Iso: Request-Private Garbage Collection

TIANLE QIU, Australian National University, Australia
STEPHEN M. BLACKBURN, Google, Australia and Australian National University, Australia

Large-scale, revenue-critical application services are often written in Java or other memory-safe languages whose type systems do not expose immutable state. Such applications are especially exposed to garbage collection performance overheads because latency, throughput, and memory consumption are first-order concerns for service providers. We observe that: i) An important class of server applications are request-based: they scale by concurrently servicing large numbers of *quasi-independent* requests. ii) Object lifetimes are strongly tied to request lifetimes. iii) Most objects remain private to the request in which they were allocated. iv) Global operations are the primary impediment to responsiveness at scale.

If we could perform *request-private garbage collection*, we might achieve both responsiveness and efficiency at scale. Unfortunately, this straightforward insight runs into significant practical problems. The most obvious of these is that a request-private collection cannot safely move objects that may be referenced outside the scope of that request, and yet moving objects is a requirement of most modern high performance collector designs. This dilemma can be sidestepped by exploiting immutability, which is unfortunately not practical in languages like Java whose type systems do not expose it. We develop Iso, a garbage collector for request-based services that exploits a mark-region heap structure to solve these impediments and deliver outstanding performance.

The key contributions of this paper are that: i) We use opportunistic copying to solve the problem of practical thread-local garbage collection for languages without exploitable immutability. ii) We provide the first detailed analysis of the behavior of Java workloads with respect to thread-local collection, identify shortcomings of existing benchmarks and introduce a new one. iii) We design, implement, and evaluate Iso, a practical and effective request-private GC. We show that dynamic tracking of object visibility, a prerequisite for request-private GC, incurs an overhead of just 2% for important request-based workloads including Tomcat and Spring. Iso demonstrates that for suitable workloads, request-based garbage collection is extremely effective, outperforming OpenJDK with its default collector, G1, by 32% and 22% in execution time in a modest heap. This work presents the first request-private garbage collector for Java. It shows a promising way forward for highly responsive collection on an important class of large scale workloads.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Software performance**.

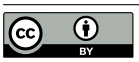
Additional Key Words and Phrases: Garbage Collection, Thread-Local Garbage Collection

ACM Reference Format:
Tianle Qiu and Stephen M. Blackburn. 2025. Iso: Request-Private Garbage Collection. *Proc. ACM Program. Lang.* 9, PLDI, Article 182 (June 2025), 23 pages. https://doi.org/10.1145/3729285

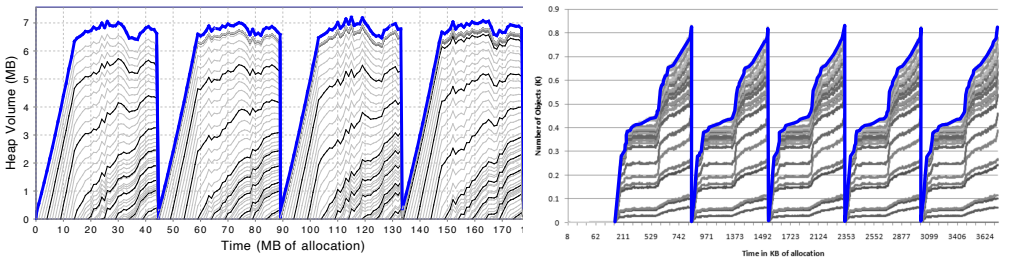
1 Introduction

Garbage collection is far from a solved problem [12]. The pursuit of application responsiveness at scale has led to production collector designs over the past decade that trade performance for responsiveness. Collectors face two independent challenges: i) identifying and reclaiming garbage in heaps with large live object graphs is expensive, and ii) collecting garbage without blocking the application is intrinsically expensive. These challenges grow as the scale of the applications grow.

Authors' Contact Information: Tianle Qiu, Australian National University, Canberra, Australia, tianle.qiu@anu.edu.au; Stephen M. Blackburn, Google, Sydney, Australia and Australian National University, Canberra, Australia, steveblackburn@google.com.



ACM 2475-1421/2025/6-ART182
https://doi.org/10.1145/3729285



(a) Heap composition: SPECjvm98 javac [8]. (b) Heap composition: PHP wordpress [34].

Fig. 1. **Well-timed garbage collection can have near-zero cost.** Heap composition graphs taken from prior work [8, 34], with the top-most line in each graph indicating total heap occupancy (annotated in blue). In each case, the total heap occupancy periodically falls to zero. On the javac benchmark, if collections are timed to occur at exactly 44 MB, 88 MB and 132 MB, there will be almost no objects live during each collection, so the collections will have near-zero cost. The same occurs in wordpress and other PHP workloads [34].

Application servers typically run many largely-independent requests which scale well because they are embarrassingly parallel, but they expose garbage collection as a major bottleneck.

In this work, we exploit the fact that many services execute individual requests as quasi-independent, mostly-isolated tasks. This observation allows us to directly address both of the challenges above: i) at the end of a request, the request's object graph will be small or completely empty, making collection cheap, and ii) when a computation is isolated, its heap can be collected independently, and if performed *between* requests, the collection will not impact executing requests.

The first of these observations has been understood for decades [53, 54] but not widely exploited. This is illustrated by the javac benchmark, which was regarded as one of the most interesting and GC-intensive benchmarks in the SPECjvm98 suite. The benchmark has a quirk: if the heap is sized just right, then the cost of GC plummets unexpectedly to near zero. If the heap is made slightly smaller or slightly larger, the GC cost will return to a significant level. Figure 1(a) illustrates why. The figure shows a heap composition graph depicting object liveness in MB as a function of time, broken down into bands of allocation cohorts.[1,2] The benchmark has four prominent phases, at the end of each of which the heap has almost no live objects. Since the cost of a tracing collector is dominated by a trace of live objects, collecting at the end of each of the phases of this benchmark is very inexpensive, so a luckily-sized heap can lead to collections that align perfectly with these phases, yielding a near-zero GC overhead. Figure 1(b) shows the same for a series of requests in the PHP wordpress workload, suggesting similar opportunities for substantial savings [34].

Unfortunately, this straightforward observation falls down in the face of concurrency. If two or more threads simultaneously and independently run workloads with profiles like those in Figure 1, the opportunity for cheap collection would only exist as long as those threads' phases were perfectly synchronized. In practice this is intractable on multithreaded Java servers so the opportunity to collect between such phases is lost. The situation can be recovered if each thread works in isolation, collecting between its requests, regardless of when other threads might collect.

The idea of thread-local collection is also decades old [19]. However, until now, it has not been practical in a language like Java despite prior effort [20]. The difficulty lies in how to detect and respond to cases where thread-local isolation is violated [18, 19]. The key idea introduced by Doligez and Leroy [19] is to maintain an invariant of privacy for a thread-local heap. If a

[1]The authors of [8] and [34] follow the convention of measuring time in allocated bytes rather than wall clock time.
[2]Allocation cohorts allow the lifetimes of contemporaneously allocated objects to be tracked, which is not relevant here.

thread is about to make a private object public, it must first make the transitive closure of that object public. Unfortunately, this either involves moving the object and its transitive closure [2] or leaving the object in situ [20]. Moving the object and its transitive closure is expensive since maintaining referential integrity either requires a full thread-local collection every time an object is published [2] or indirection through forwarding pointers [30]. Potentially collecting at every pointer store is expensive not just because of the cost of frequently tracing the entire private heap, but because it requires that the compiler make every pointer store a GC safe point, which introduces direct overheads and reduces optimization opportunities. On the other hand, leaving the objects in place also appears to be expensive since it suggests that a non-moving heap is required [20], which sacrifices locality [32]. Languages whose type system exposes immutability can sidestep this dilemma by copying immutable objects [39], but this is not an option in Java. Consequently, the idea of thread-local collection has been elusive for Java.

This paper demonstrates Iso, the first practical request-local garbage collector for Java, and shows how it can dramatically improve performance for application servers that process isolated requests. A key insight is to exploit the idea of *opportunistic copying* offered by mark-region heaps [9, 15]. Opportunistic copying allows the moving of objects when convenient and leaving them in place when it is not, giving us the best of both worlds: a fast copying garbage collector that does not require prohibitively expensive evacuation upon publication.

The contributions of this paper are as follows: i) We develop and evaluate Iso, the first practical request-local garbage collector for Java. ii) We show that opportunistic copying is the key to performant thread-local garbage collection in languages like Java where immutability is uncommon. iii) We demonstrate that request-local collection can dramatically improve the performance of workloads that service isolated requests. iv) We provide the first detailed analysis of the behavior of such workloads, identify shortcomings of existing benchmarks and introduce a new one.

