Deconstructing the Garbage-First Collector*

Wenyu Zhao
Australian National University
wenyu.zhao@anu.edu.au

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

Abstract

Garbage-First is among today's most widely used garbage collectors. It is used in the HotSpot and OpenJDK virtual machines, and shares algorithmic foundations with three other important contemporary collectors: Shenandoah, C4, and ZGC. However, the design of the core algorithms and the performance tradeoffs they manifest have not been carefully analyzed in the literature. In this work, we deconstruct the G1 algorithm and re-implement it from first principles. We retrospectively develop a concurrent, region-based evacuating collector, CRE, which captures the principal design elements shared by G1, Shenandoah, C4, and ZGC. We then evaluate the impact of each of the major elements of G1 on performance, including pause time, remembered set footprint and barrier overheads. We find that G1's concurrent marking and generational collection reduces the 95-percentile GC pauses by 64% and 93% respectively. We find that the space overhead of G1's remembered sets is very low, typically under 1%. We also independently measure the barriers used by G1 and find that they have an overhead of around 12% with respect to total performance. This analysis gives users and collector designers insights into the garbage-first collector and the other fixed-size region-based concurrent evacuating collectors, which we hope will lead to better use of the collectors and provoke future improvements.

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

Keywords Concurrent garbage collection, Garbage collection, Memory management, Barriers, Java

ACM Reference Format:
Wenyu Zhao and Stephen M. Blackburn. 2020. Deconstructing the Garbage-First Collector. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3381052.3381320

*This work is supported by ARC DP190103367 and Huawei. Any opinions, findings and conclusions expressed here in are those of the authors and do not necessarily reflect those of the sponsors.

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7554-2/20/03.
https://doi.org/10.1145/3381052.3381320

1 Introduction

Garbage-First [12], Shenandoah [15], C4 [35], and ZGC [22] are fixed-sized region-based concurrent evacuating garbage collectors that share a common ancestor in Lang and Dupont's collector from 1987 [21]. However, the underlying relationship among these collectors has not been well-identified or explored. Instead, the algorithms are often presented as independent monoliths rather than refinements and improvements over a largely coherent preexisting family of collectors. Ignorance of the underlying relationships can make collectors harder to understand, mask substantive contributions, and mislead the design of future collectors.

The G1 algorithm has some known pathologies. As an example, Nguyen et al. [25] show that G1 can sometimes consume as much as 50% of a Java program's execution time. Therefore deeper understanding of the collector's algorithmic roots is warranted. The absence of a coherent deconstruction of the core algorithm has also meant that the contribution of the various constituent parts of the collector, as well as each of the enhancements and extensions has not been systematically measured in isolation. These problems inhibit the understanding of the algorithms both for end users and garbage collection researchers, which is a loss, given the importance of this family of collectors.

The key contributions of our work are a decomposition of Garbage-First (G1); the retrospective development of a concurrent, region-based evacuating collector, CRE which embodies key design elements shared by G1, Shenandoah, C4 and ZGC; and analysis of the tradeoffs inherent in G1's design. We highlight the structural relationship between G1 and other fixed-sized region-based concurrent evacuating collectors and we explore those structural relationships in the context of Lang and Dupont's collector and CRE.

We produce a ground-up implementation using the original design of G1 [12] and its current implementation [29] as our guide. We decompose G1 into several key components. We start by implementing a trivial fixed-sized region-based stop-the-world evacuating collector (SIM) and then implement CRE, which adds concurrent tracing, before adding other algorithmic elements, producing six collectors in all, including non-generational and generational variants of G1. Each collector is a refinement, allowing the contribution of various algorithmic elements to be measured.

Based on this first-principles implementation, we conduct a performance analysis of each component of G1. These analyses include the measurement of the GC pause time, barrier overheads and remembered-set footprint.

Our performance evaluation shows that concurrent-marking and generational collection separately contribute to a reduction of 64% and 93% respectively to the 95-percentile GC pause time on the DaCapo benchmark suite. We measure the average remembered-set space overhead at just 0.66%. The write barrier used by G1 for concurrent marking has an overhead of 5.5%. G1's remembered-set barrier has an overhead of 7.7%. When combined two barriers together, G1 has an overall barrier overhead of 12.4%.

The performance analysis of G1 and its components will help users better understand the choices inherent in the selection of particular algorithms and will help GC designers better understand tradeoffs underpinning current designs.

2 Background and Related Work

We now briefly overview background work, including: Lang and Dupont's collector; CRE, a collector we synthesize to capture design elements common to G1, Shenandoah, C4, and ZGC; an overview of G1; and a brief discussion of each of the other collectors.

2.1 Lang and Dupont's Collector

In 1987, Lang and Dupont [21] described a collector that incorporates ideas common to the family of collectors we discuss here. Their collector addresses an important tension between two canonical tracing collector designs of the time. On the one hand, semi-space collectors [8, 14] collect the heap by *evacuating* live objects into a new space, allowing the space from which they came to be reclaimed entirely. This has the advantage of avoiding fragmentation and maintaining good locality, but it requires that half of the heap be kept in reserve to account for the worst-case where all objects survive a collection. Thus semi-space is very space inefficient. On the other hand, mark-sweep collectors [24] *do not move any* objects, and instead leave live objects in place and scavenge dead objects, placing them on free lists for subsequent reuse. However, since they do not move any objects, mark-sweep collectors are subject to fragmentation.

Lang and Dupont achieve much of the benefit of semi-space without needing to reserve half the heap. They do this by dividing the heap into fixed-size regions, leaving one empty region in reserve. At each collection they target one of the live regions for evacuation and the remaining regions are collected using mark-sweep. The collection uses a full heap trace, *marking* objects in non-targeted regions, and *evacuating* reachable target region objects into the reserved region. Their approach reduces the amount of space held in reserve from 1/2 to 1/N, where N is the number of fixed-size regions. They take full advantage of the mark-sweep algorithm by sweeping the non-target regions and placing freed objects on free lists. The algorithm is thus a true hybrid of the semi-space and mark-sweep algorithms.

Although this design breaks the tension between semi-space and mark-sweep, the Lang and Dupont collector has a notable shortcoming: every collection requires a trace of the whole heap—it is not possible to only trace part of the heap. Furthermore, although they discuss concurrent designs, they do not implement a concurrent collector. Mark-sweep and semi-space also suffer these shortcomings.

We refer to Lang and Dupont's collector as L&D.

2.2 CRE: Concurrent Region-based Evacuating GC

We now describe a derivative of L&D, called CRE, which we have synthesized to reflect key design elements of G1, Shenandoah, C4 and ZGC.

CRE inherits two key design characteristics from L&D and adds two more. It: (*i*) uses fixed-sized regions, (*ii*) uses a (concurrent) full-heap trace to identify liveness, (*iii*) reclaims space *strictly* through evacuation (unlike L&D), and (*iv*) preferentially targets high-yield regions (unlike L&D). **These four design elements capture the foundation that G1, Shenandoah, C4 and ZGC build upon.**

Shortcomings of CRE include that: (*i*) it can only reclaim space after a full heap trace (addressed by G1 and C4), (*ii*) it must stop the world to perform evacuation (addressed by Shenandoah, C4 and ZGC), and (*iii*) unlike L&D, it depends entirely on evacuation to reclaim space (G1, Shenandoah, C4 and ZGC all share this trait).

2.3 Garbage First

In 2004, Detlefs et al. [12] developed the garbage-first collector, which is now widely known as 'G1'. The collector has subsequently become the default collector for Oracle's JVM, and is one of the most widely used garbage collectors.

G1 shares the characteristics of CRE outlined above. It mitigates CRE's first shortcoming by supporting a generational mode, whereby young objects can be collected on the basis of remembered sets alone, foregoing the need for a full heap trace. Like CRE, G1 performs its liveness trace concurrently, but it does not perform evacuation concurrently.

As the algorithm's name indicates, the capacity to collect regions in any order introduces the possibility of preferentially targeting for collection those regions that contain the most garbage. Note that L&D can in principle also collect regions in any order, but does not. Targeting garbage contrasts with traditional generational garbage collectors [23, 36], which collect the youngest objects first and can only collect older objects when they also collect those younger than them. The idea of targeting older objects follows a rich line of prior work on age-based garbage collection [11, 34].

