 2002)*, ACM SIGPLAN Notices 38(2 supplement), pages 175–184, Berlin, Germany, February 2003. URL `http://www.hpl.hp.com/personal/Hans_Boehm/ismm/`.

Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 37(5), pages 153–164, Berlin, Germany, June 2002. doi: `10.1145/512529.512548`. URL `http://www.cs.ukc.ac.uk/pubs/2002/1363`.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Edinburgh, May 2004. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1317436.

Hans-Juergen Boehm. Reducing garbage collector cache misses. In Craig Chambers and Antony L. Hosking, editors, *Proceedings of the Second International Symposium on Memory Management*, ACM SIGPLAN Notices 36(1), pages 59–64, Minneapolis, MN, October 2000. ISBN 1-58113-263-8.

Hans-Juergen Boehm. Destructors, finalizers, and synchronization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), New Orleans, LA, USA, January 2003.

Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991. doi: 10.1145/113445.113459.

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

Wen-Ke Chen, Sanjay Bhansali, Trishul M. Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In Schwartzbach and Ball [2006], pages 332–340. doi: 10.1145/1133981.1134021.

Chen-Yong Cher, Antony L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In Shubu Mukherjee and Kathryn S. McKinley, editors, *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices 39(11), pages 199–210, Boston, MA, USA, October 2004. doi: 10.1145/1024393.1024417.

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 34(5), pages 1–12, Atlanta, GA, USA, May 1999. doi: 10.1145/301618.301633.

David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Proceedings of the Second Java Virtual Machine Research and Technology Symposium*, San Francisco, CA, USA, August 2002. USENIX. URL http://research.sun.com/jtech/pubs/02-clog.pdf.

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. *Communications of the ACM*, 21(11):965–975, November 1978. doi: 10.1145/359642.359655.

Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In Greg Morrisett and Mooly Sagiv, editors, *Proceedings of the Sixth International Symposium on Memory Management*, pages 43–54, Montréal, Canada, October 2007. ACM Press. doi: 10.1145/1296907.1296915.

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997. ISBN 0-201-63451-1. URL http://www.aw.com/cp/gosling-joy-etal.html.

David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

Laurence Hellyer. Jikes RVM alternative performance counter implementation, 2010. URL http://jira.codehaus.org/browse/RVM-885.

Laurence Hellyer, Richard Jones, and Antony L. Hosking. The locality of concurrent write barriers. In Jan Vitek and Doug Lea, editors, *Proceedings of the 2010 International Symposium on Memory Management*, Toronto, Canada, June 2010. ACM Press.

Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1993. URL ftp://self.stanford.edu/pub/papers/write-barrier.ps.Z.

Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 27(10), pages 92–109, Vancouver, Canada, October 1992. URL `ftp://ftp.cs.umass.edu/pub/osl/papers/oopsla92.ps.Z`.

Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ACM SIGOPS Operating Systems Review 27(5), pages 106–119, Asheville, NC, USA, December 1993. doi: `10.1145/168619.168628`. URL `ftp://ftp.cs.umass.edu/pub/osl/papers/sosp93.ps.Z`.

Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. ISBN 0–471–94148–4. URL `http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html`. With a chapter on Distributed Garbage Collection by R. Lins.

H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977. doi: `10.1109/SFCS.1977.5`.

Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Illinois, USA, 17–19 September 1992. Springer.

LibPfm4. Perfmon2, the hardware-based performance monitoring interface for linux, 2010. URL `http://perfmon2.sourceforge.net/`.

David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Proceedings of the ACM Conference on Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, USA, August 1984.

Gene Novark, Trevor Strohman, and Emery D. Berger. Custom object layout for garbage-collected languages. Technical report, University of Massachusetts at Amherst, 2006. URL `http://www.cs.umass.edu/~emery/pubs/06-06.pdf`. NEPLS, March, 2006.

Pekka P. Pirinen. Barrier techniques for incremental tracing. In Simon L. Peyton Jones and Richard Jones, editors, *Proceedings of the First International Symposium on Memory Management (October, 1998)*, ACM SIGPLAN Notices 34(3), pages 20–25, Vancouver, Canada, March 1999. ISBN 1-58113-114-3.

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 43(6), pages 33–44, Tucson, AZ, USA, June 2008. doi: `10.1145/1379022.1375587`.

Michael I. Schwartzbach and Thomas Ball, editors. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 41(6), Ottawa, Canada, June 2006.

Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Conference Record of the Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 37(1), Portland, OR, USA, January 2002.

David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In Erez Petrank and J. Eliot B. Moss, editors, *Proceedings of the Fifth International Symposium on Memory Management*, pages 52–63, Ottawa, Canada, June 2006. ACM Press.

Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. URL `ftp://publications.ai.mit.edu/ai-publications/1994/AITR-1417.ps.Z`. Bachelor of Science thesis.

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

Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 34(10), pages 370–381, Denver, CO, USA, October 1999. doi: `10.1145/320384.320425`.

