Bounded-Latency Regional Garbage Collection

Felix S. Klock II
Adobe Systems Incorporated
pnkfelix@ccs.neu.edu

William D. Clinger
Northeastern University
will@ccs.neu.edu

Abstract

Regional garbage collection is scalable, with theoretical worst-case bounds for gc latency, MMU, and throughput that are independent of mutator behavior and the volume of reachable storage. Regional collection improves upon the worst-case pause times and MMU seen in most other general-purpose collectors, including garbage-first and concurrent mark/sweep collectors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Design, Performance

Keywords scalable, real-time, regional garbage collection

1. Introduction

Some applications cannot tolerate long interruptions caused by garbage collection. The frequency and/or duration of gc-related interruptions can be reduced by using generational, parallel, concurrent, incremental, real-time, or reference-counting collectors, but those technologies are generally unable to guarantee a provable hard bound on the duration of gc-related interruptions in a general-purpose collector without making some assumptions about mutator behavior and/or the volume of reachable storage.

Figure 1 lists the longest gc pause for four unusually gc-intensive benchmarks that offer a fair comparison between Java and Scheme. The garbage-first and concurrent mark/sweep collectors of Oracle's OpenJDK Server VM are state-of-the-art incremental collectors that were designed to reduce gc pauses, but both of those collectors resort to full collections when necessary to get them out of trouble. That's why their pause times are worse than the Server VM's default collector on about half of the benchmarks shown in Figure 1.

Most of the so-called real-time garbage collectors must be tuned to the behavior of some specific embedded system, so they cannot provide hard real-time guarantees independent of application and problem size [18]. Reference counting does not collect cyclic garbage, so general-purpose implementations that use reference counting do not collect cyclic garbage unless they incorporate an auxiliary phase or garbage collector, and the pause times for that auxiliary method are unlikely to be bounded [27].

In this paper, we report on the performance of a general-purpose garbage collector that has a provable fixed bound on the duration of gc-related pauses, independent of application and problem size [17, 28]. This *regional* collector is now available in Larceny v0.98b1.[1]

As seen in Figure 1, Larceny's regional collector delivers bounded gc latency even for near-worst-case synthetic benchmarks that force most other incremental collectors to perform full collections. On three of the four benchmarks shown in that figure, Larceny with its regional collector consumes less cpu time than Java with its garbage-first or concurrent mark/sweep collectors.

As with other incremental collectors, Larceny's regional collector sacrifices some throughput. Across a standard set of 68 Scheme benchmarks, Larceny v0.98b1 runs about 12% slower overall (geometric mean) with its regional collector than with its default generational collector. Even so, the regional collector usually delivers better throughput than Larceny's stop&copy collector, whose overall throughput is about 23% slower than the default collector's. Section 8 reports these results in greater detail.

2. Contributions

Although this paper is based on Felix Klock's doctoral dissertation, which was completed in January of 2011 [28], we report on an implementation of the regional collector that was released nine months later and performs better than Klock's prototype.

Our previous paper, published in 2009 at the *Workshop on Scheme and Functional Programming*, defined our notion of scalability and sketched a proof of our main theorem [17]. To support that proof, we also gave a high-level description of regional garbage collection.

In this paper, we describe the algorithm and our implementation of it in more detail. We summarize its throughput for a standard set of Scheme benchmarks and also present detailed timings, pause times, and graphs of the minimum mutator utilization for several near-worst-case benchmarks that we have rewritten to provide fair comparisions with Java. These benchmarks allow us to compare our regional collector with Oracle's Garbage-First (G1) collector, which was not available at the time of our previous paper.

We also discuss the recent C4 collector, and have expanded our discussion of other related work.

2.1 Cautionary note concerning "region"

Our use of the term "region" comes from the original paper on generational garbage collection [31].

Regional garbage collection is not related to region-based memory management [22, 39, 40]. Tofte-style region-based memory management is a static strategy, and cannot guarantee the dynamic worst-case bounds on space that are stated in Theorem 1 below.