Like prior region-based collectors, such as the mature object space collector (also known as the MOS, or the train

Table 1. Relationships between the fixed-sized single-level region collector described by Lang and Dupont [21], our simple concurrent region-based evacuating collector, CRE, and the four collectors widely used today that share that design heritage.

	L&D [21]	**CRE**	**G1** [12]	**Shenandoah** [15]	**C4** [35]	**ZGC** [22]
Heap structure	Single-Level, Fixed-Size, Region-Based					
Primary Liveness	Whole-Heap Trace					
Reclamation	Mixed	Evacuation Only				
Trace	S.T.W.	Concurrent SATB [40]			Concurrent LVB [35]	
Evacuation	Stop-the-world			Concurrent Brooks barrier [7][1]	Concurrent LVB [35]	
Generational GC	Not supported		Supported	Not supported	Supported	Not supported

algorithm) [16], G1 uses remembered sets to track all cross-region pointers. The remembered sets then serve as roots when a region is independently collected; allowing the collector to update and forward the incoming references when it moves objects within the region. Unlike MOS, G1, L&D, CRE and each of the other collectors we address here require a full heap trace to ensure completeness. By doing this, they sacrifice some of MOS's incrementality but gain full flexibility in collection order without compromising completeness.

In addition to ensuring correctness and completeness, G1's full heap trace allows it to accurately track the volume of garbage in each region and thus preferentially target regions that will yield the most garbage. Once the concurrent trace is complete, G1 uses stop-the-world collection to perform evacuation of a target set of regions. The impact of the stop-the-world pause can be controlled to some extent by reducing the number of regions targeted by a particular collection, with a corresponding need to collect more frequently.

2.4 Shenandoah, C4, and ZGC

Shenandoah [15], C4 [35], and ZGC [22] are more recent fixed-size region-based evacuating garbage collectors. Although all of them share the four characteristics of CRE G1 enumerated in Section 2.2, each of them make substantial and interesting innovations, mostly with respect efficiently increasing concurrency. All of them improve over G1 by supporting concurrent evacuation. C4, like G1, is generational, while Shenandoah and ZGC are not. Despite the common algorithmic foundation seen in CRE, each of these collectors has significant innovations, and each of the published descriptions naturally focus on these.

We identify the relationships between the four collectors, L&D, and CRE in Table 1.

The first two rows of the table identify the characteristics that all six collectors have in common. They all use fixed-sized regions in a single-level hierarchy (in contrast to hierarchical region-based collectors such as Immix [6]). All rely on a full heap trace for correctness, in contrast to region-based collectors such as MOS [16] which can independently collect regions without a fully heap trace, and region-based collectors such as RCImmix [32] which use reference counting as the primary liveness criteria.

Aside from L&D, all of the collectors are strictly evacuating, in contrast to region-based collectors such as [13], and [6], which reclaim unused space from regions that are *not* evacuated in addition to recovering evacuated regions. (Those collectors allow bump-pointer allocation into 'holes' within regions that were not fully evacuated, as well as fully recovering space from evacuated regions.)

Aside from L&D, they all use a concurrent trace for primary liveness.

The next two rows of the table identify key differences. CRE and G1 perform evacuation in a stop-the-world pause, while the other collectors perform evacuation concurrently. CRE, G1 and the original Shenandoah use SATB to perform their concurrent trace, while C4 and ZGC use a loaded value barrier (LVB) to perform concurrent tracing and forwarding [35]. C4 builds directly upon the Pauseless GC [10]. ZGC is similar to C4 although completely independently developed. The original Shenandoah algorithm used a Brooks-style read barrier to implement concurrent copying.[1]

2.5 Evaluation of barrier overheads

Blackburn and Hosking [5] and Yang et al. [38] each evaluated a wide range of different read and write barriers on JikesRVM, using the DaCapo benchmarks and the SPECjvm98 benchmarks. Among those evaluated, they measured the overhead of a zone barrier, which G1 uses, and a bit stealing read barrier, which should be a lower bound on C4's LVB barrier. Yang et al. [38]'s evaluation of the zone barrier uses a generational collector with 32 MB region size. Our evaluation uses a non-generational collector with 1 MB region size. Shenandoah has a command line option that removes dependency on barriers for correctness. This mode allows barriers to be selectively enabled and evaluated [9].

3 Implementation

We consider G1 in terms of two major *phases*: marking and evacuation; and three key *algorithms*: concurrent marking,

[1] In April 2019, Shenandoah moved from a Brooks-style barrier to a Baker-style barrier, along the lines of ZGC and C4 [17–20].

Table 2. We implement and evaluate six collectors in the G1 family, starting with a very simple region collector (SIM), adding features progressively to produce non-generational and generational variants of G1 (G1 and G1G).

	SIM	CRE	REM	GEN	G1	G1G
Region Collection §3.1	✓	✓	✓	✓	✓	✓
Concurrent §3.2		✓			✓	✓
Remembered Sets §3.3			✓	✓	✓	✓
Generational §3.4				✓		✓
Pause Time Prediction §3.5					✓	✓

remembered-set-based evacuation, and generational collection. We re-implement G1 from scratch in JikesRVM [1], using the Memory Management Toolkit (MMTk) [2].

Instead of building the collector as a monolith, we build G1 in a modular way based on these deconstructed components, allowing us to better analyze and evaluate the collector and to better understand the relationships among the G1 family of collectors. As shown in Table 2, we start with implementing a simple region-based collector and then add the features step by step to build six collectors in total, the last two of which amount to bottom-up re-implementations of existing, well-known variations of G1.

The following sections describe each of the collectors.

3.1 SIM: A Simple Region-Based Collector

SIM, is a simple stop-the-world variant of CRE, and the foundation on which we develop the other collectors.

Heap Structure and Allocation Policy The SIM heap, like that of L&D, CRE, and G1, is divided into fixed-size regions. The number of regions is a function of heap size and the region size. OpenJDK's implementation of G1 has a target of 2048 regions, selecting a power of two region size between 1 MB from 1 MB to 32 MB depending the command-line-specified heap size [26]. G1's region size can also be set directly via a command line option. We use a 1 MB region, consistent with G1 for heaps < 3 GB. We evaluate the impact of region size in Section 5.3. Like G1, our implementation uses thread-local allocation buffers (TLABs) to allow fast, unsynchronized allocation into regions. Each thread maintains its own bump pointer into its TLAB, and once exhausted, requests more space from a global resource. Consecutive TLAB requests will be serviced by the same region until that region is exhausted. If the heap usage reaches a pre-defined ratio (90%), mutators are paused and the collector starts a collection cycle. SIM collections are fully stop-the-world.

Marking Like G1, SIM determines liveness via a full heap trace. Staring from roots, it performs a trace, marking objects as live. Then some number of regions are selected for evacuation and marked objects within them are copied out, yielding free space. Following G1's implementation, mark state is held in a side bitmap, using one bit per word of allocated memory. Unlike CRE and G1, SIM performs the marking phase using a stop-the-world collection.

Collection Set Selection During the marking phase, the collector calculates the space consumed by live objects in each region. The collector uses this data to perform collection-set selection. It sorts all regions in ascending order by their occupancy. Then the collector creates a collection set starting with the regions with the smallest live size and continues while the volume of selected live objects is not larger than the remaining available memory in the heap (into which the objects will be copied). SIM, CRE and G1 perform collection set selection while the mutators are stopped.

Evacuation and Reference Updating Finally, SIM evacuates live objects from regions in the collection set. Unlike G1, SIM's evacuation phase is achieved by performing a (second) stop-the-world full heap trace. This means that SIM does not require any write barrier. Later, we use this simplification to precisely measure the cost of the write barrier and its various functional components, while remaining otherwise faithful to the G1 algorithm.

Whenever the collector encounters a reference to an object within a region in the collection set, it checks the referent's header to determine whether it has already been copied. If the referent has not been copied, the collector copies it, marks the old object's header as copied, and leaves behind a *forwarding pointer* indicating the location of the new copy. If the collector encounters a reference whose referent has been copied, it uses the forwarding pointer to fix the reference to point to the new copy. At the end of this trace, all selected regions have had their reachable contents evacuated and pointers to them have been redirected. The selected regions are thus empty and can be reclaimed for use by the mutator.

At the end of this phase, the collection is complete and the mutator resumes.