V. Stenning. On-the-fly garbage collection. Unpublished notes, cited by Gries [1977], 1976.

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.

The DaCapo Group. The DaCapo website, 2006. URL http://cs.anu.edu.au/people/Steve.Blackburn/dacapobench.org/dacapo-graphs/index.html.

David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

Martin Vechev, David F. Bacon, Perry Cheng, and David Grove. Derivation and evaluation of concurrent collectors. In Andrew Black, editor, *Proceedings of 19th European Conference on Object-Oriented Programming, ECOOP 2005*, Lecture Notes in Computer Science, Glasgow, July 2005. Springer-Verlag.

Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In Schwartzbach and Ball [2006], pages 341–353. doi: 10.1145/1133981.1134022.

Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. URL ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps. Expanded version of the IWMM92 paper.

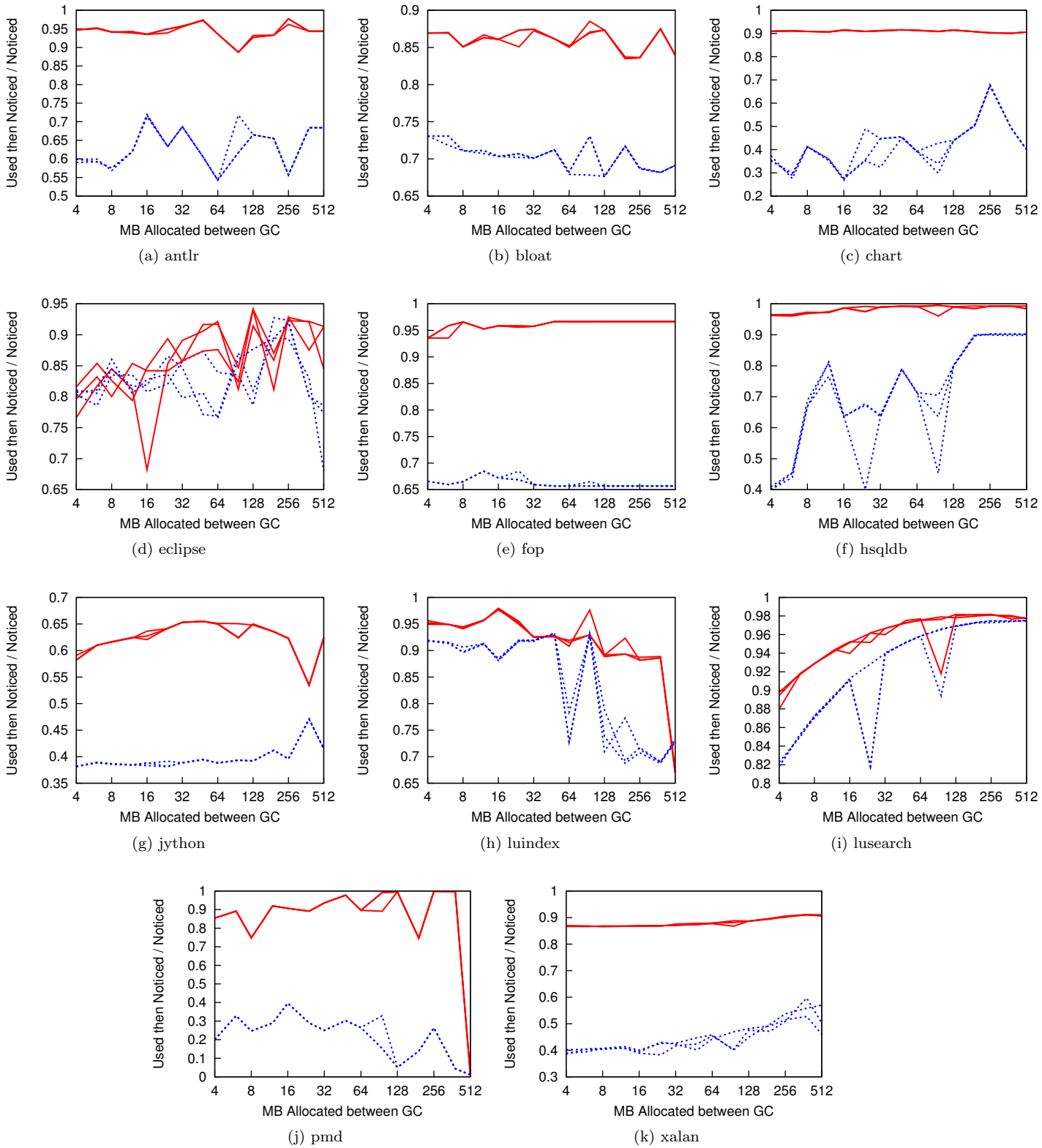
Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 24(10), pages 23–35, New Orleans, LA, USA, October 1989.

Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 26(6), pages 177–191, Toronto, Canada, June 1991. doi: 10.1145/113445.113461.

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

Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990. URL ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/CU-CS-494-90.ps.Z.

A Appendix - Reuse Order



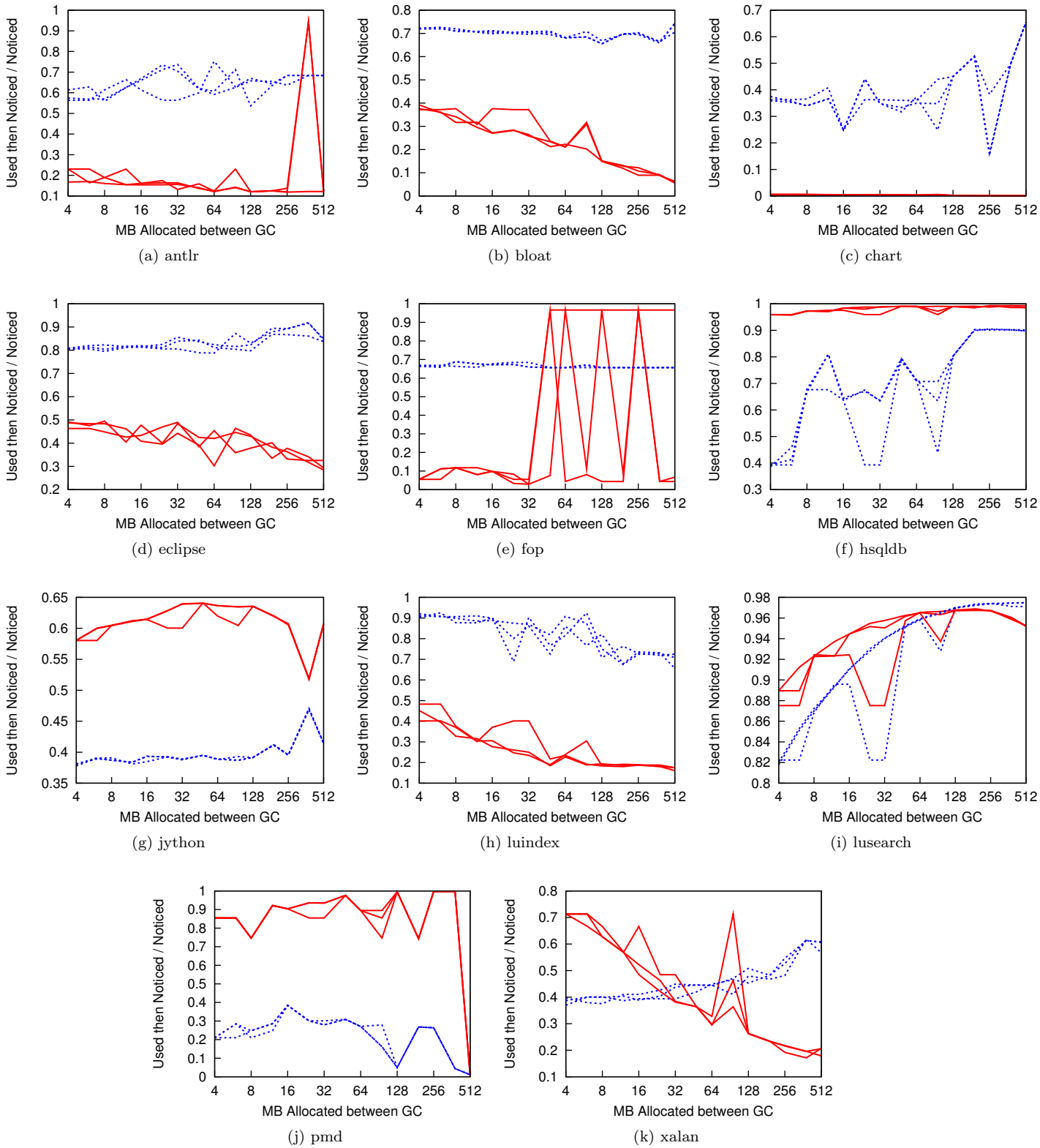
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 10: Number of objects used then noticed by mutators / number of objects noticed by mutator write barrier with finalization off. __ Deletion Barrier ... Insertion Barrier