2 Background and Related Work

2.1 Arenas, Regions, and Request-Based Workloads

The observation that program behavior may result in groups of objects dying together, and that this provides an opportunity for efficient memory management has been around since the 1960s [44]. One realization of this insight is to allocate objects into regions of memory, which Hanson [31] called arenas, and free the entire arena once all objects are unreachable. Variations of this general idea have been built into programming language implementations including Cyclone [29] and ML Kit [1]. Generalizations of the idea include hierarchical relations among regions and combining on masse reclamation supported by regions with the per-object reclamation supported by a heap [23]. The lifetime requirements of region-based approaches make them awkward in general-purpose memory-safe object-oriented languages, but they are used in Java in the special cases of real-time memory management [11] and managing native memory [13].

Lieberman and Hewitt [38] note that the height of a program's activation stack offers a clue as to how many objects might be live, and thus an opportunity for cheap garbage collection through careful timing. Wilson and Moher [54] take the idea further, noting that certain points in a program might be more opportune than others, such as when a text editor was idle, and suggested providing hints to the collector. Jibaja et al. [34] note that PHP's request-based workloads exhibit distinct phases with heap occupancy falling to near-zero between requests (Figure 1(b)). Unfortunately this observation can't be exploited with traditional (global) garbage collection techniques if the service is multithreaded and requests are unsynchronized, since the good time to collect one thread will in general not coincide with the good time to collect another. Degenbaev et al. [15] note that some applications have regular *idle time*, which allows collections to be scheduled without affecting the

user experience. They implemented this in the V8 runtime where their application is a browser with a renderer loop that requires frames to be drawn at a given frequency. If there is slack time available before the next frame needs to be drawn, they can perform a collection without affecting the frame rate. Hudson and Clements [33] observe that in the Go language, goroutines provide nice lifetime hints and that thread-local garbage collection might allow for efficient collection of objects allocated during a goroutine. As far as we can tell this idea was never implemented.

We build on this prior work in the specific domain of request-based applications, where we know when requests start and end. We make similar observations to Jibaja et al. [34], but with large multi-threaded Java application services. We address the confounding problem of concurrent requests by developing practical thread-local collection for Java.

2.2 Thread-Local Garbage Collection

The Doligez-Leroy-Gonthier (DLG) parallel collector is structured in terms of a 'global' heap and a number of 'thread-local' heaps [18, 19]. Their design maintains the invariant that *objects in a local heap can only be externally referenced by the stack of the thread that owns the heap.* We call this the **DLG invariant**. In this work, we generalize and use the term 'heap' with respect to *logical* heaps whose contents may not necessarily be contiguous. The invariant allows each local heap to be collected independently, without synchronization among threads. Since Doligez and Leroy's work, there have been many collector designs based on or extending this key idea [2–4, 20, 30, 35, 39, 46, 51, 52]. One straightforward generalization of the idea is that multiple related threads can be associated with each local heap. This is known as *task*-local garbage collection.

Terminology. We note that the principal concern is one of *visibility*, not spatial proximity. In fact the key ideas are largely orthogonal to proximity [20, 39]. For this reason, throughout the paper we use terminology of *private*, *public*, and *publication*, in place of local, global, and globalization.

Maintaining the DLG Invariant. The DLG invariant can be maintained statically or dynamically. If thread-private objects can be precisely identified statically, then it is sufficient for such objects to be placed into the correct thread-private heap at allocation time [35, 46]. More generally, the invariant can be maintained dynamically by intercepting the publication of thread-private objects. Publication occurs at the moment a pointer to a private object is installed within a public object or a public root. An important corollary of the DLG invariant is that the publishing thread can only ever be the thread that allocated the object, since by definition only the allocating thread is aware of the private object. This eliminates the possibility that publication could occur due to the action of another thread. The DLG invariant can be maintained by the owning thread transferring the to-be-published object and its transitive closure to the public heap *immediately before* installing the pointer that would affect publication. A write barrier can be used to maintain the invariant by checking for any pointer store where the source is public and the destination is private.

There are three ways to maintain the DLG invariant when an object o is about to be published. i) o can be moved. This requires a costly full trace of the entire private heap at each publication event to update all pointers to o and those transitively moved with o [2], or indirection through forwarding pointers [30]. The former also requires that the compiler make each pointer store a GC safe point. ii) A copy of o, o', can be made in the public heap if o is immutable. This avoids the need to move o. iii) The private heap can be delineated logically rather than spatially via per-object metadata such as a bitmap. This only requires a transitive closure of the object being published, rather than a trace of the entire private heap [20]. Since, by definition each object can only be published once, the amortized cost of performing this closure in the write barrier is constant time.

In this work we use a per-object 'public' bit and a write barrier to maintain the DLG invariant, allowing us to track object privacy independently of heap structure and GC strategy. This separation

of concerns allows us to explore a range of design points. For example, straightforwardly combining privacy tracking with a classic mark-sweep collector design leads to Domani et al.'s approach [20], while combining it with a classic copying collector leads to Anderson's approach [2].

Weakening the DLG Invariant. Marlow and Peyton Jones [39] note that the DLG invariant's requirement to eagerly publish all objects transitively reachable from a published object is expensive, leading them to a more lazy approach. They move just the published object, and put in place read barriers to identify when references from it to its children are *read* by another thread. When the read barrier triggers, the triggering thread makes a blocking request to the owner of the private object for it to publish the referenced object. Although read barriers are often costly, GHC already uses a read-barrier, making the marginal cost of this design low. Marlow and Peyton Jones selectively apply this strategy to immutable objects, so publication consists of making a copy in the public heap. They handle mutable objects with a non-moving strategy in the style of Domani et al. [20]. This weakening of the invariant through read barriers is also used by Dolan et al. [17] and Filatov and Mikheev [25]. We avoid this approach because of the cost of read barriers in Java [58] and the inter-thread synchronization it introduces.

We will now describe thread-local collection in terms of the three key components of garbage collection: *allocation*, *identification*, and *reclamation*.

Allocation. Each thread allocates objects into its own (logical) private heap. The allocator may preemptively allocate some objects into the public heap if it knows at allocation time that they will become public [35, 46]. Generally a bump allocator is used for speed and locality, but Domani et al. use a free list and a single physical heap with a bitmap identifying whether objects are private or public in their non-moving design [20]. Marlow and Peyton Jones [39] do the same for mutable objects in their hybrid design, while using a bump allocator for immutable objects.

Identification. Because the DLG invariant guarantees that thread-private objects are only reachable from the owning thread's stack, it is sufficient to determine liveness of thread-private objects by pausing the owning thread and, starting with its stack, perform a mark through its local heap. All pointers extending out of the private heap can be ignored during the mark. At the completion of the thread-private mark, space within the private heap occupied by unmarked objects can be reclaimed. Private collections do not need to be coordinated, so various heuristics can be used for scheduling them. Identification is performed separately for the public heap, and must include the private heaps since the DLG invariant does not prevent private objects referencing public objects.

Reclamation. Private collections are free to use any of the conventional reclamation strategies found in the literature [9, 24, 40, 47]. Reclamation by evacuation, as used in the semi-space algorithm, is simple and fast [24]. It subsumes the need for a separate mark: each object encountered during a transitive closure over the reachable objects within the thread-private heap (*from-space*) is copied (*evacuated*) to a new area (*to-space*), with forwarding pointers used to ensure that each object is only copied once and all references to the object are redirected to the copy. In the case that the public and private heaps are spatially disjoint [2], at the end of the trace, the private from-space can be completely reclaimed, and to-space becomes the new thread-private heap. When private and public objects are spatially intermingled, the from-space may be punctuated by public objects not considered by the private collection. The gaps between surviving public objects can be made available for reuse either by a free list or a system such as Immix's line recycling [9]. The principal downside of evacuation in any collector is the need to reserve a to-space as large as the from-space in order to ensure that collection can succeed in the worst case that all objects survive. This is significantly mitigated in the case of private collection since if private collections do not overlap, it

is sufficient to reserve one copy space to service all private heaps. Domani et al. and Marlow and Peyton Jones use mark-sweep [40] for their non-moving private objects [20, 39].