3.2 CRE: Concurrent Marking

The CRE collector replaces SIM's stop-the-world mark with a concurrent marking phase. Like G1 and Shenandoah, CRE uses the snapshot-at-the-beginning (SATB) algorithm to perform concurrent marking [40]. The intuition behind SATB is simple. Once an object becomes garbage, it will remain garbage. It is therefore correct to use a stale snapshot of the heap state to identify garbage. The SATB algorithm works by intercepting any attempt to overwrite references while the SATB trace is in progress, ensuring that it sees a consistent view of the now-stale snapshot. The overwritten reference is included in the trace by marking its referent as live. The algorithm thus simply requires a write barrier that captures the old reference before it is overwritten and adds that reference to the trace. Objects allocated during the SATB trace are considered live.

```
1 @Inline
2 public void SATBBarrier(ObjectReference src,
3                          Address slot) {
4   if (barrierActive) {
5     ObjectReference old = slot.loadObjectReference();
6     if (!old.isNull()) satbbuf.insert(old);
7   }
8 }
```

Figure 1. The snapshot-at-the-beginning (SATB) barrier used by G1 [12, 30]. When the barrier is enabled, for every non-null reference that is overwritten, the barrier will remember the object.

Initial Mark Pause The snapshot-at-the-beginning algorithm starts with an initial mark pause which stops the execution of all the mutators. During this pause the collector only scans the roots and marks all of the root-reachable objects.

Concurrent Marking During concurrent marking, the mutators resume and the collector marks objects concurrently. Since the roots have already been scanned, the collector simply continues the full-heap tracing concurrently to mark all reachable objects in the heap. The concurrent marking process stops when either the full-heap tracing is finished or the mutators exhaust memory. In the second case, since marking is not yet completed, but mutator allocation cannot continue, the mutators are paused and the collector completes the collection in a stop-the-world marking phase.

Snapshot-At-The-Beginning Barrier Although the intuition behind SATB is a snapshot of the whole heap, directly taking a snapshot is not practical. Instead, SATB uses a write barrier to preserve a virtual snapshot. Consider the execution of a field assignment `obj.x = y`. If `obj.x` contained the last heap reference to the referent object, `z`, then `z` will become disconnected from the heap. However, `z` could be held in a root and subsequently reconnected to the heap. If the newly referencing object had already been traced, then `z` would not be marked and the trace would be incorrect. This situation violates the *snapshot-at-the-beginning* principle. Each object to which a reference is overwritten during the collection must be conservatively marked as live since it may have been reachable from the roots. To deal with this problem, the SATB algorithm introduces a write barrier; a code fragment inserted before each pointer update, including each `putfield` or `aastore` bytecode. Figure 1 shows the write barrier used by G1 and our implementations. When the barrier is enabled (`barrierActive`), for all non-null overwritten references, the barrier remembers this object by inserting it into a buffer. The trace terminates once all entries have been consumed. We additionally implemented a conditional variant of the barrier which avoids duplicate `satbbuf` entries by using a bit in the object's header to record whether it has been added.

Final mark pause During the final marking pause, the collector drains the SATB buffer and marks all remaining objects. By using the SATB algorithm, the collector can perform most of the marking work concurrently, greatly reducing GC pause times. The SATB algorithm is inherently conservative since it can only reclaim objects that were dead at the start of the collection, leading to floating garbage.

3.3 REM: Remembered sets

The next refinement to the base algorithm is to avoid a full heap trace during the reference updating phase by using remembered sets. G1 uses *remembered sets of card tables* to track references into regions [12]. For each region T, it maintains a *set* of card tables.

Classically, a card table is a data structure that for a heap divided into power-of-two-sized *cards*, records for each card whether the card should be scanned for potentially interesting pointers [37] at the next minor collection. In the simple implementation, each time a pointer is written, a bit corresponding to the card on which the pointer resides is set.

G1's implementation is more interesting. For each region T, G1 maintains a *set* of card tables—one card table for each region S that contains pointers into T. Each card table in the set is a bit map, with one bit for each 2^n (512) byte card within the 2^m (1 M) byte region, S. If a bit is set, it indicates that the corresponding card in S at some point contained an incoming reference into the region T, so must be checked at collection time. Additionally, as an optimization, G1 maintains a global card table which it uses to filter the work of processing pointer updates into its per-region sets of card tables.

This differs from classic card marking [37], which keeps a single *global* card table, and differs from classic remembered sets which remember *pointers* into regions (rather than remembering a card table for each region S that contains incoming pointers to the region T). We follow the original G1 design faithfully in each of the collectors that use remembered sets (REM, GEN, G1, and G1G).

Figure 2 shows the data structure used by the remembered sets. A remembered set consists of a list of `PerRegionTables`. Each `PerRegionTable` is a bitmap responsible for remembering cards within a specific foreign region.

Remembered Set Barrier The remembered set barrier is additional to the write barrier presented in Section 3.2 (Figure 1). Detlefs et al. [12] inject the SATB barrier before the pointer store operation, and inject the remembered set barrier after the pointer store barrier. Figure 3 demonstrates the remembered set barrier described by Detlefs et al. [12], the current implementation in OpenJDK, and our implementation in JikesRVM. Note that unlike the classic unconditional card marking barrier [37] this barrier is *conditional*, filtering first with a zone barrier [34, 38] and then with a check of the global card table. For each object reference field assignment `obj.x = y`, the barrier checks whether `obj` and `y` belong to

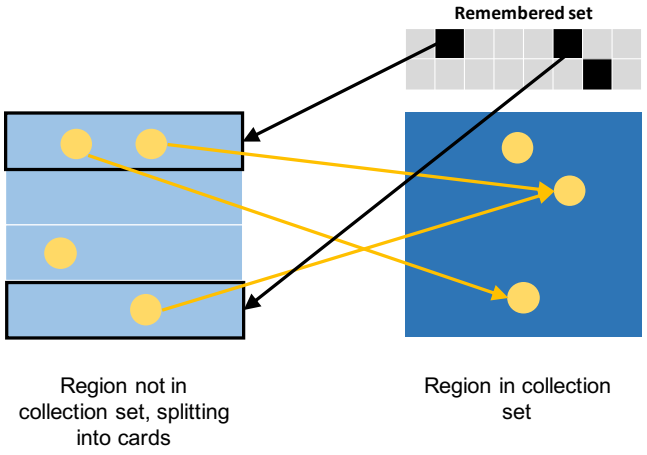

Figure 2. The remembered set data structure used by G1 and REM. Each collection region (right) has a remembered set of cards which may contain pointers into that region [12].

different regions. The barrier uses a single `XOR` operation to check for such cross-region pointers. When a cross-region pointer is created, the barrier tries to mark the global card containing the source (`obj`) as dirty. If the global card has already been marked as dirty, the barrier does nothing more. Otherwise, the barrier pushes the address of the card to a local `dirtyCardBuffer`. The local dirty card buffer has a fixed size of 256 entries. When this buffer is filled, the barrier pushes the buffer to a global `filledRSbuffers` queue [12].

Note that while OpenJDK's compiler implements the barrier as depicted in Figure 1(b), OpenJDK's interpreter implements a variation: Instead of filtering cross-region pointers, this variant simply jumps to the slow path and performs conditional card marking [29]. We implement and evaluate both variations.

Concurrent Remset Refinement When the size of the global `filledRSbuffers` queue reaches a threshold of five `dirtyCardBuffers`, a remset refinement thread is started to concurrently consume the global `filledRSbuffers` and process each `dirtyCardBuffer`. The refinement thread processes each of the cards in each `dirtyCardBuffer`. (Because the filtering performed through the global card table is unsynchronized, duplicate `dirtyCardBuffer` entries are possible.) If the card is marked in the global card table, the thread clears the card's mark, and then linearly scans the card for cross-region pointers and for each of them marks the card in the corresponding card table within the remembered-set.

Hot Card Processing Throughout the continuous process of concurrent remset refinement, some cards may be enqueued and scanned multiple times. To avoid redundant scanning, a hotness value is assigned to each card. Every scan of a card increments its hotness value. When the hotness value for a card exceeds a threshold (the default is four), this card is considered a "hot card". Hot cards are pushed into a separate hot card queue and the processing of all hot cards is delayed until the start of the evacuation phase.

GC Evacuation Phase By using remembered sets, the collector can evacuate objects from any region or set of regions without performing full-heap tracing. During the evacuation phase, the collector scans root objects, linearly scans the cards in remembered-sets, and recursively evacuates all reachable objects within the selection set. All external pointers pointing into the collection-set are guaranteed to be recorded in the remembered sets or the root set. The GC pause time due to evacuation can thus be controlled.

