Design and Analysis of Field-Logging Write Barriers

Stephen M. Blackburn
Australian National University
Australia
steve.blackburn@anu.edu.au

Abstract

Write barriers are a fundamental mechanism that most production garbage collection algorithms depend on. They inform the collector of mutations to the object graph, enabling partial heap collections, concurrent collection, and reference counting. While in principle, write barriers remember only the pointers within the object graph that were changed and do so just once, widely-used write barriers are less precise, sacrificing temporal and/or spatial precision to performance. The idea of precisely remembering just the pointers that were changed is not new, however implementing performant field-precise barriers has proved elusive. We describe a technique for efficiently implementing field-logging barriers. We implement a number of barriers and evaluate them on a range of x86 hardware. A generational field-logging barrier performs with 0.1% to 1% mutator overhead compared to a highly tuned object-logging barrier, while a preliminary implementation of a reference counting field-logging barrier performs with around 1% to 2% overhead compared to a highly tuned object-logging reference counting barrier. These results suggest that garbage collection algorithms that require the precision of exactly remembering field mutations without sacrificing performance may now be possible, adding a new mechanism to the design toolkit available to garbage collection researchers.

CCS Concepts • **Software and its engineering → Garbage collection**; **Runtime environments**.

Keywords garbage collection, write barriers, generational garbage collection, reference counting

ACM Reference Format:
Stephen M. Blackburn. 2019. Design and Analysis of Field-Logging Write Barriers. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315573.3329981

ISMM '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6722-6/19/06.
https://doi.org/10.1145/3315573.3329981

1 Introduction

Write barriers are small code fragments that mediate runtime stores to memory, allowing a garbage collector to be informed of changes to the heap made by the mutator. They have thus become an essential part of most modern garbage collector implementations.

One motivation for write barriers is that because tracing collectors depend on global knowledge to determine liveness, unless informed of changes made to the heap by the mutator, they must perform a trace of the entire heap before they can collect any garbage [10, 16, 23]. Such algorithms are non-performant and are therefore not widely used. On the other hand, a write barrier allows generational collectors to preferentially collect just recently allocated objects by informing the collector of new objects reachable from old objects by remembering when the mutator creates a pointer from an old object to a new object [22, 28]. Write barriers also allow reference counting collectors to continuously track the creation and removal of pointers to objects [12] and concurrent collectors to ensure correctness while they concurrently collect the heap [15, 26].

Widely-used write barriers are spatially and/or temporally imprecise. Such imprecision is unhelpful. A barrier that only reports that 'something' was changed 'some time' would be of little use. Conversely, a barrier that reports precisely what reference changed at exactly the first time it was changed would be ideal. However in systems engineering, the goals of precision and performance are generally in tension, and this holds true in the design of barriers. For example, a card marking barrier [30] will imprecisely mark as dirty a 2^n-byte region of memory containing the pointer field updated by the mutator. The collector must subsequently scan the dirty regions of memory to discover relevant pointer/s within them. Similarly, an object-remembering barrier [2, 6] will imprecisely remember an object in which a pointer was updated, and the collector must subsequently scan the object to discover the relevant pointer/s within it. Both are spatially imprecise, requiring the collector to later refine the information. On the other hand, a traditional boundary barrier [6] will remember the address of an updated pointer field, but may remember the same field many times, so it is temporally imprecise. Both the card-marking and object-marking barriers avoid duplicates by setting a remembered bit/byte. In the case of the card marking, a byte is unconditionally set by

the write barrier. Subsequently re-setting the byte adds no further work to the collector. On the other hand, an object-remembering barrier conditionally sets a bit in the object's header, only remembering the object if it has not already done so since the last collection. The challenge addressed by this paper is to be able to remember with temporal and spatial precision: remembering a mutated field only the first time it is changed, which implies a check against a per-field remembered bit.

Levanoni and Petrank [21] raised the possibility of a field-logging barrier. They described a coalescing reference counting barrier that worked by remembering only the first change to a given pointer in a given mutator epoch, whereas previous reference counting algorithms generated an increment and decrement operation for every pointer update. However, although the algoritm was described abstractly in terms of field-logging, in practice Levanoni and Petrank used an object-logging barrier which gave the same behavior but at the cost of spatial precision.

Our insight is that a field-logging barrier can be implemented using a very similar approach to an object-logging barrier, a widely used low-overhead write barrier [31]. The field barrier can use bits in the object header to record whether the field has been remembered, just as an object barrier does. If more bits are required, additional bytes before the object can be used to store the necessary state. When the field location is statically known, such as a field access in a statically-typed language, the instruction sequence for testing and setting a field-remembering bit is identical to that for an object barrier. Many Java objects have few, or no reference fields, so the space overhead for recording the requisite state is just 3.3% in practice (Table 5).

We implement two variations of a field-logging barrier in Jikes RVM [1] and evaluate them on six x86 microarchitectures. We show that their overhead is very low compared to the object barrier and slot barrier, which have been highly tuned for well over a decade, demonstrating that the idea is feasible. We present a detailed evaluation and analysis of the barrier, providing the memory management community with deeper insights into how this new barrier works.

The primary contributions of this paper are: (*i*) the design of a practical field-logging write barrier, (*ii*) implementation of two instances of a field-logging write barrier, (*iii*) a detailed evaluation of the barriers on six x86 microarchitectures.

2 Background and Related Work

Generally, a write barrier only tracks reference updates (*pointer* stores), however some collectors, such as Sapphire [20], use write barriers to track non-pointer stores too. In this work we focus on write barriers on reference updates (only), as used in most production garbage collection algorithms today. We develop the idea in a Java virtual machine, but the design is applicable to other languages, particularly those that are statically typed (which some of our optimizations exploit).

Java has two important bytecodes for reference updates, putfield and aastore, which respectively update a pointer field within a scalar (non-array) object, and a field within a reference array. There exist other paths by which references within heap objects get updated in a JVM, including compare and swap operations and array copy operations. We address each of those cases in our implementation, but they are fairly uninteresting engineering problems, so we focus our discussion on the putfield and aastore bytecodes which dominate all performance considerations for most workloads.

Generational and Inter-Region Write Barriers Generational collectors [22, 28] preferentially collect recently allocated objects. They are able to do this without scanning the whole heap by using a write barrier that maintains a *remembered set* of pointers from the old generation to the newly allocated objects. The write barrier intercepts each update of a heap reference and checks whether the reference points from an old object to a young object. More generally, algorithms that selectively collect regions of the heap use write barriers to track inter-region pointers and remember them in remembered sets which identify all inter-region pointers into any given region.

Multiple strategies have been used for remembering inter-region pointers. The most simple is to test each pointer store and if it points from old to young, remember the address of the written field. Wilson and Moher [30] introduced the idea of card marking, whereby the barrier unconditionally marks a *card* indicating the 2^n-byte region of memory in which the updated pointer field resides. The collector must then scan a card table which is proportional to the size of the heap to discover recently stored pointers and determine whether they point to new objects. The card marking barrier was subsequently refined [3, 11, 17, 18] and remains widely used, despite significant pathologies [14], and its inherent lack of scalability (the work of card scanning is proportional to heap size, not nursery size).