Major Implementations. The major implementations of thread-local garbage collection have been in functional languages where the type system exposes immutability. We describe the Haskell implementation above [39], and mention the OCaml implementation [17]. Sivaramakrishnan et al. [45] later reflect on the OCaml implementation: *One surprising result we will justify in our evaluation is that the stop-the-world minor collector outperforms the concurrent [thread-local] minor collector in almost all circumstances, even as we cranked up the number of cores.* Guatto et al. [30] introduce a hierarchical relationship between heaps, which maps well to their parallel ML implementation and presents an interesting variation on the classic DLG invariant: objects may reference objects in ancestor heaps safely. They use the term 'entanglement' to describe references that would violate the inter-heap invariant. The collector performs evacuation on publication, but avoids the prohibitive overhead of Anderson's approach by using forwarding pointers rather than requiring a full thread-local collection at each object publication. Westrick et al. [52] describe the time and space overheads of their dynamic invariant check as less than 5% on average. Arora et al. [3] loosen restrictions on mutation made by prior work, allowing functional programs "to use mutation at will." The key to their approach is to track ambiguous pointers [10], and by doing so, conservatively ensure that public objects are not moved during local collections.

Java Implementations. We know of only three implementations of thread-local garbage collection for Java. Domani et al. [20] implemented a non-moving thread-local collector in an early IBM JDK. As far as we know this collector has not been put to use, presumably because of it being limited to non-moving runtimes. Jones and King [35] describe a static analysis for identifying thread-private objects, but not a working collector. Filatov and Mikheev [25] propose an implementation that uses read barriers to maintain the DLG invariant, but their work is limited to an evaluation of publication rates on select JDK8 workloads and does not describe a working implementation.

2.3 Mark-Region Heaps and Opportunistic Copying

Evacuating collectors are dominant among production systems today [16, 22, 26, 48]. They allocate using a bump pointer and have good locality but can only free memory once a region (such as a semi-space) has been completely emptied, with all live objects evacuated into another space, and they must set aside space in which to copy at each collection. At the other extreme, non-moving collectors are uncommon. They rely on free lists for allocation, which are more exposed to fragmentation and lead to worse locality. A mark-region heap [9] extracts elements of both design points, breaking the heap into fine- and coarse-grained lines and blocks, correspondingly roughly to a cache line and operating system page in size. At the end of each collection, a mark region collector sweeps its blocks. Any completely free block can be returned to the operating system. Contiguous free lines can be reused by the allocator's bump pointer. This design delivers the locality benefits of copying collectors without the same space overhead or the requirement that all live objects in the target region must move. Existing mark-region designs include Immix [9], the V8 garbage collector [15], and LXR [60].

The mark-region design means that at collection time, unlike in strictly-evacuating collectors, the collector *may* move any object, but *is not required* to move any object. This gives the collector the liberty to decide whether to move each live object, without affecting correctness. This feature is known as *opportunistic copying*. Iso uses opportunistic copying to pin public objects during private collections, and pin private objects during global collections, solving the dilemma of having to choose between fully moving [2] and non-moving [20] thread-local collector designs while being unable to exploit immutability.

2.4 Isolates

Isolates are a longstanding idea for efficiently creating independent virtual machine instances within a single process. The approach uses copy-on-write to maximize reuse and minimize startup while maintaining isolation [5, 42]. Although proposed early on as an extension to Java [41], it did not see widespread adoption, possibly because of the performance overheads and possibly because of soft demand for such services twenty years ago. More recently, isolates have made a reappearance in Java through the Graal Native Image VM [55, 56]. This implementation leverages a closed world assumption to compile the entire application ahead of time, and as a consequence sidesteps some of the overheads associated with early isolates implementations. The V8 JavaScript runtime implements isolates [49], which are widely used for serverless function execution, such as Cloudflare's Workers [14]. Isolates provide an excellent solution for certain application classes, however they maintain total isolation among tasks, which for many applications is problematic. In the case of Java, the requirement of a closed world assumption will also be problematic for some applications. As we discuss later (Table 1), all of the request-based workloads we explore here share state among requests, either through non-zero publication or in the case of lusearch, through intra-thread transfer of state.

2.5 Other Collector Designs

OpenJDK's default G1 garbage collector [16] performs tracing concurrently and copying in stop-the-world pauses. While it performs well as a general-purpose collector, it suffers from poor worst-case pauses, so is problematic in latency-sensitive applications, particularly when the live object graph is very large. The C4 [48], Shenandoah [26, 36] and ZGC [22] collectors were all designed to address this shortcoming and all build upon the region-based design G1 uses by adding concurrent copying, each using slightly different approaches [59]. However, concurrent copying is intrinsically expensive, so all three designs only achieve excellent responsiveness by making significant compromises with respect to memory footprint and/or CPU overhead [12]. LXR is a general purpose collector that addresses responsiveness by using reference counting rather than concurrent copying, outperforming G1 on both throughput and responsiveness, particularly in tighter heaps [60]. None of these designs exploit request lifetimes or perform thread-local collection. The design of Iso is largely orthogonal to these existing designs — Iso will very efficiently collect thread-private objects, falling back to another collector for public objects. For pragmatic reasons, our current implementation of Iso falls back to Immix, but in principle it could fall back to a high-performance general-purpose mark-region collector like V8's collector [15] or LXR [60].

3 Request-Private Garbage Collection

Before we introduce the design of Iso, we introduce request-local garbage collection, which combines two simple ideas: i) *isolation* of thread-private objects to enable thread-local collection [18, 19], which avoids costly global pauses, and ii) *timing* of collections to exploit systematic ebbs in heap occupancy (Figure 1), either via heuristics [38, 54] or by callbacks at request boundaries.

Returning to the example of the Wordpress PHP service depicted in Figure 1(b), if the server were single threaded, the heap could be collected very efficiently by any stop-the-world tracing collector simply by timing collections to coincide with requests. This could be achieved either via heuristics [38, 54], or via a callback attached to request completion. But if the server were multi-threaded, this simple approach becomes useless unless requests were somehow synchronized. However, if each thread were largely independent, thread-local GC could be used to independently collect each server thread's heap. Just as careful timing can dramatically reduce overheads for regular GC, it can be used to the same effect to carefully time thread-local GC. Note that not

only will well-timed thread-local collections be efficient, but with sufficient server parallelism, the collections will not be on any request's critical path, masking the cost of such GCs completely.

Thread Privacy Requirements. An application that exhibits little thread-private behavior cannot benefit from thread-local collection. We explore this property with respect to Java workloads in Section 5.3 and Section 6.2. Table 1 shows that publication rates among DaCapo benchmarks range from zero (sunflow) to 68% (h2). Note that it seems unlikely that any of these workloads were (consciously) designed for thread-local collection, since thread-local collection has not been practical for Java until now. Even so, the level of thread-privacy among them is encouraging.

Lifetime Requirements. The utility of request-private collection depends on the lifetimes of request-local objects coinciding with request boundaries (see Figure 1(b)). If most data allocated within a request lives beyond the request, then there is little benefit in collecting at the end of a request. On the other hand, if most data allocated within a request is private to the request and dies at the end of the request, a request-private collection should be highly profitable. This in turn depends on the collector knowing when a request has completed. Prior work suggested using heuristics to detect such events [38, 54]. We note that modern application frameworks such as Spring, Kafka, Lucene, Cassandra, and Tomcat provide their users with higher level abstractions, allowing callbacks to be incorporated with minimal change into the underlying framework, with *no change* to applications built on the frameworks. In this work, we leverage the fact that the DaCapo benchmark suite has built-in callbacks, `Callback.requestStart()` and `Callback.requestEnd()`, which we use to trigger request-local collections [6]. If such callbacks are not available, the request-private collector reduces to a thread-local collector, falling back to whatever collection triggers the thread-local collector has in place. For example, we found that in practice, lusearch, which is composed of many very small requests, exhibits excellent performance with Iso, even without using callbacks.

Generalizing to Multi-Threaded Requests. Request-private GC can be straightforwardly generalized to the case where requests are internally multi-threaded. In the simplest case, intra-request threads remain largely thread-private and all that is needed is that the end of request callback (or heuristic) is shared among all threads participating in the request, and each thread collects their heap independently. This will work well when the design of the server is such that only minimal state is shared (published) among the intra-request threads. Beyond this simple case, we can generalize thread-local GC to allow *groups of threads*, such as intra-request threads, to share state without publication. In practice this generalization only requires that it be possible for threads to be passed shared group-private state when they are created. With that in place, maintaining the invariant that no public object points to a private object continues to work, unchanged. In practice, we found that all of the request-based workloads in the DaCapo suite are internally single threaded, so we have not yet had a reason to implement support for requests that are internally multi-threaded.