3.4 GEN: Generational Collection

GEN supports generational collection, following the design of G1. It exploits the weak generational hypothesis that says that newly allocated objects die quickly [23, 36]. The collector classifies the regions into three generations: the eden, survivor, and old generations. Regions within the eden generation contain objects that were allocated since the last collection. Regions within the survivor generation contain objects that survived one collection since allocation. Regions within the old generation contain objects that survived collections of the survivor generation. During allocation, newly allocated regions are marked as eden regions. When the ratio of the total number of eden and survivor regions exceeds a `newSizeRatio` threshold, a young collection is triggered, which targets (only) regions in the eden and survivor generations. During a young collection, live objects in eden regions are evacuated to survivor regions and objects in survivor regions are evacuated to old regions.

Young collections simply evacuate objects, and do no global marking work. Instead of determining the liveness of objects by marking, during young collections the collector considers all objects that are not in the collection set to be live. The collector simply starts from the root objects and remembered-sets to recursively evacuate all of the reachable objects in the collection set.

When the committed memory in the heap exceeds some threshold (the default is 45%), generational G1 (and GEN) will initiate a concurrent marking phase for a mixed collection. When the available memory is almost exhausted during concurrent marking, G1 switches to a stop-the-world full GC to continue the unfinished collection work (GEN does the same).

3.5 Pause Time Predictor

Following the design of G1, we add a mechanism to predict and control the GC pause time. The predictor is based on an assumption that the pause time is (approximately) proportional to the size of the collection-set, so controlling the size of the collection-set will change the collector pause time. With a concurrent marking phase, the predictor is principally

```
rTmp := rX XOR rY
rTmp := rTmp >> LogOfHeapRegionSize
// Below is a conditional move instr
rTmp := (rY == NULL) then 0 else rTmp
if (rTmp == 0) goto filtered
call rs_enqueue(rX)
filtered:
```

(a) Detlefs et al. [12].

```
if (new_val == NULL) return
xor_res = addr XOR new_val
xor_shift_res =
  xor_res >> LogOfHeapRegionSize
if (xor_shift_res != NULL)
  goto slow_path
return
```

(b) OpenJDK[27, 28]

```
@Inline
void remsetBarrier(
    ObjectReference src, Address slot,
    ObjectReference ref) {
  if (!(src.xor(ref).rshl(LOG_REGION).isZero())) {
    markAndEnqueueCard(src);
  }
}
```

(c) Our implementation

Figure 3. Three versions of the fast path of the remembered set barrier used by G1. The first is verbatim from the original article [12]. The second is pseudo-code reflecting the current OpenJDK implementation [27, 28]. The last is our implementation.

concerned with controlling the pause time of the stop-the-world evacuation phase. The task of these phases are fixed: dirty card refinement, linear scan of remembered-set cards and object evacuation.

We implement the Detlefs et al. [12] prediction equation for our pause time prediction:

$$T_{CS} = T_{fixed} + T_{card} \times N_{dirtyCard} + \sum_{r \in CS} (T_{rs\ card} \times rsSize(r) + T_{copy} \times liveBytes(r))$$

Where

T_{CS} is the total predicted pause time
T_{fixed} is the time of all extra work involved during the GC pause
T_{card} is the time for linear scanning a card for remembered set refinements
$N_{dirtyCard}$ is the number of dirty cards that have to be processed before evacuation
$T_{rs\ card}$ is the time to linearly scan a card in the remembered-set for evacuation
$rsSize(r)$ is the number of cards in the remembered-set of region r
T_{copy} is the time for evacuating a byte
$liveBytes(r)$ is the total live bytes for evacuation

By examining the remembered-sets and using the pause time prediction model at the start of each GC cycle, the collector can choose the size of the collection-set to meet a user-defined soft pause-time goal.

Pause time prediction for young GCs G1 modifies the pause time prediction model to support generational collection, and we do the same for GEN. For young collections, following OpenJDK's implementation [31], the predictor estimates the remembered set size, dirty cards and surviving bytes for each nursery region. Then the predictor controls the size of the young generation (young regions + survivor regions), based on the previously estimated data and the formula discussed above. For mixed collections, just as in non-generation G1, the predictor still controls the collection-set size to meet the pause time goal, as discussed above. For full collections, the collector does not try to meet the pause time goal and always collects as many regions as possible.

4 Evaluation

We measure the performance of G1 using our JikesRVM [1] implementation, considering three key metrics: remembered set footprint, GC pause time, and barrier overhead.

4.1 Methodology

We now describe the methodology used throughout our evaluations.

Benchmarks We use 19 benchmarks from the DaCapo [3] and SPECjvm98 [33] suites as well as the pjbb2005 benchmark [4]. We omit some of the DaCapo benchmarks because they do not run on JikesRVM.

Operating System and Hardware All of our experiments are run on machines with identical software configurations, each running Ubuntu 18.04.2 LTS (Linux 4.15.0). We ran all experiments on an Intel i7-6700k (Skylake) four-core processor running at 4 GHz with a 8 MB LLC and 16 GB of DDR4 RAM. To assess microarchitecture sensitivity, we also ran our barrier experiments on an Intel Xeon E3-1270 (Sandy Bridge) four-core processor running at 3.5 GHz with an 8 MB LLC and 4 GB of DDR3 DRAM.

JVM and Replay Methodology We base our implementations on JikesRVM at commit `087d300e4` (February 2018). Although JikesRVM only supports a 32-bit address space, we established that we could run with usable heap sizes up to 2 GB. We use JikesRVM's *warmup replay* methodology as refined by Yang et al. [38], to remove the non-determinism from the adaptive optimization system. Before conducting the final evaluations, we run each benchmark 10 times and collect run-time compiler optimization profiles from the fastest invocation. When we perform the final evaluation, we first execute each benchmark once, unoptimized, to resolve all the classes and warm up the JVM. Then we apply optimizations according to the profile gathered off line, before starting a second iteration of the benchmark, which we measure. The warmup replay method keeps the uncertainty of JikesRVM's compiler and optimizer to a minimum, and ensures that

optimizations are aggressively and uniformly applied. We evaluate each configuration 20 times (invocations) and report the min, max and average for each measurement.

4.2 GC Pause Time

We evaluate GC pause time for several variants of the collectors. The complexity of the G1 implementation, which includes concurrent helper threads, makes it difficult to break down the pause-time impact of every element of the collector design. Here we measure the pause time for the following four collectors:

- SIM: Stop-the-world marking + no remsets - non generational
- CRE: Concurrent marking + no remsets - non generational
- REM: Stop-the-world marking + remsets - non generational
- GEN: Stop-the-world marking + remsets + generational (fixed 15% nursery ratio)

G1's pause time prediction algorithm always attempts to control the GC pause time to meet a pre-defined pause time goal. Here, we turn the pause time predictor *off* to expose the impact of the underlying GC on pause time, in isolation.

For each benchmark, we collect the GC pause time data for each mutator thread for four different heap sizes, from small to large. Note that we use the same set of *fixed* heap sizes for all benchmarks because the elements we evaluate are affected by absolute heap size rather than benchmark-specific, relative heap size. To measure latency, we attach a high resolution timer to each mutator to record the duration between when the mutator starts waiting for a collection and when it resumes after collection. We log the times and analyze them after the benchmark has completed. We summarize the results to report the median, average, maximum and 95%-percentile GC pause times.

4.3 Remembered Set Footprint

As stated in Section 3.3, the remembered set data structure we implement follows the original G1 design. Here we evaluate the remembered set footprint and compare the results from the original G1 GC paper [12]. Note that the worst-case space overhead has a quadratic complexity with respect to the total region count within the heap, since each region maintains a remembered set of card tables, one for each referring region.

We record the peak footprint value for each GC cycle. The highest space overhead happens after all the dirty cards are processed, before the evacuation phase of a GC. At this point, no card is marked as dirty, thus the remembered-set reaches its largest overhead within the current GC cycle to remember all necessary cards.

We define the *remset footprint ratio* as the ratio of the memory allocated for remembered sets versus the total committed memory at a specific execution point. This reflects the proportion of the heap that remembered sets consume.

$$\text{Remset footprint ratio} = \frac{\text{Committed memory for remsets}}{\text{Total committed memory}} \times 100\%$$

Note that G1 has a remembered set for each region. Also, the remembered set size for each region depends on the number of regions that contain pointers into the target region. Such a remembered set structure means that different region sizes may significantly affect the remembered set footprint. Therefore, we also evaluate the remembered set footprint for different region sizes, ranging from 64 KB to 8 MB, to analyze the impact of region size on remembered set footprint.