In this paper we focus on the object-logging write barrier because it is a well-known high-performing barrier [2, 31], has design similarities to the field barrier, and has similarities to other barriers we consider, including a snapshot-at-the-beginning (SATB) barrier [32] and a high performance reference counting barrier [21].

The object-logging barrier has some similarity to the card marking barrier. It remembers objects (rather than 2^n-byte cards) that contain updated pointer fields. It does so by adding the address of the object to a buffer for processing by the collector, and marking a bit in the object header to indicate that it has been remembered. The mark of the object header bit usually requires a compare-and-swap (CAS) because other threads may be modifying the same object's

header. The garbage collector processes all remembered objects, scanning them for relevant pointers, and resetting the bit in their header.

The object barrier avoids the pathology of card marking described by Dice [14] because the remembered bit is per-object, not in a side-structure, and it is only remembered once per mutator cycle. It also avoids the scalability problem of the card marking barrier because the collector's work is proportional to the number of updated objects rather than the size of the heap.

We are unaware of any prior attempt to implement a field-logging generational barrier.

Concurrent Collector Barriers Concurrent garbage collectors use barriers to ensure that the mutator and collector cooperate [15, 26, 27, 32]. Among the barriers used by concurrent collectors, two are directly relevant to this work.

The first is the snapshot-at-the-beginning (SATB) barrier by Yuasa [32]. Conceptually, this barrier creates a snapshot of the heap as it was at the beginning of a mutator epoch. It does this by remembering (preserving) any pointer field the mutator overwrites immediately before the pointer is overwritten. The SATB barrier can be implemented with an object-remembering barrier that conservatively remembers the previous values of all reference fields of any object which has a pointer field updated. The same algorithm could more precisely perform the SATB work by using a field-logging barrier that just remembered the fields that were updated.

The second is the class of algorithms that use *pointer coloring* to record state within a pointer [27]. This can be done either with low bits, which must be checked before use, or with high bits which can exploit the TLB, mapping multiple pointers to the same physical pages (note that masking of these bits is still necessary in order to ensure correct behavior for pointer equality etc.). Most pointer coloring barriers require a simple read barrier. Yang et al. [31] performed a detailed analysis of conditional read barriers and found that across a wide range of benchmarks the overhead of the two-instruction (`TEST`, `JNE`) barrier was 10.3%, significantly higher than the overheads we observe here.

Reference Counting Write Barriers Write barriers are fundamental to reference counting because they are used to generate increments and decrements when references are created and overwritten (respectively).

Levanoni and Petrank [21] use a barrier that motivates this work. Their reference counting barrier happens to be very similar to the object-remembering SATB barrier but additionally remembers the address of the updated object. The first time a pointer field within an object is updated, the barrier will remember all of the pointer fields in the object (before the new update is installed). This captures a before-image of the object. The barrier also remembers the address of the updated object, so at collection time, the collector can establish an after-image of the object by examining the pointers in the object. The collector then applies an increment to each object referenced by the after-image, and a decrement for each object referenced by before-image. A field-logging barrier would increase the spatial precision of this barrier. In particular, it would ensure that the collector workload was proportional to pointer field updates, avoiding pathologies associated with large, sparse reference arrays, which otherwise generate unnecessary collector and mutator work.

Hardware Support There have been numerous attempts to provide hardware support for garbage collection. The (software) bit-stealing barrier evaluated by Yang et al. [31] and the LVB barrier used by Tene et al. [27] might benefit from hardware that offered instructions that mask low-order bits. Such instructions have been available on past architectures such as the SPARC, but to the best of our knowledge are not supported by any mainstream current hardware. If such an instruction were available, a per-reference field logged bit could be stored in the low order bits of each pointer. The write barrier would check the bit on the to-be-overwritten value and conditionally log the field, before unconditionally storing the pointer with the bit set. Regular pointer reads would mask the bit, avoiding the need for a read barrier in the common case.

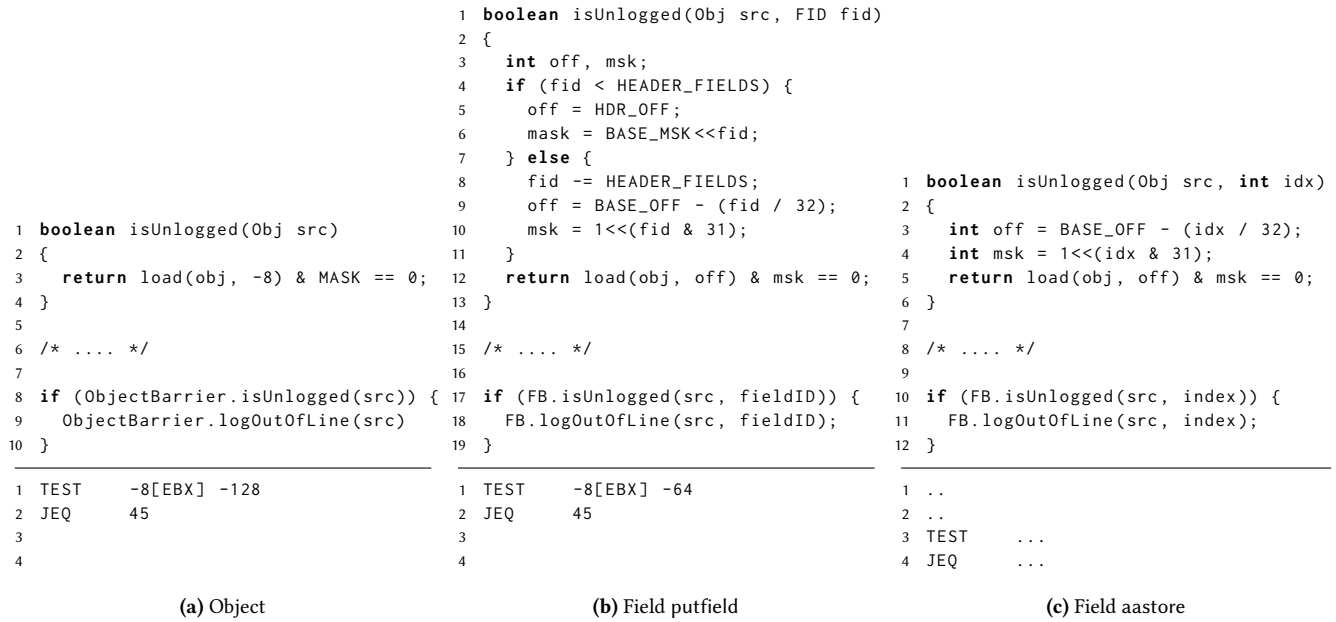
Barrier Performance There are a number of studies of barrier performance in the literature [6, 7, 19, 31, 34]. We build upon the work of Yang et al. [31], using the same virtual machine, and where appropriate follow similar methodologies. We focus on the same object logging and boundary barriers included in their study.

Barrier Elision Given sufficient information, a compiler may be able to elide barriers [13, 29, 33]. The semantics (and thus correctness) of barrier elision are dependent on the intended semantics of the barrier. Barrier elision is orthogonal to this work, but relevant to the broader objective of minimizing barrier overhead.