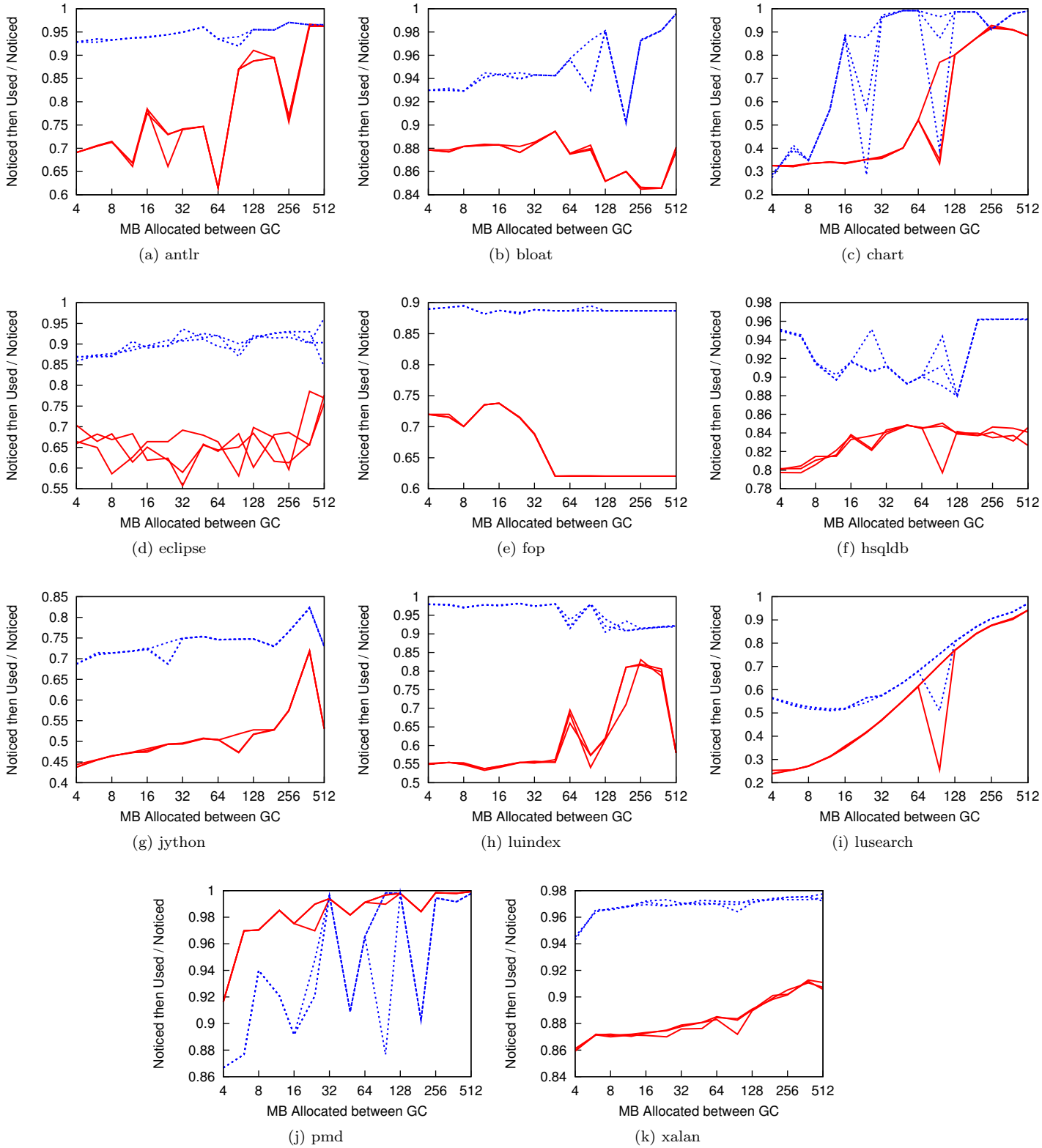
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 11: Number of objects used then noticed by mutators / number of objects noticed by mutator write barrier with finalization on. __ Deletion Barrier ... Insertion Barrier