4.4 Barrier overhead

We measure both barriers used by G1, using the method introduced by Yang et al. [39]. We use SIM to measure these barriers. We do this for two reasons. First, following Yang et al. [39]'s method, we choose this collector because although it shares structural and algorithmic elements with G1, it does not require any barriers, which allows us to add and remove the barriers without affecting the correctness of the program. Second, this collector has no concurrent helper threads (such as the marking thread and remset refinement thread), so we can measure the barrier overheads in isolation, without interference from these other sources. We apply each of the barriers to SIM, and compare the mutator time to SIM without the barriers. This approach is similar to the one outlined for Shenandoah [9].

SATB Barrier This barrier is used for concurrent tracing. It first checks the `barrierActive` flag, loads the to-be-overwritten value, and if non-null, unconditionally enqueues it to a thread-local SATB buffer [30] (We call it **Unconditional SATB Barrier**). Note that this barrier can result in duplicate SATB buffer entries. We considered and evaluated a variation of the SATB barrier (the **Conditional SATB Barrier**) that checks whether the to-be-overwritten value has already been enqueued (via atomic test and set of a logging bit in the header), and if so avoids enqueuing a duplicate.

Remset Barrier The fast path of the remembered set barrier uses an `XOR` operation to filter out intra-region pointers (§3.3 and Figure 3). The slow path checks to global card table and if unmarked, it pushes the address of the relevant card to the `dirtyRSBuffer`. For this measurement, we modify the barrier so that it never triggers a concurrent remset refinement, which would confound the measurement and is not part of the barrier itself.

Unfiltered Remset barrier In OpenJDK's implementaton of G1, the interpreter uses the card-marking barrier without the XOR [29] filter. We build a separate GC to measure the

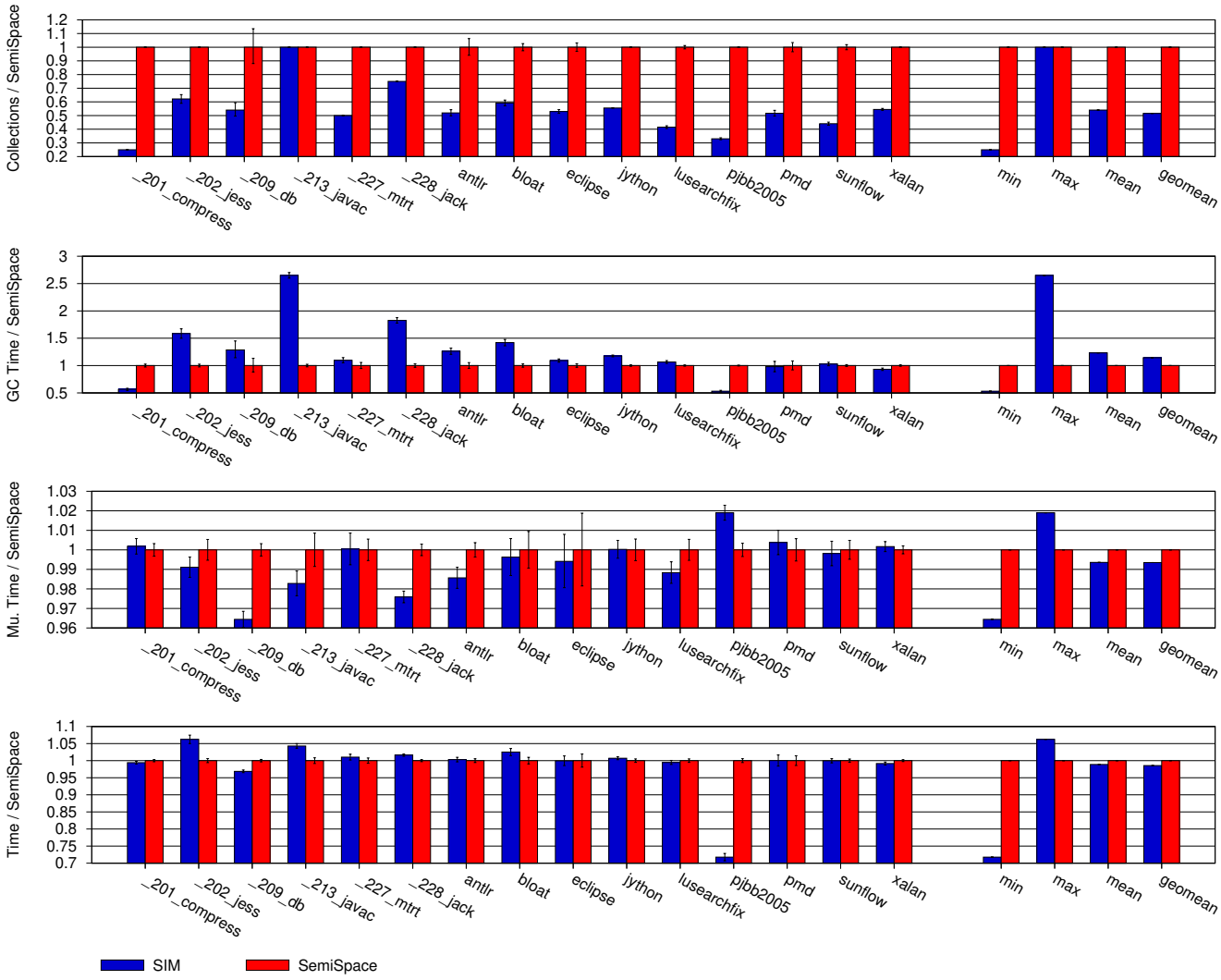

Figure 4. Collection count (top), collection time, mutator time, and total time (bottom) for SIM, normalized to semi-space [8, 14]. SIM strips away concurrency and remembered sets, but retains the core region collection algorithm of G1. Performance is evaluated at 2.5× the minimum heap size. SIM's reduction in number of collections is offset by more expensive collections.

unfiltered remset barrier overhead and compare it with the (default) filtered barrier.

Combined barriers We also combine the SATB and remset barriers to measure the overall barrier overhead of G1's barriers.

For the barrier evaluations, we use both Sandy Bridge and Skylake microarchitectures. We limit the number of processors to one to reduce the uncertainty caused by CPU task scheduling. We report the mutator time only, normalizing to the base case with no barrier. We also evaluate the impact on the zone barrier overhead of varying region sizes.

5 Results

We start with an evaluation of the SIM collector, which is the most basic foundation of the G1 design. We then report and analyze the three key dimensions of the G1 design: pause time, remembered set footprint, and barrier overheads.

5.1 The Simple Region-Based Collector

Figure 4 shows the performance of the SIM collector, compared to a semi-space collector, which is the canonical evacuating garbage collection algorithm [8, 14], and thus the most basic point of comparison for a simple copying algorithm such as SIM. Recall that SIM has all of the fundamental elements of G1, but strips away concurrency and performs a full-heap evacuating trace rather than using remembered sets. This evaluation thus exposes the fundamental costs of

	Conc. Mark	Remset	Gen	Pause-time Predictor	Mean				Median				95%				Max			
					256	645	1125	1696	256	645	1125	1696	256	645	1125	1696	256	645	1125	1696
SIM					34	61	76	88	29	41	69	93	61	153	189	212	75	356	201	224
CRE	✓				5	11	11	13	1	3	3	4	22	95	111	120	49	360	348	372
REM		✓			43	91	114	142	37	78	98	133	73	210	290	402	102	327	338	411
GEN		✓	✓		7	10	10	10	6	7	7	7	16	33	28	27	70	87	174	246
G1	✓	✓		✓	4	5	5	7	1	1	1	2	15	22	29	42	114	161	205	320
G1G	✓	✓	✓	✓	4	5	13	16	2	3	7	8	14	14	59	59	27	52	137	116

Table 3. Pause times in milliseconds for four variants of G1 collectors we construct, measured over 20 invocations of nineteen benchmarks and reported with respect to four heap sizes. The pause-time goal for G1 and G1G collector is set to 40 ms.

such region-based collectors, which are otherwise hard to measure because they are performed concurrently.