3 Field-Logging Barriers

Our design of field-logging barriers rests on the following insights: (*i*) in most statically typed languages, the identity of a field is statically known, allowing aggressive optimization; (*ii*) empirically, frequently allocated scalar types are small [4], so have few reference fields; (*iii*) the semantics of an aastore include an array bounds check, potentially absorbing some of the cost of a field barrier, such as access to the array header. These insights led to the idea of using a vector of logging bits prepended to an object containing reference fields.

Basic Design The basic design is that each object containing reference fields is prepended with one or more words containing logging bits that the barrier will use to check and record whether a particular field needs remembering. Just as



```
boolean isUnlogged(Obj src)
{
  return load(obj, -8) & MASK == 0;
}

/* .... */

if (ObjectBarrier.isUnlogged(src)) {
  ObjectBarrier.logOutOfLine(src)
}
```

```
TEST    -8[EBX] -128
JEQ     45


```

(a) Object

```
boolean isUnlogged(Obj src, FID fid)
{
  int off, msk;
  if (fid < HEADER_FIELDS) {
    off = HDR_OFF;
    mask = BASE_MSK<<fid;
  } else {
    fid -= HEADER_FIELDS;
    off = BASE_OFF - (fid / 32);
    msk = 1<<(fid & 31);
  }
  return load(obj, off) & msk == 0;
}

/* .... */

if (FB.isUnlogged(src, fieldID)) {
  FB.logOutOfLine(src, fieldID);
}
```

```
TEST    -8[EBX] -64
JEQ     45


```

(b) Field putfield

```
boolean isUnlogged(Obj src, int idx)
{
  int off = BASE_OFF - (idx / 32);
  int msk = 1<<(idx & 31);
  return load(obj, off) & msk == 0;
}

/* .... */

if (FB.isUnlogged(src, index)) {
  FB.logOutOfLine(src, index);
}
```

```
..
..
TEST    ...
JEQ     ...
```

(c) Field aastore

Figure 1. Object and field barriers, showing Java source for the barrier and x86 assembler for the fast path. The key insight here is that in a statically-typed language such as Java, the field identifier used in a putfield operation will be known statically. Thus the 13 lines of code used to test for a putfield field barrier (b) reduce to just a test and branch, just as for the object barrier (a). Because array indicies are generally not known, the aastore barrier cannot be optimized the same way. The putfield field barrier above will use bits in the object header for the first `HEADER_FIELDS` reference fields in any object. We also include in our evaluation the slot barrier described by Yang et al. See [31] for details of its implementation.

the object logging barrier does [2, 6], we apply the simple optimization of 'inverting' the sense of the logging bits, using `1` to indicate a field that requires logging (*unlogged*), and `0` to indicate a field that has been logged. This saves explicit initialization of new objects since new (nursery) objects don't need to be logged. We discuss a number of refinements and optimizations over this basic design below.

We considered an alternative design where logging bits were stored in side metadata. That approach simplifies a number of aspects of the design, but complicates others, reduces locality, and removes the opportunity for three important optimizations (static optimization of putfield barriers, object-level atomic logging bits, and the stealing of header bits, both discussed below). Side metadata does have the nice property of a fixed space overhead (1/32 when remembering 32-bit fields). In practice we find that the space overhead of our system is similar (3.3%, Section 5.3). We chose to use per-object prepended metadata because of the optimization opportunities that it offers.

Figure 1 shows pseudocode for the fast paths of our putfield and aastore barriers alongside that for the default object logging barrier. It is important to note that the object logging barrier and the putfield barrier are compiled to exactly the same instruction sequence. This is because they both require the test of a single bit at a fixed offset from the object pointer, and in both cases the bit mask and offset are fixed at compile time. In the case of the object-logging barrier this is because the offset and bit mask are constants, while in the putfield field barrier, it is because the field identifier is statically known, so the mask and offset can be partially evaluated by the compiler. This is not true for the aastore barrier since the index is not generally known at compile time.

Resetting of Log Bits The use of a log bit ensures that a field is only logged once.[1] However, the log bit must be reset once the collector has consumed the logged information. There are many ways this could be approached. If the logging bits were stored in side meta-data, then the side meta-data could be reset by the collector in bulk.

When the bits are embedded in per-object metadata as they are in our design, one might reset all fields for an object; reset just the bit that was logged; or reset the entire word containing the affected bit. If the barrier just logs the address of the updated *field*, then there needs to be a way of finding the associated log bit from the address of the field, which will be difficult or impossible in the absence of additional metadata. In our initial implementation, we log a tuple of the updated field address and the address of the word that

[1] See discussion about atomicity for precise guarantees.

contains the log bit. We then reset the entire log bit word when processing each log entry. (The bit stealing optimization described below introduces a small complication which we step around by using the low order bit of the address of the log bit word to record whether it is a regular log bit word or an object header. This allows us to reset the whole word or reset just the log bits, respectively.)

Atomicity Requirements In any language that supports mutator concurrency, the concurrency semantics of barriers need to be carefully considered. We consider this two ways: first, ensuring the correctness of the barrier *mechanism*, and second, ensuring that the barrier *semantics* are correctly observed. We implemented the field barrier for a generational collector and a reference counting collector. The semantics of the first are sound in the face of a race because when two threads race to remember a field, the race is benign since remembering an old-to-young pointer more than once is not incorrect, just marginally inefficient. On the other hand, the semantics of the reference counting barrier depend on strong concurrency guarantees because the barrier must log both the before-value of the to-be-updated field, as well as the field's address. At collection time, the field's address is dereferenced to determine the after-value of the field. If the barrier fires twice, it will likely remember two different before-values, but will only see one after-value, leading to incorrect reference counts being generated.

For the generational barrier, this means that the fast path can perform an unsynchronized check of the relevant logging bit. If logging is required, the barrier remembers the address of the modified field and then sets the logging bit. Because there may be a race to update the word holding the logging bit, a compare-and-swap (CAS) is required when setting that bit. As noted above, it is not incorrect for this barrier to log a field twice.

For the reference counting barrier, it is essential that the entire logging operation is performed only once, and is performed atomically. This means that another thread: (*i*) must not log an already logged field, and (*ii*) must not write to a field while it is being logged. Note that these semantics apply equally to the object-logging reference counting barrier.

These atomicity requirements mean that the reference counting slow path logging operation needs to be performed in a critical section. It is sufficient to guard the logging with a per-object *logging* bit, just as is done by the object-logging reference counting barrier.

The reason that a per-object bit is sufficient is that any mutator that attempts to write to a field being logged will see the field as unlogged and thus enter the slow path, at which point it will be locked out by the per-object *logging* bit. The per-object *logging* bit obviates the need for a CAS when setting the per-field logging bit (because the lockout is per-object, there is no longer a race to update any other logging bits). The fast path remains unsynchronized, identical to the generational field-remembering barrier.

Stealing Header Bits We observe that the two bits in the object header used by the default object-remembering barrier (Section 2), could as an optimization, be repurposed for field-logging bits. We furthermore observe that (for the generational barrier) only a single bit is required per remembered field, allowing two reference fields to have their state recorded in the object header. More generally, if there are more available bits in the object header, for example, in a 64-bit implementation, these bits can be used for logging field state, (significantly) reducing the need for prepended logging state.