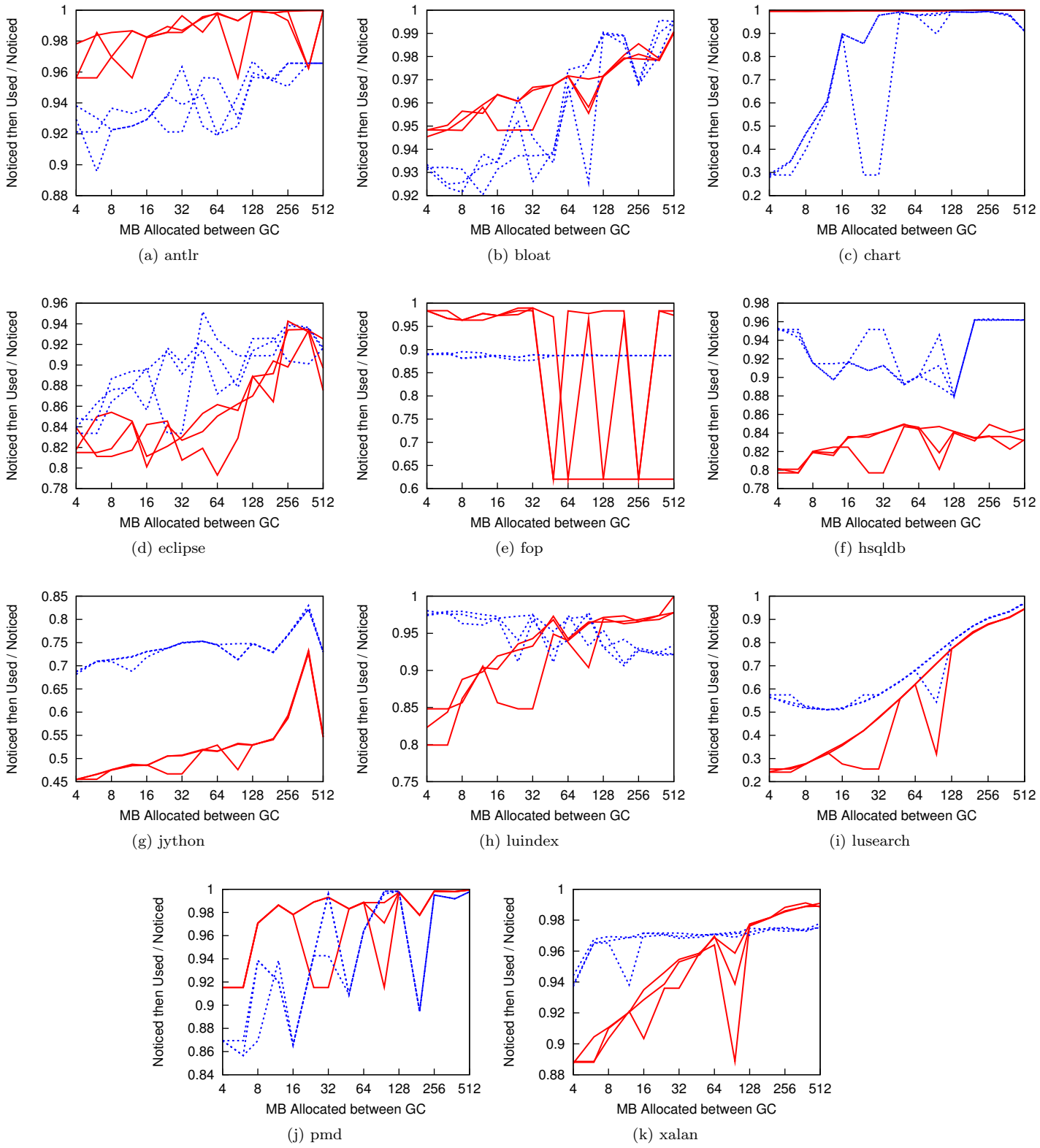
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 12: Number of objects noticed then used by mutators / number of objects noticed by mutator write barrier with finalization off. __ Deletion Barrier ... Insertion Barrier