We trigger the SIM collection when the heap is over 90% full. Figure 4 shows a geomean 48% decrease in number of collections but a 14.3% increase in collection time compared to semi-space. The sharp reduction in number of collections is because SIM does not need to hold half the heap in reserve. This is more than outweighed by more the inherent overheads associated with tracing through each live object twice: once for the initial mark, and again for evacuation, leading to a net increase in total collection time. This observation is important because while much of this overhead can be obscured by concurrent tracing, it is inherent to all collectors we study here. On the other hand, the mutator performance of SIM is better than semi-space, resulting in a net performance advantage. The reason for the better mutator performance is better locality. SIM has slightly better last-level cache mutator performance and 10% better L1D cache mutator performance (not shown). While both collectors use a bump pointer allocator, each time they collect, they move objects and displace hot data from the cache. Not only does semi-space perform nearly twice as many collections, but it also copies all objects at every collection, which is substantially more disruptive than SIM, which only moves about 10% of the objects at any collection. SIM leaves objects in their allocation order for longer than semi-space.

5.2 GC Pause Time

Table 3 shows our evaluation of six collector variants with respect to pause times. As discussed in §4.2, the pause time predictor is not implemented in the first four collectors, which increases determinism and exposes the underlying behavior of the collectors. G1 and G1G both have the pause time predictor enabled with a pause time goal of 40 ms. We perform the evaluation on four logarithmically increasing heap sizes: 256 MB, 645 MB, 1125 MB and 1696 MB. Note that each data point in the table corresponds to a statistic with respect to 20 invocations of all nineteen benchmarks. We produce each data point by concatenating the pause time data from all 20 invocations of all nineteen benchmarks, and taking the relevant statistic.

First, we note that the pause times generally grow with heap size. This is explained in the case of SIM, REM, and GEN because they trace the entire heap at each stop-the-world collection. CRE does not, but it performs evacuation in the stop-the-world phase, which is also heap size-dependent. By comparing SIM and CRE, we see that concurrent marking reduces the 95-percentile pause time by around 41%–64% on four measured heap sizes. Unsurprisingly, the results highlight the importance of concurrent tracing in reducing pause times.

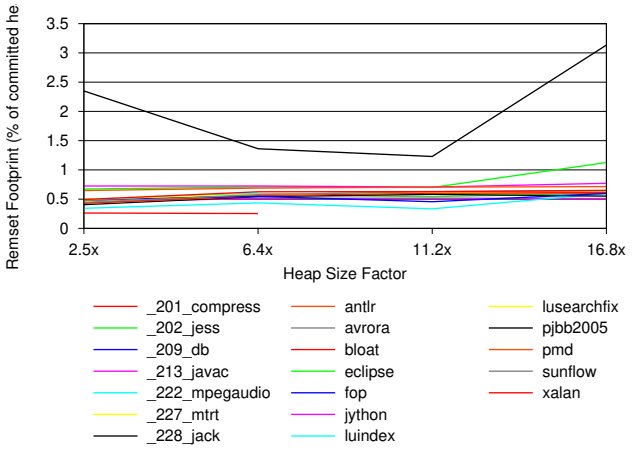

Figure 5. Remembered set footprint for non-generational G1 (REM), for four heap sizes, as a percentage of heap size. This shows the percentage of memory used for the remembered sets versus total committed memory. Most benchmarks use under 0.6% of the heap. pjbb2005 has the highest footprint, but remains within 1.5%. jython is the next highest footprint benchmark, at around 0.7%.

However, the results for SIM and REM show that using remembered sets for evacuation instead of full-heap tracing increases the 95-percentile pause time by 20%, 37%, 53%, and 90% respectively on four measured heap sizes. The remembered set scanning overhead is proportional to the heap size. Note that these collectors only perform collection when the heap is over 90% full and always collect as many regions as possible. This result highlights that the overhead of remembered set scanning can be significant, and will be exposed if too many regions are collected. These results motivate generational collection and avoiding full GCs.

By comparing REM and GEN, we see a 95-percentile pause time reduction by 78%, 84%, 90%, and 93% respectively on

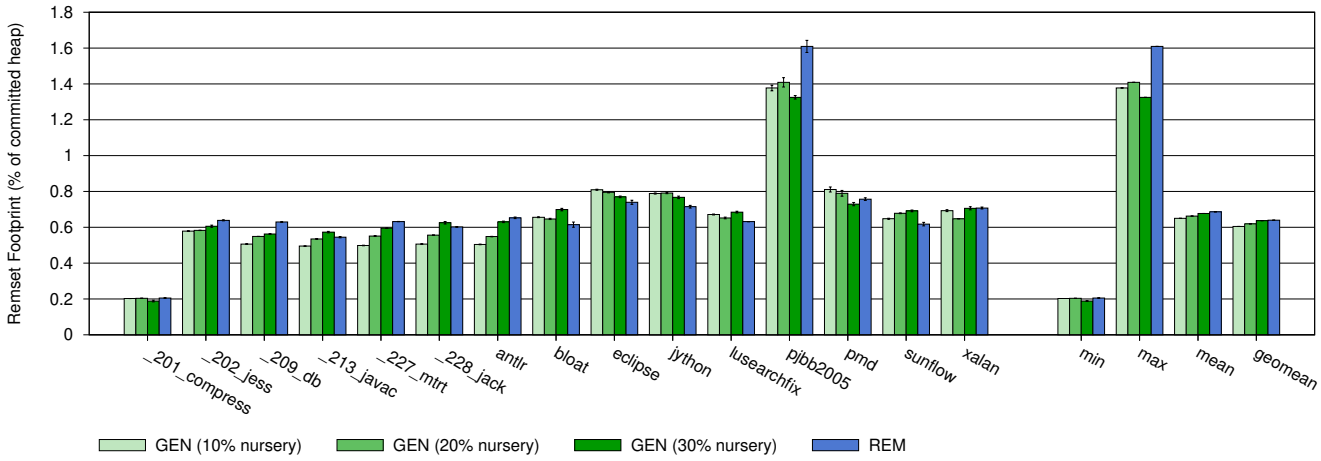

Figure 6. Remembered-set footprint for GEN and REM.

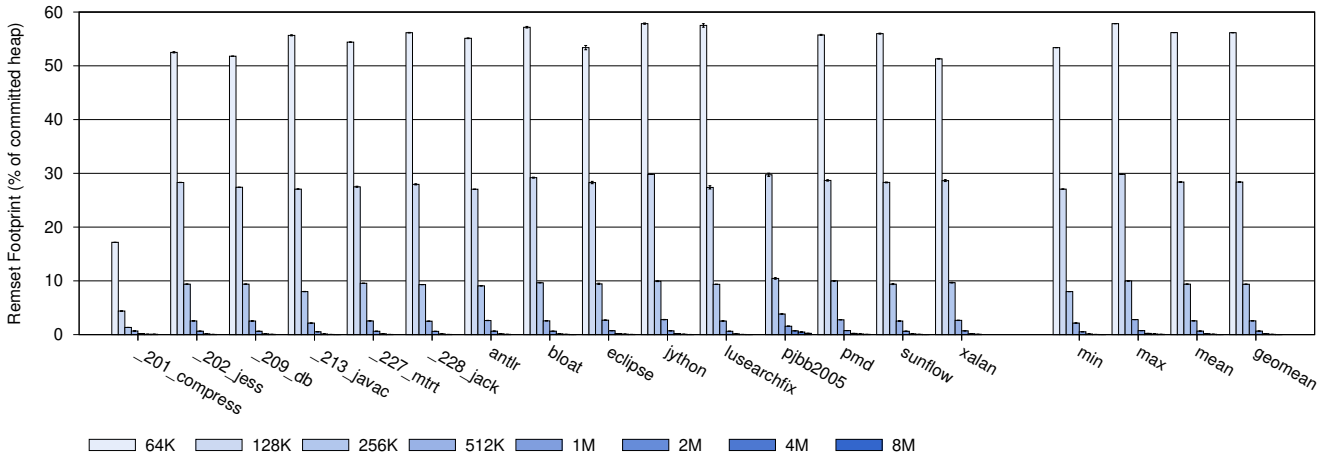

Figure 7. Remembered-set footprint for REM as a function of region size. At 64 KB, the overhead is so high that pjbb2005 does not run to completion.

the four measured heap sizes simply due to the introduction of a generational policy. We also note that as the heap size grows, the overhead of full GC remembered set scanning increases, and the collector will benefit (with respect to GC pause time) by performing generational collection.