4 Iso's Design

At the high level, Iso's algorithm is fairly simple. Each object is assigned a *public* bit which the runtime maintains dynamically to respect the DLG invariant that no public pointer refer to a private object. Iso allocates objects into 32 KB blocks owned by the thread allocating the object. Objects larger than half a block are handled separately, in the large object space (LOS). Iso is distinctive in that while it maintains the DLG invariant, it also allows private and public objects to co-reside spatially within a block. These are *mixed* blocks, in contrast to *private* and *public* blocks which contain only private and public objects respectively. Iso maintains a second key invariant, which we call the **Block invariant**: *A block may not contain private objects owned by more than one thread.*

```
1  void checkDLG(Object src, Object tgt) {
2      if (!isPublic(src)) {
3          return;
4      }
5      if (tgt == NULL || isPublic(tgt)) {
6          return;
7      }
8      publish(tgt);
9  }
```

Listing 1. **Pseudo-code for the write barrier we use to maintain the DLG invariant.** It first checks whether the source object is public. If not, then the pointer store cannot be affecting a publication. It then checks whether the target is `NULL` or already public, in which case there's no work to be done. Otherwise the target must be published, which requires setting its public bit and, transitively, those of its children.

Consequently, the set of blocks owned by a thread, i.e. its local heap, may contain public objects but can never contain objects private to another thread.

Periodically, each thread will perform a collection over its local heap, reclaiming space due to unreachable private objects, defragmenting by opportunistically moving reachable private objects, and leaving public objects in place. Occasionally Iso will perform a global collection, reclaiming space due to unreachable objects (public and private), defragmenting by opportunistically moving reachable public objects, and leaving reachable private objects in place.

Iso is built within the most recent version of MMTk [7], which is a high performance Rust-based garbage collection framework with bindings to a number of modern runtimes including OpenJDK, Julia and Ruby. MMTk includes an Immix [9] implementation which we use as the basis for Iso.

4.1 Maintaining the DLG Invariant

Iso maintains the DLG invariant dynamically, using the write barrier illustrated in Listing 1. In OpenJDK there are four broad situations in which an object could be published: i) simple changes to the object graph due to user code updating an instance field (`putfield`) or static field (`putstatic`), ii) changes to the object graph due to the language-level implementation of `arraycopy()`, iii) a reference from user native code through JNI, and iv) changes to internal runtime roots, such as pointers to code objects. As mentioned before, each object has a public bit. The current implementation of Iso in OpenJDK uses side metadata for the per-object public bit. Accessing the bit requires bit arithmetic on the object's address, yielding a bit number and byte address with which the public bit can be tested or set. In principle we could use a bit in the object's header, however this is complicated by OpenJDK's implementation of lock inflation which temporarily moves the object's header word to the stack, making the metadata location conditional on the object's lock state.[3] Augmenting our purely dynamic approach with static techniques [35] is something we may explore in future work.

Putfield and Putstatic. Implementing the barrier in Listing 1 to capture user code changes through `putfield` and `putstatic` operations is relatively straightforward. OpenJDK implements static fields as fields of `Class` objects. As a consequence, its compilers ultimately don't differentiate between `putfield` and `putstatic` when injecting write barriers. Static objects are explicit roots (since they are visible to all threads), so we just set the public bit on static objects at birth and allow the write barrier to publish any private object referenced by them. During our testing, we found that in the

[3]This limitation was addressed by a recent change to OpenJDK 21 [37].

case of nested constructors, the C2 optimizing compiler, unaware of the semantics of our barrier, would occasionally use code motion to lift the barrier above the code that initializes the target object, leading to `publish()` (line 8) being applied to an uninitialized object, triggering a crash. Five benchmarks (cassandra, h2, spring, springjc, and tomcat) are affected by this. Details in Section 5.2.

Arraycopy. A call to `arraycopy()` with reference array arguments should trigger the write barrier. Rather than naively call the standard barrier in a loop, runtimes including OpenJDK typically special-case `arraycopy()` since it is performed in bulk, and depending on the barrier semantics, may be aggressively optimized. For this reason, OpenJDK has an `arraycopy()` barrier API. Unfortunately the API does not expose the base addresses of the source and destination array fields, which we need so that we can locate the public bits. We had to re-engineer this interface to expose the array base addresses.

JNI. References from native heaps are public. The Java Native Interface (JNI) introduces a level of indirection between user-level native code and the Java heap. Updates to JNI fields are well abstracted within OpenJDK so injecting the DLG barrier on JNI stores was straightforward.

Runtime Roots. Runtimes typically hold runtime-specific references into the heap that are not exposed through user-level code or APIs. We observed two broad categories of such roots: i) strong roots — these are enumerated by the runtime at garbage collection time, establishing the liveness of (part of) the heap, and ii) weak roots — these are held silently by the runtime.

In the case of strong roots, the problem we faced was that although the runtime enumerates them at collection time, it does not dynamically track creation or modification of all strong roots. Since these runtime roots are shared and are thus public, the DLG invariant requires that Iso be informed of the pending root creation *before* it occurs. We therefore had to exhaustively identify all instances in the OpenJDK code base where strong root references are created, and ensure that the referent objects are marked as public in order to maintain the invariant. Examples include classloader references and code objects.

In the case of weak roots, the runtime is essentially silently caching object references. These runtime-held references do not affect the liveness of the referent heap objects. There are two considerations for weak roots that affect Iso. First, the runtime may allow other threads to discover the referent objects through its cache. In that case, the referent object is public and must be marked as such. Second, regardless of whether the runtime publishes the referent object, if Iso were to move an object silently referenced by such a runtime cache, it would lead to a break in referential integrity and a crash. Examples include string interning and constant pool caching, which both serve to canonicalize common constant objects.

A surprising false-positive we discovered was that inflated locks may be private. We had reasoned that lock inflation implied a contended monitor, and thus publication, so we included an assertion to ensure that lock inflation only occur with respect to public objects. We discovered that the OpenJDK runtime will inflate uncontended locks during deoptimization,[4] and adjusted our assertions accordingly.

4.2 Allocation and Heap Structure

Iso uses the same allocation strategy and heap structure as Immix [9]. Each thread has a bump allocator which it uses to allocate into 32 KB blocks comprised of 128 × 256 B lines. The block may have been completely empty initially, or it may be *recycled*. If the block is recycled, some lines within the block will be marked, indicating that they contain live objects. The allocator will

[4]Deoptimization is the term used in HotSpot to describe the process of converting a more optimized stack frame into a less optimized stack frame, which is needed to implement speculative optimization.

move monotonically through the block advancing the bump pointer according to the size of the allocation request, skipping over any marked lines. When a block is full, the thread requests a new recycled block. If there are no recycled blocks available, the thread requests a clean block. Iso deviates slightly from Immix because a thread can only consume recycled blocks that contain no objects private to another thread. It does this by maintaining a thread-local list of recyclable *private* and *mixed* blocks and a global list of recyclable *public* blocks. The thread will first use blocks in the thread-local list before attempting to find a block from the global list. Using a public block for thread-local allocation will turn the block into a mixed block. Following the Immix design, objects may straddle lines but not blocks. Iso also follows Immix's design of using a clean overflow block for any allocation that is larger than a line in size. Iso uses MMTk's large object space to allocate objects larger than half a block (16 KB).

4.3 Thread-Local Collection

Iso performs thread-local collections using the owning application thread, not by delegating to a separate garbage collection thread. It uses opportunistic copying, with a copy reserve of two clean blocks made available to the collecting thread. Unlike a conventional global collection, the roots of a thread-local collection are just the references from the thread's stack. Global roots such as statics and JNI roots are not relevant to a local collection since they are by definition public. The DLG invariant ensures that any objects reachable from those global roots will have been published.

The collecting thread performs a transitive closure from its stack. Whenever it encounters a public object, it treats the object as if marked. Whenever it encounters an unmarked private object it will opportunistically copy it unless the copy reserve is exhausted. When an object is copied it leaves behind a forwarding pointer, so that if the object is encountered again later in the trace, the reference can be redirected to the new copy. At the end of the collection, the thread performs a sweep of its local blocks. Blocks with no live objects are returned to MMTk's block manager. Blocks with live private objects are returned to the thread's local recyclable block list, after zeroing freed lines. Blocks that only contain live public objects are now public blocks and so are returned to the global recyclable block list. Recall that local collections cannot move public objects since a public object may be referenced from outside the scope of the collection.