The bit-stealing optimization is shown in Figure 1(b), in lines 4-6, where `HEADER_FIELDS` indicates the statically-determined number of reference fields that may have their state in the object header. In our experiments, we set this constant to 2 for the generational collector, corresponding to the number of bits previously committed to the object barrier. We evaluated this optimization on both the putfield and aastore barriers and found that while there was a significant advantage for the putfield, there was no advantage for the aastore, and sometimes a slowdown. This is explained by two factors. First, because the calculation of offsets in the aastore barrier cannot generally be statically evaluated (since the index is generally unknown at compile time), the space-saving optimization would introduce a new conditional into the fast path of the aastore barrier. Second, reference arrays generally have more reference fields in them, so the marginal space-saving advantage of the optimization is greatly diminished anyway. Consequently, we only apply the optimization to the putfield barrier.

Optimizing aastore More Aggressively While in general, the array index is not known statically, there is a substantial literature on compiler optimizations for arrays, including array bounds check elimination [9]. In principle, such targeted optimizations could be used to more aggressively optimize the aastore barrier. We have not explored such optimizations, but see that as an important consideration if the aastore field barrier is to be used in a production setting.

Coarser Array Remembering One of the motivations for the field-logging barrier was to avoid the pathology of remembering large, sparsely updated reference arrays. The field barrier takes that objective to an extreme, remembering exactly the set of modified fields. In principle, the same design could be coarsened to remember 2^n fields at a time. The effect would be to logarithmically reduce the rate at which the aastore slow path is taken. We have implemented this optimization, but as we report in Section 5.4, found that it made no significant impact on the overall performance of our aastore barrier on the workloads we evaluated.

Threshold For Field Remembering A simpler variation on the barrier design is to conditionally apply it to large arrays, leaving scalars and small arrays to use the object-remembering barrier. This approach introduces a size test in the aastore fast path. We explored this approach with the reference counting collectors, and found that was effective at reducing the overhead, but in doing so, the common case barrier is less precise. This technique thus introduces a 'bounded precision' as opposed to the full precision of the field barrier.

Impact on Free List Implementations Some free list allocators, including the one used by one of our reference counting collectors, require the ability to map from an object instance to the region of memory (cell) into which the object was allocated, which may be larger than the object, and may or may not have the same alignment. This is relatively straightforward with a fixed-sized object header. However, the introduction of logging bits prepended to the object complicates this reverse lookup, adding an additional type-dependent step to the calculation. We found engineering an efficient solution to this problem to be tedious.

4 Methodology

In this section, we present the software, hardware, and measurement methodologies we use. We use as our methodological starting point the barrier analysis work introduced by Yang et al. [31].

We implement the barriers in version 3.13 of Jikes RVM [1], with a production configuration that uses a stop-the-world generational Immix [8] collector, a free list-based reference counting collector, and an instance of RCImmix [24] and a high performance free list-based reference counting collector.[2]

Measurement Methodology We hold heap size constant for each benchmark, but because our focus is not the performance of the garbage collector itself, we use a generous 6× minimal heap size for each benchmark use a fixed 32 MB nursery for the generational collector and force collections every 32 MB for the reference counting collectors.

We use Jikes RVM's profile-based two-stage builds which maximize performance of the runtime by profiling the runtime before recompiling the runtime with compiler advice gathered during that profile run. When executing each benchmark, we use the *warmup replay* methodology to remove non-determinism inherent to the adaptive optimization system. Before running any experiment, we first gather compiler optimization profiles from the best performance run from a set of runs for each benchmark. Then, when we run each experiment, the benchmark is first completely executed, allowing the run-time to warm up (allowing all the class loading and method resolving work to be done), and then the compiler uses the pre-collected optimization profiles to aggressively compile the benchmark and disallows further recompilation, before executing a second, timed iteration of the workload. This methodology greatly reduces non-determinism from the adaptive optimizing compiler. Note that we use the compiler optimization advice gathered from the status quo build. However, since our different builds impose little change in the run-time system, we expect the bias introduced by using the same advice to be minimal as well.

We execute most benchmarks 50 times but we execute noisy benchmarks many more times in order to reduce the measurement error. We report means and 95% confidence intervals for each such measurement. In Section 5.2, we briefly discuss the architecture-sensitive nature of the variability in workload performance.

The variability in the workloads is a function of both workload and microarchitecture. For example, 50 iterations of luindex yield a 95% confidence interval of 4.5% on the i7-6700K, but well under 1% on the i7-920 and FX-8320. On the i7-6700K, we executed luindex 600 times to yield 95% confidence intervals of under 1%.

Hardware and Software Environment Table 1 lists the characteristics of the machines used in our evaluation. Our principal experiments are conducted on the 14 nm Intel Core i7-6700K processor (Skylake, 4 GHz) with 16 GB of 1866 MHz DDR4 RAM. To evaluate the impact of microarchitecture, we also use three other Intel processors and two AMD processors, listed in Table 1. We use Ubuntu 18.04.2 LTS server distribution running a 64 bit (x86_64) 4.15.0-21 Linux kernel on all of the machines.

Benchmarks We use the SPECjvm98 [25], DaCapo [4], and JBB2005 [5] benchmark suites. Unfortunately some benchmarks from the DaCapo suites do not run with the version of Jikes RVM we base our work on, because it relies on the old Classpath class libraries. Consequently we use all of the benchmarks from the dacapo-2006-10-MR2 suite plus avrora and sunflow from the dacapo-9.12-bach suite.

Table 2 outlines the characteristics of these benchmarks with respect to the behavior of the default object-remembering barrier, when executed with a modest 32MB nursery. The first column gives the mutator running time in milliseconds. The next three columns show the rate at which the fast path is taken for putfield (stores to non-array reference fields), aastore (stores to reference arrays) bytecodes, and both combined, expressed in fast path executions per microsecond. The next three columns show how often the respective slow paths are taken per millisecond. The final column shows how many bytes are logged per microsecond by the write barrier.

Note that there is considerable variation in each of these measures among the benchmarks. In particular, SPECjvm98 benchmarks, which are much simpler workloads [4], execute the fast path at about half the rate of the other benchmarks,

[2] Our source code is available on github: https://github.com/steveblackburn/jikesrvm-fieldbarrier-ismm/.

Table 1. Processors used in our evaluation.

Vendor	Intel				AMD	
Architecture	**i7-6700K**	**D-1540**	**E3-1270**	**i7-920**	**A10-7850**	**FX-8320**
Model	Core i7-6700K	Xeon D-1540	Xeon E3-1270	Core i7-920	A10-7850K	FX-8320
Family	Skylake	Broadwell	Sandy Bridge	Bloomfield	Steamroller	Piledriver
Technology	14 nm	14 nm	32 nm	45 nm	28 nm	32 nm
Clock	4.0 GHz	2.0 GHz	3.4 GHz	2.6 GHz	3.7 GHz	3.5 GHz
Cores × SMT	4 × 2	8 × 2	4 × 2	4 × 2	4 × 1	8 × 1
LL Cache	8 MB	12 MB	8 MB	8 MB	4 MB	8 MB
Memory	16 GB DDR4-1866	16 GB DDR4-2133	4 GB DDR3-1333	3 GB DDR3-1066	8 GB DDR3-2133	16 GB DDR3-1866

Table 2. Characteristics of the benchmarks used in this evaluation, as measured when using a generational barrier with a modest 32 MB nursery on the i7-6700K. The table shows mutator time; fast path take rates; slow path take rates; and bytes buffered. The measurements show take rates for a barrier on putfield, aastore and both.