The last two rows of Table 3 reveal that with the pause-time prediction feature turned on, both G1 and G1G attempt to control the pause time within the pre-defined 40 ms pause-time goal. Both collectors do well with respect to mean and median pauses, and with three exceptions (42 ms, 59 ms, 59 ms) stay within their goal at the 95th percentile. However, their maximum pauses all exceed the goal, and in the case of G1, its maximum pauses are the worst of the six collectors studied at the smallest heap size. Some of this is due to remembered set processing (which is performed in a stop-the-world setting), also seen in the results for REM.

5.3 Remembered Set Footprint

First, we measure the footprint on the REM and GEN collectors, fixing the nursery size for GEN at 10%, 20% and 30%. As shown in Figure 6, all four scenarios have geomean footprints of around 0.6%. For generational G1, the footprint increases slightly as the nursery size ratio increases. In our measurements, the lowest geomean footprint (0.61%) occurs when the nursery ratio is 10%. REM has the highest overhead of 0.64%. This result highlights the space efficiency of G1's remembered set design at the heap sizes we evaluate. Note that the quadratic nature of remsets means that the footprint may grow considerably at heap sizes that are significantly larger than the ones we evaluate.

Next we evaluate the impact of heap size on remembered set footprint. Figure 5 shows the footprint of remembered sets as a function of heap size for each benchmark for REM. The region size is fixed to 1 MB. All but two benchmarks have

overheads under 0.7%. The highest among them is pjbb2005 with a peak overhead of just 3.1%. The remembered set footprint does not have notable increase as the heap size increases.

Region size and remembered-set footprint In Figure 7 we evaluate the impact of region size on remembered set size, for regions ranging from 64 KB to 8 MB. The impact of region size is dramatic, with the footprint decreasing exponentially to insignificantly small numbers. The remset overhead is so high at small region sizes that the 64 KB region size failed to run to completion on pjbb2005.

5.4 Barrier Overheads

Figure 8 and Table 4 evaluate the overheads of four different barriers, compared to the SIM collector with no barriers. In each case, the only change is the introduction of the barrier — the collector performs identically in each case. Since the collector behavior is unchanged, and we are concerned only with the barrier overhead, we only present mutator overhead in these results. The graph shows mutator time normalized to the time of the base case with no barriers. The top graph uses a recent Skylake microarchitecture, while the bottom uses an older Sandy Bridge machine. We use two microarchitectures because prior work has demonstrated that barrier performance can be sensitive to microarchitecture [38]. The overheads in Figure 8 are slightly higher than those observed on the Skylake machine.

The SATB barrier results show that the conditional SATB barrier we propose has a measurable advantage compared to unconditional SATB barrier, which is used by OpenJDK's G1. The advantage is likely to be greater when the whole system is measured because the conditional barrier generates fewer SATB entries and thus less SATB tracing work.

Comparing the filtered and unfiltered remset barriers (7.7% v 10.7%), it is clear that filtering out intra-region pointers with the XOR is effective. We also evaluated the remset barrier overhead as a function of region size, from 64 KB to 8 MB. We saw no significant difference in terms of the barrier overhead (not shown).

The overhead results show that the cost of the combined barrier (12.4%) is higher than the sum of the parts (3.4% + 7.7%). We measured the L1 instruction cache miss rate of all of the measured barriers, normalized to the base case with no barrier. The results show a high instruction cache miss rate for both the unconditional SATB Barrier and the combined barrier, compared to other barriers, which is consistent with these barriers having significantly larger fast paths. This likely explains the discrepancy between the sum and the parts we see here.

	Percent Overhead / SIM (no barriers)			
	Skylake		**Sandy Bridge**	
	Time	L1I Miss	Time	L1I Miss
Conditional SATB	3.4	5.4	4.0	7.7
Unconditional SATB	5.5	13.9	6.9	20.9
Remset	7.7	9.6	9.3	12.4
Unfiltered Remset	10.7	4.5	12.2	12.3
Remset + Uncond. SATB	12.4	26.5	14.0	31.3

Table 4. Mutator overheads for barriers, in time and L1I misses, relative to SIM, the base case with no barriers. Geomean of 19 benchmarks. Results shown as a percentage.

6 Threats to Validity

The key contribution of our work is based on a methodology of deconstruction. This approach allows insight into the underlying algorithms, but comes with methodological risks and threats to validity of our findings.

First, our re-construction of G1 is built upon MMTk [2] and Jikes RVM [1], which provides us with a modular, performant GC framework ideal for such a study. Other GC frameworks are not readily amenable to such studies, as evidenced by the absence of prior studies of this kind. However, the major drawbacks of using Jikes RVM are that it only supports 32-bit machines with up to 2GB heap sizes, and only supports up to Java 1.6 (due to its use of GNU Classpath).

Second, no matter how careful, any reconstruction of a nearly-two-decade mature production system may not reflect every nuanced optimization both in the algorithm itself and in the runtime's adaptation to that algorithm (particularly the compiler).

We thus identify five major threats to the validity of findings reported here:

- The behavior of G1 on 64-bit machines and large heap sizes remains unexplored by our work. A 64-bit platform may impact tradeoffs in barriers and remembered set design due to lower pointer density (unless compressed pointers are used). Larger heap sizes would allow exploration of the scalability of the family of algorithms, particularly with respect to pause times.
- We only evaluate the remembered set footprint with a maximum region size of 1 MB with modest heap sizes (< 2 GB), so have not considered the case of very large heaps with large numbers of regions. Instead, we use small region sizes (64 KB to 512 KB) to increase the number of regions.
- We can only perform our measurements using relatively outdated benchmarks that compile on Java 1.6.
- Although our implementation is faithful to the original G1 algorithm and the key algorithmic features present in the implementation within OpenJDK today, further maturation of our implementation will inevitably affect performance.
- Our study does not attempt to compare among the various production collectors, such as G1, Shenandoah, C4 and ZGC. Each of those collectors is highly tuned, and is constantly refined. A study of the relative merits