Iso's mixing of private and public objects within local heaps and its use of opportunistic copying distinguish it from prior work, which either make local heaps a logical concept and move nothing [20] or make local heaps spatially distinct and maintain an invariant that they may only contain objects private to the owning thread [18, 19].[5]

Performing local collections within the application thread has two nice properties: i) it avoids the overheads of synchronization between the application thread and a garbage collection thread, and ii) it achieves mutual exclusion with global collection at zero cost. However, it requires careful engineering. Before it can correctly scan its stack, the application thread must be at a GC safe point. Issuing a safepoint yield request would cause the thread to yield correctly but would also yield all other application threads, so is not tenable. Instead we use OpenJDK's thread-local handshake mechanism [21]. Once the thread-local collection starts, the thread will not return to a safe point again until it completes the collection. As a side effect, a global collection will never conflict with any thread-local collections since this mechanism delivers mutual exclusion.

4.4 Global Collection

Thread-local collections cannot collect public objects, so like other thread-local collectors, Iso uses another collector with global scope to collect them. The current implementation of Iso falls back to

[5]This seems to explain why the terms *local* and *private* are often conflated in prior work.

Immix [9] for global collection, but in practice it could use another collector. The primary constraint on the choice of global collector is that it be able to maintain the Block invariant introduced at the start of Section 4 — that no block may contain objects private to more than one thread.

During global collections, Iso maintains the Block invariant straightforwardly — by only moving public objects. Iso achieves this by opportunistically copying public objects and leaving private objects in place. Private blocks are never selected as sources for defragmentation since they never contain objects that are movable in global collections. A more aggressive Iso design might move private objects. Such a design might yield better defragmentation at the cost of complexity and overhead associated with establishing each private object's owner, locating a suitable target block for the copy, and synchronizing with other collector threads before writing into that block.

The restriction that global collections only move public objects limits opportunities for defragmentation. Iso mitigates this by performing thread-local defragmentations at the start of global collections. Threads with large thread-local heaps are targeted for thread-local traces, but unlike local collections, the trace is performed by a garbage collection thread and both public and private objects can be moved. Each local trace is performed strictly by a single thread, making maintaining the Block invariant straightforward since only one thread's private objects can be encountered during the trace. Local defragmentations are followed immediately by the global trace, which moves only public objects and uses forwarding pointers to restore referential integrity with respect to any public object moved during the local traces.

4.5 GC Triggering

Threads trigger local collections at the end of requests if their local heap has reached a configurable size threshold. We found that a trigger of 512 KB worked well among the DaCapo request-based workloads. Each thread has a copy reserve of two clean blocks in which it may copy surviving objects during thread-local collections. Note that the thread-local copy reserve does not impact the total heap budget since it can be drawn from the copy reserve set aside for global collections, since they are mutually exclusive. Iso triggers global collections when the heap budget is exhausted. Each global collection has a configurable percentage of the heap held in reserve for copying, just as Immix does. By default this is 5%.

5 Methodology

5.1 Hardware and Software Environment

Unless otherwise stated, we performed our experiments on AMD Ryzen 7950x machines with 16/32 cores running at 4.5 GHz with a 64 MB LLC, and 4800 MHz 64 GB DDR5 memory. To assess its sensitivity to the hardware platform, we also evaluated Iso on a number of other x86-64 machines, including an Intel Core i9-12900KF Alderlake with 8/16 performance cores and 8/8 efficiency cores running at 3.2 GHz and 2.4 GHz respectively with 12 MB of LLC. We found that the results were consistent across the various platforms we evaluated, so do not discuss the other platforms further. All machines run an Ubuntu 22.04 image with a Linux 6.8.0-40 kernel.

5.2 OpenJDK

We implemented Iso in the new MMTk [7] using its OpenJDK binding, which uses OpenJDK fork jdk-11.0.19+1. We build Iso on top of Immix, which comes with MMTk. We build G1 from the same OpenJDK fork. MMTk's OpenJDK binding lacks support for certain runtime features including class unloading, compressed pointers and weak reference processing. During evaluations, we disable these features in G1 to ensure all three collectors are evaluated using the same runtime features.

As mentioned in Section 4.1, we found that in certain circumstances involving the inlining of nested constructors, OpenJDK's C2 optimizing compiler, unaware of our barrier's semantics, would perform code motion that was unsound, leading to the publication process being applied to an object that was not yet initialized, which would generate a crash. We found that we could work around this on springjc by using a command line flag to avoid optimization of a specific class, but for cassandra, spring, tomcat, and h2, we did not find a simple workaround and had to resort to disabling the C2 compiler. In all four cases, we applied the same command line flags to all three collectors we tested to ensure that they were evaluated using the same compiler configuration. To quantify the impact of disabling C2 for these four workloads, we conducted an experiment where we evaluated the impact of disabling C2 on the bottom line, using workloads that *could* use C2. We found that the impact on the results was insignificant,[6] which gives us confidence in the results we attained for the four benchmarks where C2 was not available. The other benchmarks were unaffected.

5.3 Benchmarks

We use the Chopin-23.11-MR1-RC1 release of the DaCapo benchmark suite [6] and introduce a new benchmark, springjc, which we describe below. All of these benchmarks implement request begin and end callbacks which we use to provide Iso with a collection hint.

As we will show in Section 6.1, and as is evident in Table 1, a careful examination of the DaCapo benchmarks reveals that none are ideal matches to our target workload of application services with large number of quasi-independent requests. Among DaCapo's eight request-based workloads, three (h2, tradebeans, and tradesoap) do not issue isolated requests. Instead, all requests are dominated by computation over a shared in-memory database. As a result, their publication rates are extremely high (68%, 53%, and 43%). The cassandra workload also does not issue isolated requests, leading to an overall publication rate of 33%. The kafka workload is problematic for two reasons. First, it is implicitly singled threaded because of the way it dispatches queries. Second, its request begin and request end callbacks are called from different threads, making attribution of work intractable. The kafka workload is the subject of an open pull request which seems to address some of these issues [57]. The majority of tomcat's work is done in the client, which makes it uninteresting as a server benchmark.

This leaves lusearch and spring, neither of which are ideal as representative of large scale application servers. The lusearch workload is not implemented as a client-server benchmark; it directly runs server queries. The spring workload has a very light server-side workload, querying a small database that models a veterinary practice rather than performing any substantive computation.

Because of these shortcomings, we found it necessary to create our own workload, springjc. It is a compilation server based on the spring benchmark. Instead of querying a tiny database as DaCapo's spring does, springjc uses the standard Java compiler, `javac`, to compile one of 172 small single-file Java source programs, chosen according to the request it receives. It thus exercises the same Spring application framework, which is a modern, widely-used framework [50] and by building upon spring, it benefits from the DaCapo harnessing infrastructure, but conducts a more realistic and substantial workload. We will make our workload available.

[6] Concretely, for lusearch, Iso's improvement over Immix went from 17.4% to 16.8%, while for springjc it went from 44.9% to 46.7%, indicating a 0.6% reduction for lusearch and a 1.8% improvement for springjc.

Table 1. **Benchmark publication characteristics.** The first two columns indicate each workloads' total allocation and allocation rate. The next two columns indicate the publication rate, expressed with respect to bytes allocated and execution time. The fifth column indicates the overhead of adding the DLG-enforcing write barrier. The first nine benchmarks (springjc – tradesoap) are request-based.

	Allocation MB		Publication MB		DLG
Benchmark		**/sec**	**/MB alloc**	**/sec**	**overhead**
springjc	97324	6746	0.21	1398	1.02
cassandra	6700	953	0.33	315	1.10
h2	36793	11654	0.68	7872	3.47
kafka	4543	848	0.20	166	1.00
lusearch	27720	13131	0.01	116	1.08
spring	15588	4813	0.03	131	1.01
tomcat	6360	1736	0.10	168	1.02
tradebeans	1509	2606	0.53	1371	2.48
tradesoap	1229	2415	0.43	1049	1.57
avrora	195	66	0.06	4	1.00
batik	472	383	0.55	210	1.21
biojava	12098	2120	0.00	0	1.04
eclipse	8238	918	0.36	326	1.16
fop	519	811	0.09	75	1.06
graphchi	12297	5208	0.02	107	1.06
h2o	15280	4980	0.37	1858	1.07
jme	178	26	0.05	1	1.00
jython	5359	2324	0.37	848	1.61
luindex	2226	779	0.06	50	1.04
pmd	8464	7077	0.13	946	1.14
sunflow	22155	13575	0.00	0	1.02
xalan	6403	11684	0.20	2324	1.05
zxing	1671	2866	0.06	183	1.01

6 Evaluation

6.1 Overall Publication Behavior and DLG Barrier Overhead

Table 1 quantifies the allocation and publication behavior of springjc and each of the DaCapo benchmarks. The first two columns indicate total allocation and the allocation rate. The next two

Table 2. **Request processing publication characteristics.** The first pair of columns indicate the requests' allocation rate with respect to time and KB per request. The next pair of columns indicates the fraction of requests that are server-side. The next pair indicates median and 90th-percentile of per-request publication rate. The last pair indicates median and 90th-percentile of per-request private survival rate.