Benchmark	**Time**	**Fast Path/μs**			**Slow Path/msec**			**Buffered**
	msec	pf	aa	pf+aa	pf	aa	pf+aa	B/μs
compress	1724	0	0	0	0	0	0	0
jess	320	13	12	25	4	75	79	1559
db	1313	2	21	23	0	0	1	280
javac	730	19	2	21	49	43	92	0
mpegaudio	983	6	0	6	0	0	0	0
mtrt	214	6	8	14	1	0	1	9
jack	485	25	6	31	237	18	255	3254
SPECjvm mean	*824*	*10*	*7*	*17*	*42*	*19*	*61*	*729*
antlr	530	7	1	8	6	6	12	0
avrora	2193	10	0	10	813	0	813	3
bloat	2159	136	2	137	5	25	29	31
eclipse	10543	6	9	15	24	38	62	2782
fop	582	3	0	3	20	0	20	0
hsqldb	534	27	8	35	1549	1543	3092	272
luindex	610	14	1	15	133	0	133	7
lusearch	959	29	2	30	2365	146	2511	15
pmd	482	37	6	43	8723	255	8977	225
sunflow	1367	37	0	37	197	102	299	17
xalan	473	42	25	68	3051	144	3195	8699
DaCapo mean	*1857*	*32*	*5*	*36*	*1535*	*205*	*1740*	*1095*
jbb2000	2323	22	10	32	1174	53	1227	11
min	*214*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
max	*10543*	*136*	*25*	*137*	*8723*	*1543*	*8977*	*8699*
Total mean	**1501**	**23**	**6**	**29**	**966**	**129**	**1095**	**903**

and execute the slow path more than an order of magnitude less frequently. Among the benchmarks, pmd has by far the highest slow-path take rate, and this is dominated by putfield operations. We chose to include the SPECjvm98 benchmarks in our analysis because they exhibit relatively extreme behaviors, which focused much of our effort in optimizing the barriers.

5 Results

We now present an evaluation of the field-logging barrier. We focus on the generational field-logging barrier but also provide a brief overview of the preliminary implementation of a reference counting field-logging barrier.

5.1 Generational Barrier

Figure 2 and Table 3 show the mutator performance of the generational field-logging barrier relative to the object-logging

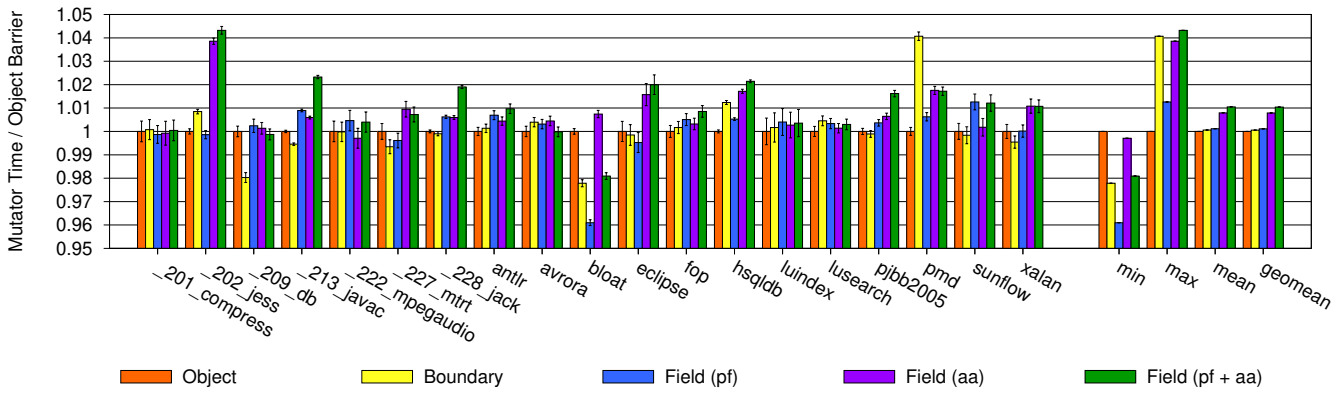


Figure 2. Performance characteristics of generational field-logging barriers and the boundary barrier on the i7-6700K. The graph shows mutator time for each barrier, normalized to the time for the object-logging barrier. Error bars indicate 95% confidence intervals.

barrier, running on the i7-6700K, the most recent of our six test machines. The field barrier had no statistically measurable affect on the performance of the generational collector, so we focus our evaluation on the mutator overhead (which is directly affected by the barrier).

Our implementation allows both field and object barriers to coexist. This lets us measure the putfield (pf) and aastore (aa) barriers separately as well as presenting the cost when both are used (pf+aa). When only one of the barriers is used, the other reverts to the default object barrier.

It is clear from Table 3 that the putfield barrier escapes the long-held tradeoff between precision and performance. On average over our 19 benchmarks, we see just a 0.1% overhead, and on eight of the benchmarks we see net improvements. This result is not entirely unexpected. For objects with two fields or fewer, the logged bit used by the putfield barrier resides in the existing object header, so there is no space or locality overhead. Furthermore, the field barrier removes to need to consult the object type and scan the object's fields during the slow path, instead directly remembering just the changed field. Thus unless a program makes heavy use of objects with more than two reference fields, the field-remembering barrier should present a performance win.

The aastore barrier presents a more complex tradeoff. Since arrays are more likely to be large, and perhaps sparsely accessed, the field barrier may reduce the cost of the slow path takes significantly. On the other hand, because the logging bits are always stored in extra words, there is always a space overhead, even for very small reference arrays. Also, because the array index is generally not statically known, the barrier cannot be reduced to something as simple as the object barrier in the common case. Specifically, the index must be converted into a word address and bit mask for every aastore. Nonetheless, the overhead of the aastore barrier is modest across the full set of benchmarks, with a net overhead of just 0.8% (1.008).

Putting the putfield and aastore barriers together, the field barrier performs well. Overall, the barrier has an overhead of 1.0% (1.010) on the i7-6700K. When the garbage collection overhead is included, the overall (total time) performance is the same (1.0%).

Examining the overhead across the benchmark suites is interesting. The overhead is lowest on the more realistic and richer workloads that comprise the DaCapo suite. We even see a statistically significant improvement on bloat (-1.9%). On the other microarchitectures (Section 5.2), the difference between the suites is even more marked, with a modest overall performance advantage on DaCapo on the E3-1270 (-0.6%).

The fast and slow path take rates from Table 2 show that DaCapo exercises the barriers significantly more heavily, which suggests that the overhead is not due to the direct cost of the barriers. This, and the relative simplicity of the SPEC benchmarks, suggests that perhaps the locality impact of the slightly larger object header is felt more acutely in the SPEC benchmarks. In Section 5.3 we evaluate the space overhead of the field-logging metadata in detail. As on-going work, we will investigate our locality conjecture via hardware performance counters.