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

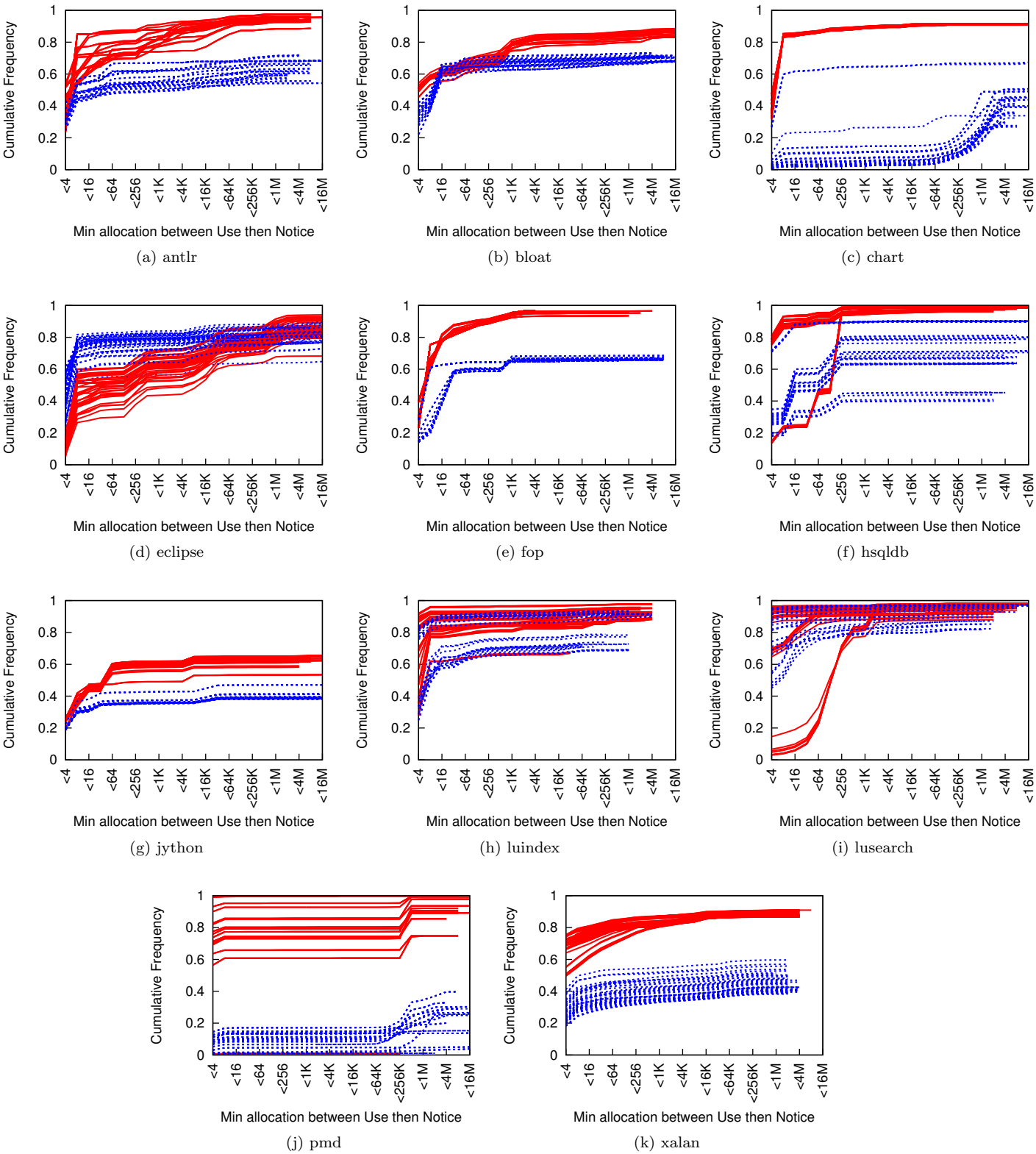
(d) eclipse (e) fop (f) hsqldb

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

(j) pmd (k) xalan

Figure 13: Number of objects noticed then used by mutators / number of objects noticed by mutator write barrier with finalization on. __ Deletion Barrier ... Insertion Barrier

B Appendix - Locality by Reuse Order



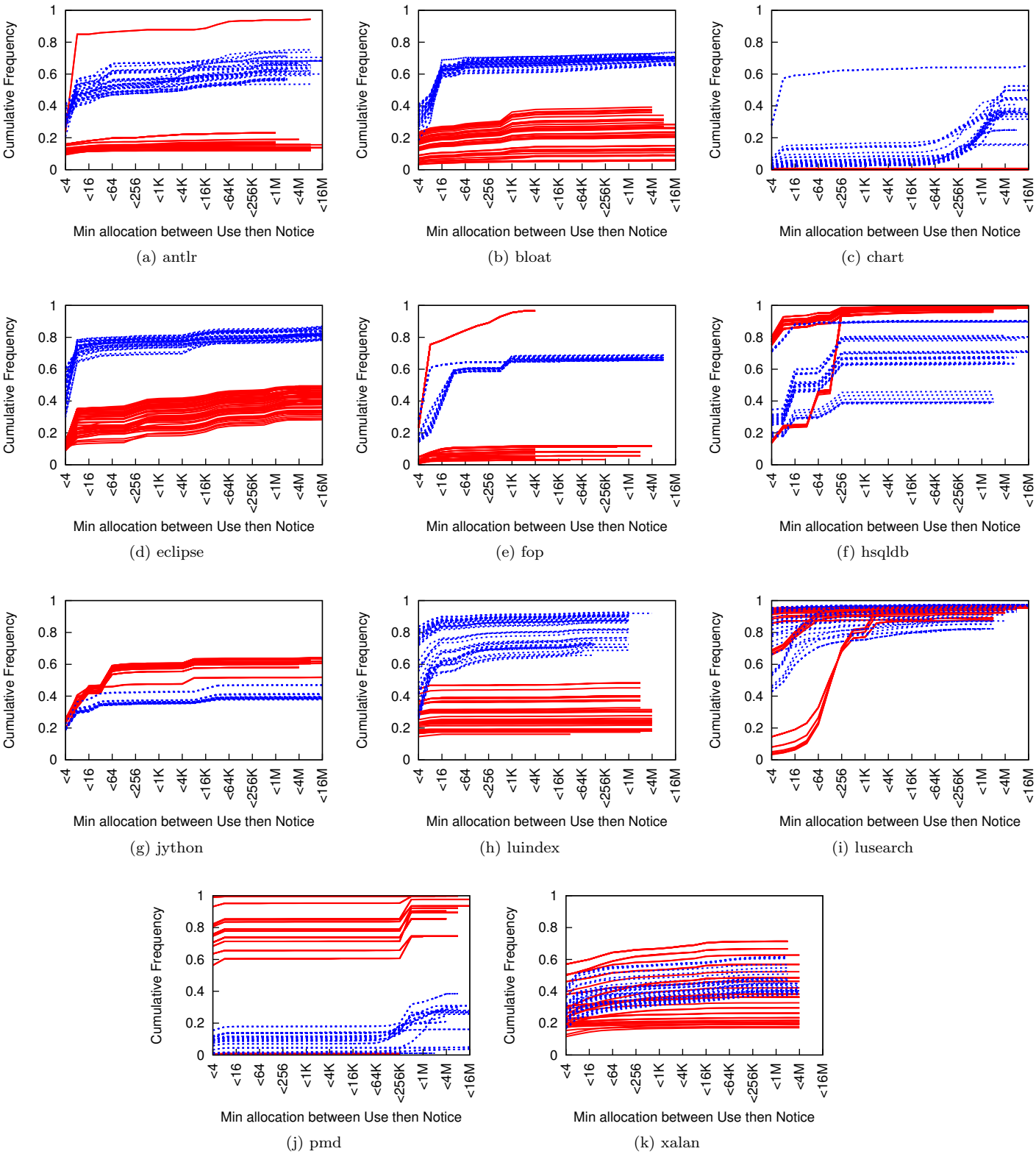
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 14: The distribution of minimum distance between Using an object and Noticing the object. A point (x, y) indicates that y% of the Used objects are Noticed by the mutator within $\pm x$ bytes of allocation. The figures show curves for each barrier and for each collection frequency with finalization off.
__ Deletion Barrier ... Insertion Barrier