	Allocation		Server		Publication		Private Survival	
	MB/Sec	KB/rq	B/all	Obj/all	50%-ile	90%-ile	50%-ile	90%-ile
springjc	6746	12166	1.00	1.00	0.17	0.17	0.00	0.00
cassandra	1339	32	0.50	0.53	0.26	0.40	0.00	0.00
h2	11658	377	–	–	0.63	0.68	0.00	0.00
kafka	916	5	–	–	–	–	–	–
lusearch	13128	54	1.00	1.00	0.00	0.02	0.19	0.19
spring	4812	1948	0.95	0.98	0.21	0.46	0.00	0.00
tomcat	1765	81	0.14	0.24	0.15	0.49	0.00	0.00
tradebeans	2632	85	–	–	0.54	0.63	0.00	0.00
tradesoap	2442	285	–	–	0.30	0.48	0.00	0.00

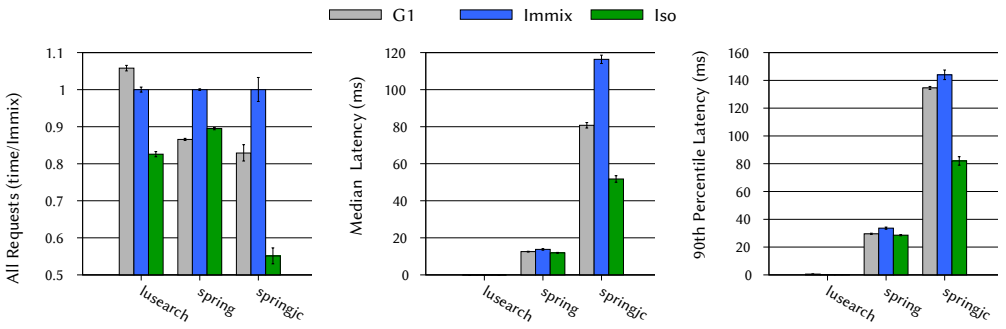
columns indicate publication rate in terms of bytes and time. The top nine benchmarks are request-based. The table shows that some workloads, including lusearch (0.01) and spring (0.03) have very low publication rates. Some of the non-request-based workloads also have very low publication rates, including biojava, graphchi, and sunflow. Four of the request-based workloads have notably high publication rates: h2, tradebeans, tradesoap, and cassandra (discussed in Section 5.3). Workloads such as these are clearly not suitable targets for thread-local garbage collection.

The last column of Table 1 indicates the overhead of the DLG invariant-enforcing write barrier (Listing 1). It measures the mutator overhead of our baseline Immix-based system without any thread-local collection, when publication tracking is enabled relative to when it is not enabled. Note the variability in this overhead. Many workloads, including important ones like spring, springjc, and tomcat have overheads of 2% or less. Note that eight of the fourteen non-request-based workloads have overheads of 7% or less and four of those have overheads of just 2% or less (avrora, jme, sunflow, and zxing). We have focussed exclusively on large application servers with isolated requests, so have yet to explore the applicability of Iso to such promising non-request-based workloads. On the other hand, h2, tradebeans and tradesoap have extremely high overheads, clearly rendering the approach non-viable for such workloads. Nonetheless, this result leads to a key finding of our work: that it is possible to maintain the DLG invariant on large-scale Java workloads in a production JVM at very low overhead for important server workloads such as lusearch (Lucene [27]), spring, springjc (Spring [50]), and tomcat (Tomcat [28]).[7]

6.2 Request Characterization

Table 2 characterizes the behavior of requests among springjc and DaCapo's eight request-based workloads, highlighting the three workloads that are dominated by isolated server-side requests (springjc, lusearch, and spring). The first two columns indicate the allocation rate of the request

[7]Overheads are also low for kafka and cassandra but we exclude these benchmarks for reasons discussed in Section 5.3.



(a) Total request processing time. (b) Median latency (ms). (c) 90th percentile latency (ms).

Fig. 2. **Throughput and latency performance of Iso** at 2.0× the minimum heap in which Immix will run. The left graph shows the elapsed time for the request-processing portion of each benchmark, normalized to our baseline, Immix. The right two graphs show the median and 90th percentile metered latencies in msec. Iso's latency performance ends up being dominated by its fall-back collector, Immix.

processing part of the workloads, first expressed in MB per second and then in KB per request. Workloads with higher allocation rates are more likely to place stress on a garbage collector, so lusearch, h2, springjc and spring are prime targets, however h2's very high publication rate makes it unsuitable for thread-local garbage collection. Note that the amount allocated per request varies by more than three orders of magnitude among these benchmarks, from 5 KB (kafka) to 12.1 MB (springjc). In our experience, some industrial web services readily allocate 10's or 100's of MB per request. The next columns indicate what fraction of the request-time allocation occurs on the server side (versus the client side). This varies greatly from no explicit server-side activity (h2, kafka, tradebeans, tradesoap), to entirely server side (springjc, lusearch). Our work targets server-side workloads, which makes springjc, lusearch, and spring most interesting.

The next pair of columns indicate the per-request publication rate, showing the median (50%-ile) and 90%-ile rates among all of the issued requests. Among the interesting workloads, the rates vary from lusearch's extremely low 0-2% to spring's relatively high 21-46%, while springjc is very consistent with a moderate 17% for both median and 90th percentile. It is interesting to view these publication rates alongside the private object survival rates, which are shown in the last two columns. Notice that although spring and springjc have moderate to high publication rates, they have zero survival rates for private objects. This indicates that the data which is not shared with other threads is crisply bounded in lifetime to the life of the request. On the other hand, lusearch has near-zero publication but a moderate survival rate, which suggests that thread isolation is very strong, and results from requests are handled within the requesting thread, not shared with other threads. Indeed, each lusearch thread directly writes its request results to file rather than handing them off to another thread for processing.

The zero survival rate of private objects revealed in this analysis nicely confirms the hypothesis underlying the motivation for this paper. It shows that the thread-private aspects of these large modern Java workloads do in fact conform with the liveness properties depicted in Figure 1.

6.3 Performance Analysis

Figure 2 shows the throughput and latency performance of the three key workloads. We use Immix as our baseline because the current implementation of Iso is built on Immix, and thus Iso should be

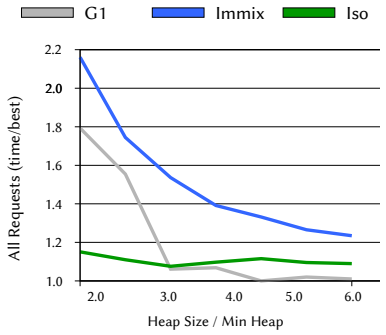


Fig. 3. **Heap size sensitivity for G1, Immix and Iso on springjc.** Iso is substantially more space-efficient than G1 or Immix. As the heap size grows larger, Iso's space advantage is diminished.

seen as an optimization over Immix. An implementation of Iso built on a more performant collector such as LXR should perform better, accordingly. We also show results for G1 since it is the default collector within OpenJDK and prior work shows that OpenJDK's alternatives either significantly compromise latency or throughput [12]. This analysis is conducted in a modest heap, set to 2.0× the minimum heap in which Immix will run.

Figure 2(a) shows the total time for the request-processing segment of each workload.[8] Iso outperforms Immix substantially: by 17%, 10.5%, and 45% in total execution time. It even outperforms G1 substantially on lusearch and springjc: by 22% and 33%. These results make it clear that for workloads such as these, request-private garbage collection is not only viable, but has an impressive advantage over existing approaches. Note that on lusearch, Iso has an 8% overhead due to its DLG barrier (Table 1), but is able to overcome this to outperform the other collectors by 17% and 22%. (The barrier overhead for spring and springjc was just 1% and 2% respectively.)