5.2 Architectural Sensitivity

Table 4 presents an evaluation of the architectural sensitivity of the generational field-logging barrier across the six x86 microarchitectures listed in Table 1. Each column reports the normalized mutator performance of the field barrier (both putfield and aastore) relative to the object barrier on the respective microarchitecture. Note that for the DaCapo benchmarks, our default machine, the i7-6700K, is an outlier with 0.8% average overhead. The next worst microarchitecture

Table 3. Performance characteristics of generational field-logging barriers and the boundary (bdry) barrier compared to an object-remembering barrier on the i7-6700K. 95% confidence intervals are shown in grey.

Benchmark	Time/Object Barrier			
	bdry	pf	aa	pf+aa
compress	1.001 ±0.006	0.999 ±0.007	0.999 ±0.008	1.000 ±0.006
jess	1.008 ±0.036	0.999 ±0.047	1.039 ±0.007	1.043 ±0.002
db	0.980 ±0.021	1.002 ±0.000	1.001 ±0.001	0.999 ±0.003
javac	0.995 ±0.029	1.009 ±0.015	1.006 ±0.018	1.023 ±0.001
mpegaudio	1.000 ±0.010	1.005 ±0.006	0.997 ±0.013	1.004 ±0.006
mtrt	0.993 ±0.018	0.996 ±0.016	1.009 ±0.002	1.007 ±0.005
jack	0.999 ±0.021	1.006 ±0.014	1.006 ±0.014	1.019 ±0.001
SPECjvm mean	*0.997*	*1.002*	*1.008*	*1.014*
geomean	*0.997*	*1.002*	*1.008*	*1.014*
antlr	1.001 ±0.011	1.007 ±0.005	1.004 ±0.008	1.010 ±0.003
avrora	1.004 ±-0.001	1.003 ±0.000	1.004 ±-0.002	1.000 ±0.003
bloat	0.978 ±0.005	0.961 ±0.022	1.007 ±-0.024	0.981 ±0.002
eclipse	0.998 ±0.028	0.994 ±0.031	1.015 ±0.011	1.019 ±0.006
fop	1.002 ±0.011	1.005 ±0.007	1.003 ±0.009	1.009 ±0.004
hsqldb	1.012 ±0.010	1.005 ±0.017	1.017 ±0.005	1.021 ±0.001
luindex	1.002 ±0.011	1.004 ±0.008	1.003 ±0.009	1.004 ±0.008
lusearch	1.005 ±0.001	1.003 ±0.003	1.001 ±0.004	1.003 ±0.003
pmd	1.041 ±-0.021	1.006 ±0.013	1.018 ±0.002	1.017 ±0.003
sunflow	0.998 ±0.019	1.013 ±0.004	1.002 ±0.016	1.012 ±0.005
xalan	0.995 ±0.019	1.000 ±0.014	1.011 ±0.004	1.011 ±0.004
DaCapo mean	*1.003*	*1.000*	*1.008*	*1.008*
geomean	*1.003*	*1.000*	*1.008*	*1.008*
jbb2000	0.999 ±0.019	1.004 ±0.014	1.006 ±0.012	1.016 ±0.002
min	*0.978*	*0.961*	*0.997*	*0.981*
max	*1.041*	*1.013*	*1.039*	*1.043*
Total mean	**1.001**	**1.001**	**1.008**	**1.010**
geomean	**1.000**	**1.001**	**1.008**	**1.010**

Table 4. Performance overhead of field barriers across the six microarchitectures listed in Table 1. 95% confidence intervals are shown in grey.

Benchmark	Intel				AMD	
	6700	2600	1270	920	7850	8320
compress	1.000 ±0.006	1.000 ±0.004	0.993 ±0.005	1.000 ±0.002	1.005 ±0.001	1.000 ±0.000
jess	1.043 ±0.002	1.038 ±0.002	1.024 ±0.003	1.033 ±0.002	1.024 ±0.002	1.039 ±0.003
db	0.999 ±0.003	1.013 ±0.007	1.015 ±0.004	1.028 ±0.004	1.021 ±0.001	1.022 ±0.002
javac	1.023 ±0.001	1.022 ±0.001	1.023 ±0.001	1.014 ±0.002	1.004 ±0.002	1.009 ±0.001
mpegaudio	1.004 ±0.006	1.001 ±0.011	1.005 ±0.010	1.002 ±0.002	0.996 ±0.002	1.016 ±0.002
mtrt	1.007 ±0.005	1.015 ±0.008	1.008 ±0.004	1.009 ±0.004	1.008 ±0.003	1.003 ±0.002
jack	1.019 ±0.001	1.014 ±0.001	1.020 ±0.002	1.018 ±0.003	1.022 ±0.002	0.995 ±0.002
SPECjvm mean	*1.014*	*1.015*	*1.012*	*1.015*	*1.012*	*1.012*
geomean	*1.014*	*1.015*	*1.012*	*1.015*	*1.011*	*1.012*
antlr	1.010 ±0.003	1.007 ±0.003	0.997 ±0.008	1.012 ±0.005	1.000 ±0.009	1.003 ±0.001
avrora	1.000 ±0.003	1.001 ±0.003	0.988 ±0.003	0.998 ±0.003	0.995 ±0.006	0.992 ±0.003
bloat	0.981 ±0.002	0.959 ±0.002	0.956 ±0.003	0.980 ±0.003	0.981 ±0.002	0.973 ±0.002
eclipse	1.019 ±0.006	1.016 ±0.006	1.013 ±0.007	0.995 ±0.016	1.012 ±0.007	1.002 ±0.008
fop	1.009 ±0.004	1.004 ±0.003	0.984 ±0.002	0.999 ±0.003	0.993 ±0.007	1.005 ±0.005
hsqldb	1.021 ±0.001	1.034 ±0.012	1.011 ±0.001	1.016 ±0.005	1.037 ±0.012	1.040 ±0.005
luindex	1.004 ±0.008	0.983 ±0.014	1.001 ±0.013	1.005 ±0.006	0.999 ±0.018	0.995 ±0.005
lusearch	1.003 ±0.003	1.001 ±0.006	0.985 ±0.006	0.995 ±0.005	0.996 ±0.006	0.998 ±0.006
pmd	1.017 ±0.003	1.015 ±0.006	0.989 ±0.006	0.993 ±0.005	1.005 ±0.003	1.008 ±0.004
sunflow	1.012 ±0.005	1.019 ±0.009	1.013 ±0.004	1.017 ±0.011	1.008 ±0.011	1.002 ±0.013
xalan	1.011 ±0.004	1.012 ±0.003	0.991 ±0.004	1.005 ±0.003	1.001 ±0.004	1.000 ±0.004
DaCapo mean	*1.008*	*1.005*	*0.994*	*1.001*	*1.002*	*1.002*
geomean	*1.008*	*1.004*	*0.993*	*1.001*	*1.002*	*1.001*
jbb2000	1.016 ±0.002	1.011 ±0.002	1.001 ±0.002	1.004 ±0.001	1.009 ±0.002	1.072 ±0.004
min	*0.981*	*0.959*	*0.956*	*0.980*	*0.981*	*0.973*
max	*1.043*	*1.038*	*1.024*	*1.033*	*1.037*	*1.072*
Total mean	**1.010**	**1.009**	**1.001**	**1.006**	**1.006**	**1.009**
geomean	**1.010**	**1.009**	**1.001**	**1.006**	**1.006**	**1.009**

has half the average overhead (0.4%), and on the E3-1270, there is a modest average speedup on DaCapo (-0.7%).