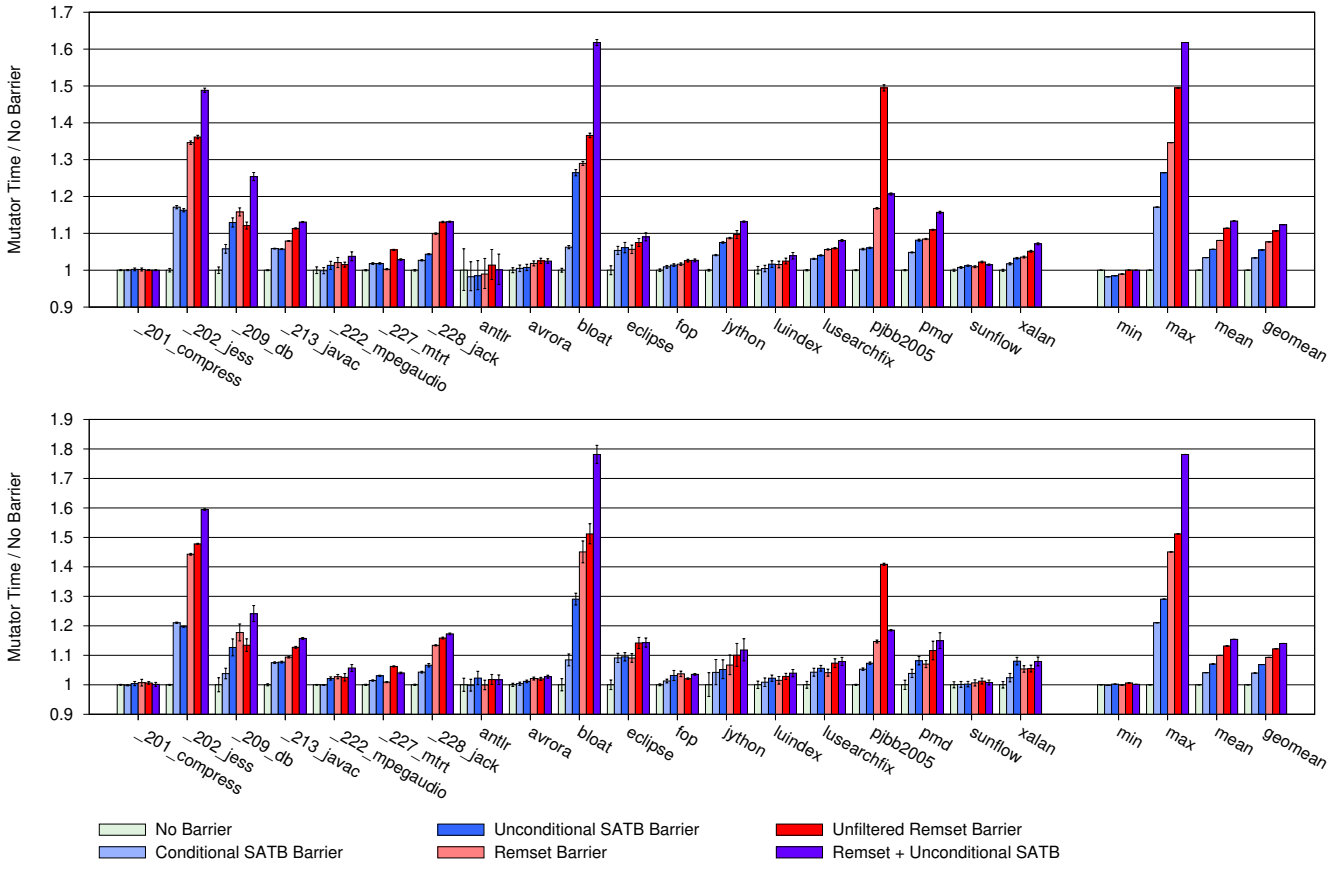

Figure 8. Barrier overheads for Skylake (top) and Sandy Bridge (bottom) microarchitectures.

these collector implementations is beyond the scope of this paper.

7 Conclusion

We explore the family of fixed-sized region-based concurrent evacuating garbage collectors by deconstructing the design and implementation of G1 and showing its relationship to other contemporary collectors. We synthesize a collector, CRE, reflecting an algorithmic foundation common to G1, Shenandoah and ZGC. We implement six variations of the most simple underlying region-based collector, SIM, offering insight into the various tradeoffs inherent to the design of G1. We identify the overheads associated with the collection strategy employed by G1, and measure the overheads of its write barriers and remembered sets.

We ground our evaluation with a study of an absolutely minimal fixed-size region-based evacuating collector, SIM, and identify its fundamental advantages and costs compared to the most simple of copying collectors, semi-space. We then build and analyze region-based collectors of incrementally greater sophistication, finishing with two standard variations on G1. We illustrate how each of these design steps contributes to the performance of G1 in terms of through put and pause time. Remembered sets are core to the design of G1, so we conduct a detailed analysis of the space overheads associated with remembered sets (they are very low), and how those overheads grow as region size shrinks. We also evaluate the overheads of each of the barriers used to support G1, revealing how key aspects of the algorithm impact mutator performance.

Our hope is that these insights will inform those using such collectors and those designing the next generation of algorithms.

8 Acknowledgements

We thank Dengyu Gu, Kathryn S. McKinley, Aleksey Shipilev, Kunshan Wang, and our anonymous reviewers for their generous feedback.

References

[1] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.

[2] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. 2004. Oil and water? High performance garbage collection in Java with MMTk.

In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 137–146.

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190. https://doi.org/10.1145/1167515.1167488

[4] Stephen M. Blackburn, Martin Hirzel, Robin Garner, and Darko Stefanović. [n. d.]. pjbb2005: The pseudoJBB benchmark. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

[5] Stephen M. Blackburn and Antony L. Hosking. 2004. Barriers: friend or foe?. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24 - 25, 2004*. ACM, 143–151. https://doi.org/10.1145/1029873.1029891

[6] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 22–32.

[7] Rodney A Brooks. 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 256–262.

[8] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. 13, 11 (Nov. 1970), 677–678.

[9] Iris Clark, Roman Kennke, and Aleksey Shipilev. 2015. OpenJDKWiki: Shenandoah GC. https://wiki.openjdk.java.net/display/Shenandoah#Main-PerformanceAnalysis Accessed February 2020.

[10] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

[11] W. D. Clinger and L. T. Hansen. 1997. Generational Garbage Collection and the Radioactive Decay Model. Las Vegas, NV, 73–85.

[12] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

[13] Robert T. Dimpsey, Rajiv Arora, and Kean Kuiper. 2000. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal* 39, 1 (2000), 151–174. https://doi.org/10.1147/sj.391.0151

[14] Robert R Fenichel and Jerome C Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612.

[15] Christine H Flood, Roman Kennke, Andrew 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*. ACM, 13.

[16] Richard L. Hudson and J. Eliot B. Moss. 1992. Incremental Collection of Mature Objects. In *Proceedings of the International Workshop on Memory Management (Lecture Notes on Computer Science)*. Springer-Verlag, 388–403.

[17] Roman Kennke. 2019. Shenandoah GC in JDK 13, Part 1: Load reference barriers. https://developers.redhat.com/blog/2019/06/27/shenandoah-gc-in-jdk-13-part-1-load-reference-barriers/

[18] Roman Kennke. 2019. Shenandoah GC in JDK 13, Part 2: Eliminating the forward pointer word. https://developers.redhat.com/blog/2019/06/28/shenandoah-gc-in-jdk-13-part-2-eliminating-the-forward-pointer-word/

[19] Roman Kennke. 2019. Shenandoah GC in JDK 13, Part 3: Architectures and operating systems. https://developers.redhat.com/blog/2019/07/01/shenandoah-gc-in-jdk-13-part-3-architectures-and-operating-systems/

[20] Roman Kennke. 2020. Shenandoah 2.0. In *FOSDEM'20, Brussels, February 1–2, 2020*. https://fosdem.org/2020/schedule/event/shenandoah/

[21] Bernard Lang and Francis Dupont. 1987. Incremental incrementally compacting garbage collection. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques, 1987, St. Paul, Minnesota, USA, June 24 - 26, 1987*, Richard L. Wexelblat (Ed.). ACM, 253–263. https://doi.org/10.1145/29650.29677

[22] Per Liden and Stefan Karlsson. 2018. ZGC: A Scalable Low-Latency Garbage Collector. http://openjdk.java.net/jeps/333

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

[24] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

[25] Khanh Nguyen, Lu Fang, Guoqing Xu, and Brian Demsky. 2015. Speculative region-based memory management for big data systems. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. ACM, 27–32.

[26] OpenJDK. 2018 (accessed Feb 15, 2020). heapRegion.cpp. https://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/hotspot/share/gc/g1/heapRegion.cpp#l63

[27] OpenJDK. 2018 (accessed March 4, 2019). g1BarrierSetC1.cpp. https://hg.openjdk.java.net/jdk/jdk11/file/tip/src/hotspot/share/gc/g1/c1/g1BarrierSetC1.cpp#l149

[28] OpenJDK. 2018 (accessed March 4, 2019). g1BarrierSetC2.cpp. https://hg.openjdk.java.net/jdk/jdk11/file/tip/src/hotspot/share/gc/g1/c2/g1BarrierSetC2.cpp#l452

[29] OpenJDK. 2018 (accessed March 4, 2019). g1BarrierSet.cpp. https://hg.openjdk.java.net/jdk/jdk11/file/tip/src/hotspot/share/gc/g1/g1BarrierSet.cpp#l99

[30] OpenJDK. 2018 (accessed March 9, 2019). g1BarrierSet.inline.hpp. https://hg.openjdk.java.net/jdk/jdk11/file/tip/src/hotspot/share/gc/g1/g1BarrierSet.inline.hpp#l36

[31] OpenJDK. 2018 (accessed Nov 27, 2019). g1Policy.cpp. https://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/hotspot/share/gc/g1/g1Policy.cpp#l213

[32] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. 2013. Taking off the gloves with reference counting Immix. *ACM SIGPLAN Notices* 48, 10 (2013), 93–110.

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

[34] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-Based Garbage Collection. Denver, CO.

[35] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices* 46, 11 (2011), 79–88.

[36] 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.

[37] P. R. Wilson and T. G. Moher. 1989. A "Card-marking" Scheme for Controlling Intergenerational References in Generation-based Garbage Collection on Stock Hardware. 87–92. https://doi.org/10.1145/66068.66077

[38] Xi Yang, Stephen M Blackburn, Daniel Frampton, and Antony L Hosking. 2012. Barriers reconsidered, friendlier still!. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[39] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *Proceedings of the 11th International Symposium on Memory Management, ISMM 2012, Beijng, China, June 15 - 16, 2012*. ACM.

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