Figure 3 shows the three collectors' sensitivity to heap size on springjc. Iso is substantially more space efficient than both G1 and Immix, giving it a large performance advantage in modestly sized heaps. As the heap size grows the performance advantage diminishes. Nonetheless, Iso significantly outperforms Immix even at extremely generous heap sizes.

Despite outperforming Immix by 10.5% on spring (Figure 2(a)), Iso lags G1 by 3.4%. We determined that this is due to Immix requiring nearly twice the minimum heap size of G1 on spring. This is an example of Iso inheriting a performance characteristic from the specific collector implementation it is built upon, something that would be addressed if Iso were built upon a more advanced collector.

Figure 2(b) and Figure 2(c) show the median and 90th percentile metered request latency. The DaCapo benchmarks use metered latency to model a server's request queue, whereby an interruption due to the scheduler or a garbage collection not only affects the latency of the running request/s, but also all those in the request queue [12]. Both figures show Iso substantially reducing latency compared to both G1 and Immix, particularly on the more substantial springjc workload. Iso's tail latency would further improve if it used a fall-back collector that performed collections concurrently, rather than simple stop-the-world, whole-heap collection performed by this version of Immix. LXR and V8's collector are examples of advanced low-latency mark-region collectors that could better fill this role [15, 60].

[8]DaCapo separately captures the execution time for the dominant request-based portion of workloads, so that it can be separated from any time the benchmark may take to start the server or perform other such one-time initialization work, etc.

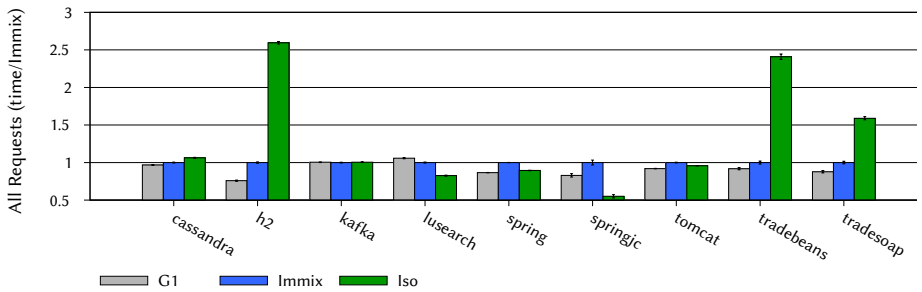


Fig. 4. **Request processing time for all DaCapo request-based benchmarks and springjc.** The lack of isolation and correspondingly high publication rates of h2, tradebeans, and tradesoap and thus their high DLG barrier overhead (Table 1) make them unsuitable for thread-local collection; borne out by these results.

6.4 Other Request-Based Workloads

Figure 4 shows Iso's performance across all nine request-based workloads, again normalized to Immix, and including G1. The three workloads with very high publication rates; h2, tradebeans, and tradesoap all show substantial slowdowns of 2.59×, 2.40×, and 1.59×. For h2, this slowdown is substantially less than the 3.47× slowdown due to the barrier overhead (Table 1), while for the two trade benchmarks, the overheads are about the same as the barrier overhead. Among the remaining three workloads, the overheads are awash, with tomcat showing a 4% win, kafka neutral, and cassandra showing a 6% slowdown.

This reinforces the point that request-based garbage collection is not a general purpose approach. It depends on workloads that issue largely-isolated requests. Workloads that are mostly public, not isolated, such as h2, tradebeans, and tradesoap cannot benefit much from local collections, and pay a significant overhead for dynamic enforcement of the DLG invariant.

7 Discussion and Threats to Validity

Iso demonstrates that request-private garbage collection is not only practical but very effective for suitable workloads. It also demonstrates that opportunistic copying solves the challenge of efficiently implementing request-local garbage collection in the absence of exploitable immutability. The implementation of Iso was a substantial engineering undertaking, and although it is built in a production JVM and can run successfully with large applications and with a rich benchmark suite, there are limitations to our implementation that are important to note.

Generality. While Iso is complete and should run any general-purpose Java workload correctly, it is not designed to be a general purpose garbage collector. This approach is consistent with two of the five production collectors that presently ship with OpenJDK: Shenandoah and ZGC. These collectors specifically target low-latency application domains, but introduce throughput overheads for general purpose programs similar to and higher than Iso's. (Compare the overheads in Table 1 to those in Table VIII of Cai et al. [12].) Iso specifically targets multithreaded, performance-critical application services that run large numbers of concurrent, largely isolated requests. Section 6.4 showed that workloads with unsuitable characteristics will perform very poorly, primarily due to the cost of dynamically maintaining the DLG invariant on such workloads. It seems possible that other techniques such as static analysis could greatly reduce overheads in such cases, but we leave that exploration for future work. Iso's design is not Java-specific. In principle it could be useful in any language runtime used to serve large numbers of concurrent, largely-isolated requests.

Other Collectors. The Iso design is underpinned by a general purpose collector used to ensure public objects are reclaimed efficiently. The initial implementation uses Immix [9], which is a stop-the-world, full-heap collector, and its stop-the-world pauses will affect tail latency. A more advanced implementation of Iso would fall back to a concurrent mark-region collector such as LXR [60] or V8's collector [15]. In our evaluation we compared with Immix, which we built upon, and G1, which is the default OpenJDK performance collector. The LXR collector is a general purpose collector with very good throughput and latency performance. We did not compare with LXR for two reasons. First, LXR has not been merged into MMTk and is separately maintained. The differences between its codebase and ours make comparison difficult. Second, the improvements made by LXR are largely orthogonal to Iso. Ideally Iso would be retargeted to LXR, and the comparison between the LXR-based Iso and LXR would be interesting. The relative simplicity of Immix made it the ideal starting point for this first successful demonstration of request-private garbage collection.

Workloads. Iso is evaluated on three workloads, one which we constructed. The lack of strong preexisting workloads is perhaps a reflection of the lack of existing thread-local collectors for Java. We expect that the promise of this approach will lead to adoption and to other related work, and with it more workloads. In some cases, modest extensions or modifications to existing workloads may make them more realistic and more interesting [57]. Our exploration of the properties of existing DaCapo workloads (Section 6.1, Section 6.2) invites further research on why publication rates vary as much as they do (Table 1), and whether through different software engineering patterns there may be relatively simple mitigations.

OpenJDK and Missing Features. We built Iso on MMTk and evaluate it with MMTk's OpenJDK binding. MMTk's binding currently targets OpenJDK 11, so our evaluation only considers that version. We don't see any reason why moving to the most recent OpenJDK would affect our findings, but of course cannot rule out that possibility. As we mentioned in Section 4.1 and Section 5.2, a bug related to code motion affecting the correctness of the write barrier led us to disabling the C2 optimizing compiler on four benchmarks (cassandra, h2, spring, and tomcat). It is possible that the results for these benchmarks will change once the bug is addressed. We speculate that the main result will be better optimization of the write barrier, which would tend to improve Iso's performance. The MMTk OpenJDK binding is limited by lack of support for class unloading, compressed pointers and weak reference processing. We see no reason to believe that the implementation of any of these missing features will affect the Iso design or implementation, but it's not easy to predict with confidence exactly what impact the addition of any such feature will have.

Lightweight Threads. Many runtimes implement lightweight threads by scheduling them on top of operating system-level threads. As an example, Virtual Threads were recently introduced into the Java Platform [43], becoming available in OpenJDK 21. One of the motivations for Virtual Threads was to make it practical to build thread-per-request services (presently application services typically run requests sequentially within heavyweight threads). A distinct benefit of lightweight threads is that when services are built on a thread-per-request basis, the identification of request boundaries becomes trivial, obviating the need for a request-end callback (Section 3). The thread scheduling mechanism has no impact on the high level algorithm or the concept of thread-local collection. From the standpoint of Iso's implementation, the principal implication is that virtual thread switches will need to also change allocators, so as to maintain Iso's *block invariant* (Section 4). We have not yet implemented support for virtual threads in Iso.

8 Conclusion

We introduce Iso, the first request-private garbage collector for Java. Iso substantially improves the overall performance of suitable large application services, by as much as 32% and 22% compared to the established production garbage collector, G1. The key to Iso is request-private garbage collection, which brings together the longstanding ideas of carefully timing garbage collection to minimize graph traversal costs [38, 54] with thread-local garbage collection [18, 19]. To do this, Iso had to solve the problem of efficient thread-local garbage collection in a language without a type system that exposes immutability. The solution to this problem was to use opportunistic copying, which allows objects to be left in place by the collector when necessary and moved when opportune. Iso's strong performance opens the door to request-private garbage collection and with it, new avenues for memory management research.