The Intel E3-1270 has the lowest overhead overall (0.1%). The overheads on SPECjvm are very similar across architectures, only ranging from 1.1% to 1.5% overhead. The greatest variation is on jack, where the AMD architectures see a slowdown of 2.2% and a speedup of 0.5%. Among the DaCapo benchmarks, many were very consistent across architectures, including bloat, which sees consistent speedups. hsqldb sees the biggest range, from 1.1% to 4% slowdown. The largest variation is seen on jbb2000, which varies from 0.1% (E3-1270) to 7.2% (FX-8320) slowdown. Note that the jbb2000 results have very tight 95% confidence intervals, suggesting that this is worthy of further investigation, as it may help better characterize the sources of performance differences between field and object remembering barriers.

Table 5. Space overhead of field barriers using the generational collector. The first column shows the total allocation (MB) of each benchmark when using the object barrier. The other columns show the normalized amount of allocation for field barriers with putfield, aastore, and both.

Benchmark	MB	Field Barrier		
	Alloc	pf	aa	pf+aa
compress	65	1.000	1.000	1.000
jess	262	1.011	1.038	1.049
db	74	1.000	1.009	1.010
javac	174	1.042	1.004	1.054
mpegaudio	0	1.039	1.002	1.041
mtrt	93	1.022	1.009	1.027
jack	253	1.058	1.002	1.061
SPECjvm mean	*132*	*1.025*	*1.009*	*1.035*
antlr	215	1.013	0.999	1.013
avrora	52	1.088	1.001	1.088
bloat	1099	1.002	0.983	1.007
eclipse	2741	1.024	1.021	1.045
fop	49	1.023	1.006	1.028
hsqldb	116	1.068	1.025	1.093
luindex	35	1.009	1.002	1.010
lusearch	3846	1.001	1.008	1.005
pmd	337	1.022	1.013	1.029
sunflow	1141	1.017	1.009	1.002
xalan	882	1.006	1.009	1.022
DaCapo mean	*956*	*1.025*	*1.007*	*1.031*
jbb2000	1723	1.027	1.038	1.054
min	*0*	*1.000*	*0.983*	*1.000*
max	*3846*	*1.088*	*1.038*	*1.093*
Total mean	**693**	**1.025**	**1.009**	**1.034**
geomean	**188**	**1.025**	**1.009**	**1.033**

We found it interesting to note how the microarchitecture affected the stability of measurements on certain benchmarks. In particular, eclipse and luindex are striking. Among the Intel machines, i7-6700K and i7-920 had a six-fold difference in the 95% confidence intervals on luindex (4.43% v 0.73%). Of the two AMD machines, A10-7850 had a 4.6-fold higher confidence interval than FX-8320 (3.25% v 0.70%). Conversely, the A10-7850 had a slightly lower error (0.95%) than the FX-8320 (1.20%) on eclipse, demonstrating that stability is a function of both workload and microarchitecture. We eventually tightened the confidence intervals to under 1% for all benchmarks by running the benchmarks for 400 invocations or more when necessary, rather than the 50 invocations used by default. We are unsure why those particular combinations of benchmark and microarchitecture are so unstable. However, the instability is not related to the field barrier; the same instability was present for each barrier in each case.

5.3 Space Overhead

Table 5 presents an evaluation of the space overhead of the field barrier. The first column reports the bytes allocated (in MB) during the execution of the benchmark when using the default object-logging barrier. The remaining three columns report the normalized number of bytes of allocation for the field barrier applied to putfield, aastore and both. The totals show an overhead of 2.5% for putfield, 0.9% for aastore, and 3.4% for both together.

Recall that the field barrier requires a *logged* bit for each reference field or array element of the object. For arrays, these bits are packed into words preceding the object's header, while for scalars the first two bits are packed into the object header and any remaining bits are packed into words preceding the object header. Perhaps surprisingly, the average space overhead for putfield (2.5%) is more than twice that of the overhead for aastore (0.9%). This is probably explained both by the predominance of scalar types, and the relative scarcity of small reference arrays (where the space overhead is greatest).

5.4 Reference Counting

Table 6 shows mutator performance of the reference counting field-logging barrier relative to the object-logging reference counting barrier, running on the i7-6700K. As in Section 5.1, we show overheads for putfield, aastore and both.

The results reveal that the reference counting barrier has not been carefully tuned, yielding an overhead of about 2%, substantially higher than that of the generational field barrier. The results also show a very high level of variability in the results. A number of benchmarks, including lusearch, pmd, eclipse, and jess show considerable variability.

Recall that the reference counting barrier differs from the generational barrier only in its implementation of the slow path, which must capture both old and new values, and must do so atomically. We considered the possibility that the field barrier leads to more slow path events and thus more costly atomic operations, and hence worse performance than seen with the field barrier for the generational collector. We first thoroughly checked the efficiency of the slow path implementation, and also checked the frequency of slow path takes. Interestingly, we noted that on the most problematic benchmark, jess, the number of slow-path takes was 30× higher with the field barrier than for the default object-based reference counting barrier. However, we also noted that the number of slow-path takes was orders of magnitude lower for the reference counting barriers than for the generational barriers. These investigations led us to implement a variable-granularity aastore barrier and a size threshold (Section 3). We also considered the possibility that the slowdown was related to the reference counting collector's use of a free list. We eliminated this concern by also implementing the barrier in RCImmix [24], a high performance reference counting

Table 6. Performance characteristics of reference counting field-logging barriers compared to an object-remembering reference counting barrier on the i7-6700K. 95% confidence intervals are shown in grey.

Benchmark	Time/Object Barrier		
	pf	aa	pf+aa
compress	0.998 ±-0.001	0.994 ±0.003	0.992 ±0.004
jess	0.988 ±0.162	1.162 ±-0.010	1.148 ±0.002
db	1.086 ±0.026	1.016 ±0.095	1.099 ±0.013
javac	1.069 ±0.008	1.019 ±0.063	1.068 ±0.010
mpegaudio	1.012 ±0.019	1.009 ±0.022	0.998 ±0.026
mtrt	1.010 ±0.009	1.007 ±0.013	1.011 ±0.009
jack	1.021 ±0.004	1.015 ±0.010	1.022 ±0.002
SPECjvm mean	*1.026*	*1.032*	*1.048*
geomean	*1.026*	*1.031*	*1.047*
antlr	0.984 ±0.020	1.008 ±-0.001	0.998 ±0.014
avrora	0.996 ±-0.004	0.990 ±0.002	0.988 ±0.004
bloat	1.049 ±0.001	1.009 ±0.040	1.043 ±0.008
eclipse	1.004 ±0.051	1.037 ±0.023	1.011 ±0.045
fop	0.998 ±0.009	0.997 ±0.013	0.996 ±0.009
hsqldb	1.006 ±0.019	1.037 ±-0.011	1.022 ±0.005
luindex	1.076 ±0.044	1.048 ±0.070	1.027 ±0.049
lusearch	1.127 ±0.025	0.997 ±0.137	1.095 ±0.050
pmd	1.034 ±-0.098	1.028 ±-0.117	0.876 ±0.112
sunflow	1.002 ±0.005	1.000 ±0.005	1.001 ±0.008
xalan	1.029 ±0.014	1.014 ±0.029	1.037 ±0.007
DaCapo mean	*1.028*	*1.015*	*1.009*
geomean	*1.027*	*1.015*	*1.007*
jbb2000	1.011 ±0.015		1.020 ±0.003
min	*0.984*	*0.000*	*0.876*
max	*1.127*	*1.162*	*1.148*
Total mean	**1.026**	**0.968**	**1.024**
geomean	**1.026**	**1.021**	**1.022**

collector that uses a bump pointer allocator rather than a free list. However, we observed a very similar performance slowdown to that with the traditional reference counting collector.