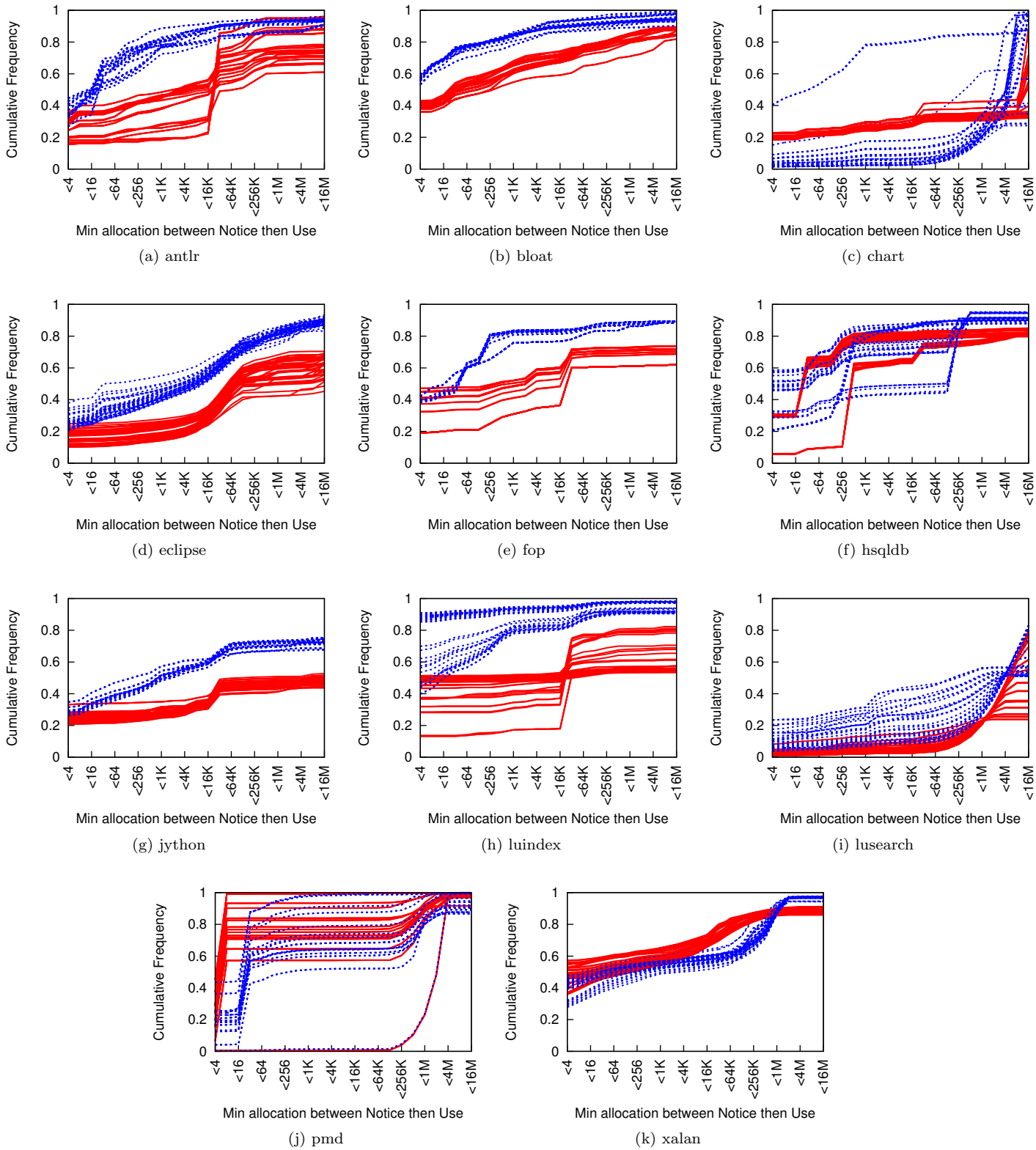
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 15: The distribution of minimum distance between Using an object and Noticing the object. A point (x, y) indicates that y% of the Used objects are Noticed by the mutator within $\pm x$ bytes of allocation. The figures show curves for each barrier and for each collection frequency with finalization on.
__ Deletion Barrier ... Insertion Barrier