Acknowledgments

We are particularly grateful for the inspiration, feedback, and encouragement we received throughout this project from Zixian Cai, Kathryn McKinley, Eliot Moss, Kunal Sareen, and Wenyu Zhao.

References

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, David W. Wall (Ed.). ACM, 174–185. doi:10.1145/207110.207137

[2] Todd A. Anderson. 2010. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, Jan Vitek and Doug Lea (Eds.). ACM, 21–30. doi:10.1145/1806651.1806655

[3] Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1558–1583. doi:10.1145/3591284

[4] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. 2011. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, Jeffrey S. Vetter, Madanlal Musuvathi, and Xipeng Shen (Eds.). ACM, 51–57. doi:10.1145/1988915.1988929

[5] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. 2000. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, Michael B. Jones and M. Frans Kaashoek (Eds.). USENIX Association, 333–346. http://dl.acm.org/citation.cfm?id=1251252

[6] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John N. Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 940–954. doi:10.1145/3669940.3707217

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 137–146. doi:10.1109/ICSE.2004.1317436

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2007. TR-CS-07-01: The DaCapo benchmarks: Java benchmarking development and analysis (Extended Version).

[9] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 22–32. doi:10.1145/1375581.1375586

[10] Hans-Juergen Boehm and Mark D. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.* 18, 9 (1988), 807–820. doi:10.1002/SPE.4380180902

[11] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. 2000. *The Real-Time Specification for Java*. Addison-Wesley.

[12] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 46–57. doi:10.1109/ISPASS55109.2022.00005

[13] Maurizio Cimadamore. 2023. JEP 442: Foreign Function & Memory API (Third Preview). https://openjdk.java.net/jeps/442

[14] CloudFlare. 2024. How Workers works. https://developers.cloudflare.com/workers/reference/how-workers-works/

[15] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. 2016. Idle time garbage collection scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 570–583. doi:10.1145/2908080.2908106

[16] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, David F. Bacon and Amer Diwan (Eds.). ACM, 37–48. doi:10.1145/1029873.1029879

[17] Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. In *OCaml 2014: The OCaml Users and Developers Workshop*. https://anil.recoil.org/papers/2014-oud-multicore.pdf

[18] Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 70–83. doi:10.1145/174675.174673

[19] Damien Doligez and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 113–123. doi:10.1145/158511.158611

[20] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local heaps for Java. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 183–194. doi:10.1145/512429.512439

[21] Robbin Ehn. 2017. JEP 312: Thread-Local Handshakes. https://openjdk.org/jeps/312

[22] Per Lidén et al. 2018. ZGC: The Z Garbage Collector. https://wiki.openjdk.java.net/display/zgc/Main

[23] Yi Feng and Emery D. Berger. 2005. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 workshop on Memory System Performance, Chicago, Illinois, USA, June 12, 2005*, Brad Calder and Benjamin G. Zorn (Eds.). ACM, 68–77. doi:10.1145/1111583.1111594

[24] Robert Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612. doi:10.1145/363269.363280

[25] A. Yu. Filatov and V. V. Mikheev. 2019. Quantitative Evaluation of Thread-Local Garbage Collection Efficiency for Java. *Program. Comput. Softw.* 45, 1 (2019), 1–11. doi:10.1134/S0361768819010043

[26] Christine H. Flood, Roman Kennke, Andrew E. Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, Walter Binder and Petr Tuma (Eds.). ACM, 13:1–13:9. doi:10.1145/2972206.2972210

[27] The Apache Software Foundation. 2024. Apache Lucene. https://lucene.apache.org/

[28] The Apache Software Foundation. 2024. Apache Tomcat. https://tomcat.apache.org/

[29] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. doi:10.1145/512529.512563

[30] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 81–93. doi:10.1145/3178487.3178494

[31] David R. Hanson. 1990. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Softw. Pract. Exp.* 20, 1 (1990), 5–12. doi:10.1002/SPE.4380200104

[32] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 69–80. doi:10.1145/1028976.1028983

[33] Richard Hudson and Austin Clements. 2016. Request Oriented Collector (ROC) Algorithm. https://golang.org/s/gctoc

[34] Ivan Jibaja, Stephen M. Blackburn, Mohammad R. Haghighat, and Kathryn S. McKinley. 2011. Deferred gratification: engineering for high performance garbage collection from the get go. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, Jeffrey S. Vetter, Madanlal Musuvathi, and Xipeng Shen (Eds.). ACM, 58–65. doi:10.1145/1988915.1988930

[35] Richard E. Jones and Andy C. King. 2005. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*. IEEE Computer Society, 129–138. doi:10.1109/SCAM.2005.1

[36] Roman Kennke. 2021. Shenandoah in OpenJDK 17: Sub-millisecond GC pauses. Blog post. https://developers.redhat.com/articles/2021/09/16/shenandoah-openjdk-17-sub-millisecond-gc-pauses#

[37] Roman Kennke. 2022. JDK-8291555: Implement alternative fast-locking scheme. https://bugs.openjdk.org/browse/JDK-8291555

[38] Henry Lieberman and Carl Hewitt. 1983. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (1983), 419–429. doi:10.1145/358141.358147

[39] Simon Marlow and Simon L. Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 21–32. doi:10.1145/1993478.1993482

[40] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. doi:10.1145/367177.367199

[41] Krzysztof Palacz. 2001. JSR 121: Application Isolation API Specification. https://jcp.org/en/jsr/detail?id=121

[42] Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynès. 2002. Incommunicado: efficient communication for isolates. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 262–274. doi:10.1145/582419.582444

[43] Ron Pressler and Alan Bateman. 2023. JEP 444: Virtual Threads. https://openjdk.org/jeps/444

[44] Douglas T. Ross. 1967. The AED free storage package. *Commun. ACM* 10, 8 (1967), 481–492. doi:10.1145/363534.363546

[45] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. doi:10.1145/3408995

[46] Bjarne Steensgaard. 2000. Thread-Specific Heaps for Multi-Threaded Programs. In *ISMM 2000, International Symposium on Memory Management, Minneapolis, Minnesota, USA, October 15-16, 2000 (in conjunction with OOPSLA 2000), Conference Proceedings*, Craig Chambers and Antony L. Hosking (Eds.). ACM, 18–24. doi:10.1145/362422.362432

[47] Peter Styger. 1967. *LISP 2 garbage collector specifications.* Technical Report TM-3417/500/00. System Development Cooperation, Santa Monica. https://www.softwarepreservation.org/projects/LISP/lisp2/TM-3417_500_00_LISP2_GC_Spec.pdf

[48] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: the continuously concurrent compacting collector. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 79–88. doi:10.1145/1993478.1993491

[49] V8. 2024. V8 Isolate Class Reference. https://v8docs.nodesource.com/node-0.8/d5/dda/classv8_1_1_isolate.html

[50] VMWare. 2024. Spring. https://spring.io/

[51] Sam Westrick. 2022. *Efficient and Scalable Parallel Functional Programming Through Disentanglement.* Ph. D. Dissertation. Carnegie Mellon University.

[52] Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement detection with near-zero cost. *Proc. ACM Program. Lang.* 6, ICFP (2022), 679–710. doi:10.1145/3547646

[53] Paul R. Wilson. 1988. Opportunistic garbage collection. *ACM SIGPLAN Notices* 23, 12 (1988), 98–102. doi:10.1145/57669.57679

[54] Paul R. Wilson and Thomas G. Moher. 1989. Design of the Opportunistic Garbage Collector. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA 1989, New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*, George Bosworth (Ed.). ACM, 23–35. doi:10.1145/74877.74882

[55] Christian Wimmer. 2019. Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM. https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e

[56] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29. doi:10.1145/3360610

[57] Xi Yang. 2024. Add multi-producer and inflight request control for the Kafka benchmark. https://github.com/dacapobench/dacapobench/pull/268

[58] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, Martin T. Vechev and Kathryn S. McKinley (Eds.). ACM, 37–48. doi:10.1145/2258996.2259004

[59] Wenyu Zhao and Stephen M. Blackburn. 2020. Deconstructing the garbage-first collector. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci (Eds.). ACM, 15–29. doi:10.1145/3381052.3381320

[60] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 76–91. doi:10.1145/3519939.3523440

Received 2024-11-15; accepted 2025-03-06