Threshold and Variable-Granularity Aastore Barrier In Section 3 we considered two variations on the aastore barrier, one that changed the granularity of the remembered bit to be 2^n fields, and one that only used the field barrier for arrays larger than some threshold. When n is 0 in the variable-granularity barrier, the barrier reduces to the original aastore field barrier, remembering array fields one at a time. As n grows, the number of fields that are remembered at a time doubles. n is a constant for any particular build of the virtual machine, allowing aggressive optimization. When n is 1 for the threshold barrier, the barrier reduces to the original aastore field barrier, and as n rises, arrays of size n or less use the default object-remembering barrier.

We were surprised to find that neither of these optimizations were effective in bringing the overhead of the reference counting field barrier down significantly. In the both cases, the granularity/thresholding introduced noticeable overhead reductions, but they only removed some of the overhead, and were not able to yield performance similar to the generational field barrier. We were unable to resolve this performance anomaly by the time this paper was completed, so leave it as ongoing work.

6 Conclusion

Write barriers are a ubiquitous and performance-critical mechanism used by garbage collectors. Commonly-used barrier designs sacrifice temporal and/or spatial accuracy in order to be performant. A low-overhead, temporally and spatially precise barrier has been elusive until now. We present the first design of a write barrier that remembers fields with temporal and spatial precision, and does so at low overhead. We present a detailed analysis of the barrier and its performance. The barrier design and our analysis should be helpful to garbage collection researchers, providing them with fresh insights into barrier behavior and a new mechanism to consider when designing collectors.

Acknowledgments

This material is based upon work supported by Huawei and the Australian Research Council under grant DP190103367. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of Huawei or the Australian Research Council.

References

[1] B. Alpern, , S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes RVM Project: Building an Open Source Research Community. *IBM System Journal* 44, 2 (2005), 399–418.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM System Journal* 39, 1 (Feb. 2000). https://doi.org/10.1147/sj.391.0211

[3] A. Azagury, E. K. Kolodner, E. Petrank, and Z. Yehudai. 1998. Combining Card Marking with Remembered Sets: How to Save Scanning Time.

In *ACM International Symposium on Memory Management*. Vancouver, Canada, 10–19. https://doi.org/10.1145/286860.286862

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. 169–190. https://doi.org/10.1145/1167515.1167488

[5] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. 2007. pjbb2005: The pseudoJBB benchmark. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

[6] S. M. Blackburn and A. Hosking. 2004. Barriers: Friend or Foe?. In *ACM International Symposium on Memory Management*. Vancouver, Canada, 143–151. https://doi.org/10.1145/1029873.1029891

[7] S. M. Blackburn and K. S. McKinley. 2002. In or Out? Putting write barriers in their place. In *The ACM International Symposium on Memory Management*. 175–184.

[8] S. M. Blackburn and K. S. McKinley. 2008. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Locality. In *ACM Conference on Programming Language Design and Implementation*. Tuscon, AZ, 22–32. https://doi.org/10.1145/1375581.1375586

[9] R. Bodík, R. Gupta, and V. Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 321–333. https://doi.org/10.1145/349299.349342

[10] H-J. Boehm, A. Demers, M. Weiser, B. Hayes, D. Bobrow, and S. Shenker. 1990. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*. San Francisco, CA.

[11] C. Chambers. 1992. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the 1992 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 615*. Springer-Verlag, Utrecht, The Netherlands.

[12] G. O. Collins. 1961. Experience in Automatic Storage Allocation. *Commun. ACM* 4, 10 (Oct. 1961), 436–440.

[13] D. Detlefs and V. K. Nandivada. 2004. *Compile-time Concurrent Marking Write Barrier Removal*. Technical Report. Mountain View, CA, USA.

[14] D. Dice. 2011. False sharing induced by card table marking. https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card

[15] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-The-Fly Garbage Collection: An exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 965–975.

[16] R. R. Fenichel and J. C. Yochelson. 1969. A LISP Garbage-collector for Virtual-memory Computer Systems. *Commun. ACM* 12, 11 (Nov. 1969), 611–612. https://doi.org/10.1145/363269.363280

[17] U. Hölzle. 1993. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA*, J. Eliot B. Moss, Paul R. Wilson, and Benjamin Zorn (Eds.). ftp://ftp.cs.utexas.edu/pub/garbage/GC93/hoelzle.ps

[18] A. L. Hosking and R. L. Hudson. 1993. Remembered Sets Can Also Play Cards. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

[19] A. L. Hosking, J. E. B. Moss, and D. Stefanović. 1992. A Comparative Performance Evaluation of Write Barrier Implementations. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*. Vancouver, BC, 92–109.

[20] R. L. Hudson and J. E. B. Moss. 2001. Sapphire: Copying GC Without Stopping the World. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)*. 48–57. https://doi.org/10.1145/376656.376810

[21] Y. Levanoni and E. Petrank. 2001. An On-the-fly Reference Counting Garbage Collector for Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. ACM, Tampa, FL, 367–380.

[22] H. Lieberman and C. E. Hewitt. 1983. A Real TIme Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (1983), 419–429.

[23] J. McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. https://doi.org/10.1145/367177.367199

[24] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. 2013. Taking off the Gloves with Reference Counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM, New York, NY, USA, 93–110. https://doi.org/10.1145/2509136.2509527

[25] Standard Performance Evaluation Corporation 1999. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98

[26] G. L. Steele, Jr. 1975. Multiprocessing Compactifying Garbage Collection. *Commun. ACM* 18, 9 (Sept. 1975), 495–508. https://doi.org/10.1145/361002.361005

[27] G. Tene, B. Iyengar, and M. Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[28] D. M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *ACM Software Engineering Symposium on Practical Software Development Environments*. 157–167. https://doi.org/10.1145/800020.808261

[29] M. T. Vechev and D. F. Bacon. 2004. Write Barrier Elision for Concurrent Garbage Collectors. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1029873.1029876

[30] P. R. Wilson and T. G. Moher. 1989. A "Card-marking" Scheme for Controlling Intergenerational References in Generation-based Garbage Collection on Stock Hardware. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. 87–92. https://doi.org/10.1145/66068.66077

[31] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. 2012. Barriers Reconsidered, Friendlier Still!. In *Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16*. https://doi.org/10.1145/2258996.2259004

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

[33] K. Zee and M. Rinard. 2002. Write Barrier Removal by Static Analysis. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 191–210. https://doi.org/10.1145/582419.582439

[34] B. G. Zorn. 1993. The Measured Cost of Conservative Garbage Collection. *Software Practice & Experience* 23, 7 (1993), 733–756.