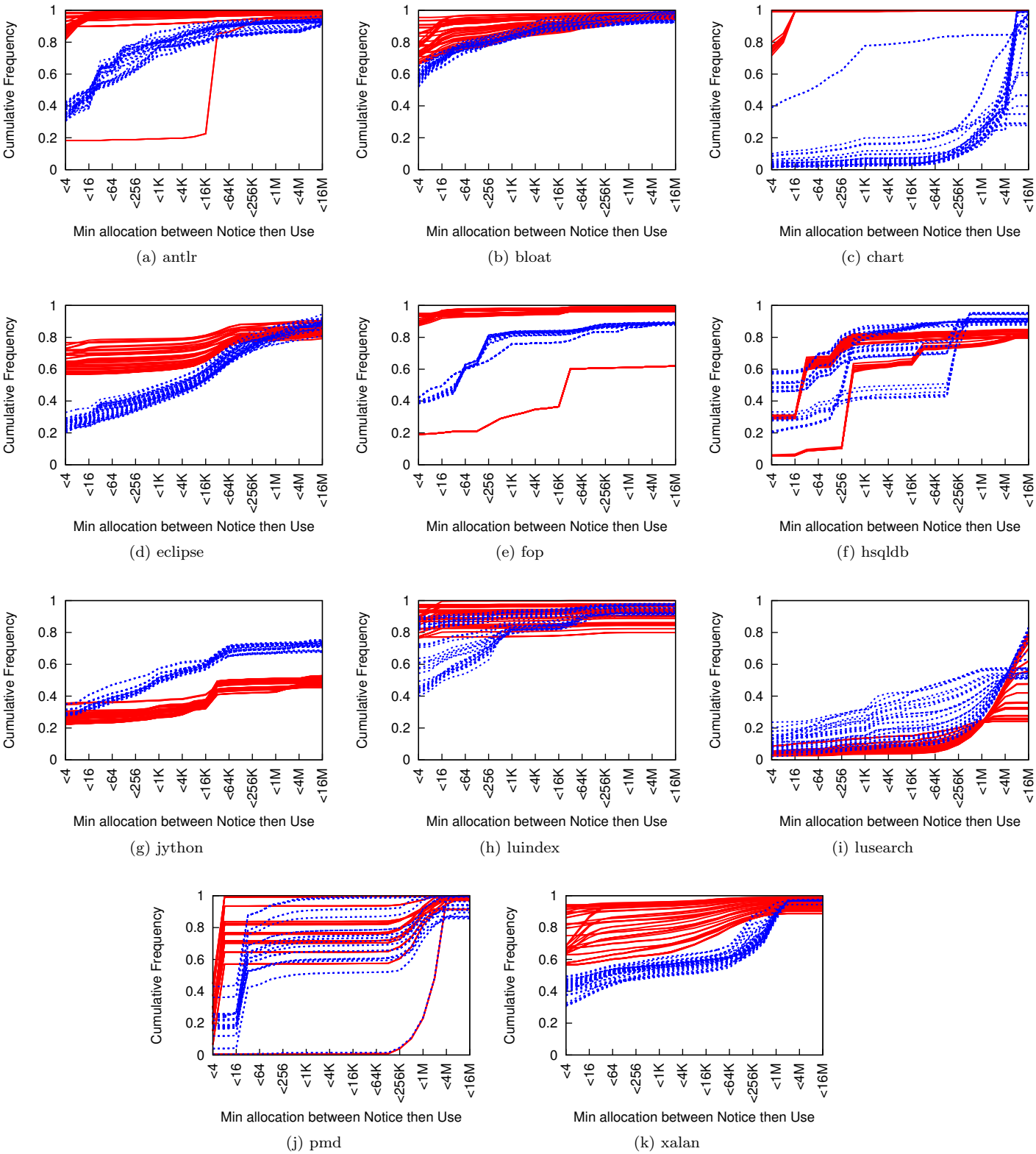
(a) antlr (b) bloat (c) chart

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

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

(j) pmd (k) xalan

Figure 16: 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 with finalization off.
__ Deletion Barrier ... Insertion Barrier



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

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

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

(j) pmd (k) xalan

Figure 17: 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 with finalization on.
__ Deletion Barrier ... Insertion Barrier