3. Scalability

Regional garbage collection is a scalable variant of generational garbage collection [17, 28]. Theorem 1 below, whose proof was the

[1] `http://www.larcenists.org/`

DLS'11, October 24, 2011, Portland, Oregon, USA.

system	gc technology	longest gc pause (seconds) 5gcbenchJ:24	permJ	queueJ	pueueJ
Scheme (Larceny v0.98b1)	regional	.12	.13	.09	.21
Scheme (Larceny v0.98b1)	generational	3.13	3.23	4.28	4.45
Scheme (Larceny v0.98b1)	stop&copy	2.94	3.44	4.62	4.74
Java (OpenJDK 19.0-b09)	default (same as parallel?)	3.02	3.07	3.00	3.31
Java (OpenJDK 19.0-b09)	parallel gc	2.78	2.93	3.24	3.32
Java (OpenJDK 19.0-b09)	garbage-first (G1)	2.13	4.68	4.29	5.84
Java (OpenJDK 19.0-b09)	concurrent mark/sweep	15.45	.50	.45	5.94

Figure 1. Longest gc pause time (in seconds) observed for four near-worst-case benchmarks. If the size of heap memory is reduced by 20%, the concurrent mark/sweep collector's longest pause time increases to 10 seconds or more on all four benchmarks. See Section 8.

main result of our previous paper, guarantees non-trivial theoretical worst-case bounds for space, collection pauses, and minimum mutator utilization (explained in Section 4.1). The space is $O(n)$, where n is the volume of reachable storage, and the duration of collection pauses is bounded by a fixed constant that is independent of the application and the volume of reachable storage.

Most garbage collectors cannot guarantee those scalability properties. Conventional generational collectors do not have those properties because they must occasionally perform a full collection that takes time proportional to live storage, and mutator operations are normally excluded during that collection.

Our proof of Theorem 1 was the first to guarantee simultaneous, non-trivial, and mutator-independent bounds for space, collection latency, and minimum mutator utilization [17, 28]. It might be possible to prove similar theorems for a few (though not all!) of the so-called real-time collectors, but those collectors sacrifice some throughput even for programs that allocate little storage and generate little or no garbage. To understand why, note first that any scalable general-purpose collector must be able to move objects; otherwise there would be no theoretical worst-case bound on fragmentation, hence no theoretical worst-case bound on the ratio of heap space to reachable storage. Real-time collectors that move objects generally use a read barrier, which may be implemented in hardware using memory protection or in software by an extra level of indirection through a handle or forwarding pointer. Read barriers degrade amortized performance on all programs [18].

Real-time collectors that attempt to move objects without using a read barrier generally use mutual exclusion to prevent the mutator from running while the collector is moving objects, and attempt to limit the work performed by the collector while the mutator is stopped. That approach works for some programs, but runs into a problem with *popular objects*, defined as objects that are referenced by many other objects. When a popular object is moved, and no read barrier is used, all references to the popular object must be updated to point to the new location of the object. Since the only *a priori* bound on the number of references to the popular object is the size of the heap, the theoretical collection latency is unbounded and the collector is not scalable.

Our regional collector never moves popular objects. In fact, the regional collector never even tries to collect the small regions that contain popular objects. That works because the regions that contain popular objects can never account for more than a bounded fraction of the heap. That was the key insight and chief novelty of regional garbage collection and of our proof of Theorem 1.

In this paper, we build upon the theoretical focus of our previous paper by answering some pragmatic questions: Are the theoretical worst-case bounds good enough to offer practical benefit? How well does our regional collector perform on near-worst-case benchmarks when compared to other incremental collectors? How well does the regional collector perform on typical programs?

4. Scalable collection: space, time, MMU

The overheads of garbage collection involve both space and time.

The space overhead includes *floating garbage*, which is defined as objects on the application's heap that are unreachable but have not yet been reclaimed by the collector. For example, a generational collector's remembered set may contain references to objects that are otherwise unreachable, causing those objects (and all objects reachable from them!) to be retained across many partial collections. A collector is *scalable* with respect to space if and only if its worst-case memory overhead, including floating garbage, is $O(P)$ where P is the peak reachable heap storage.

The time overhead degrades both *throughput* and *responsiveness*. Degraded throughput increases an application's total running time. Degraded responsiveness may take the form of long pauses while the garbage collector runs and the mutator doesn't, or may take the form of intervals in which the mutator's share of machine resources is so puny as to annoy users or fail to satisfy real-time guarantees.

4.1 Evaluating responsiveness: pauses and MMU

One measure of responsiveness is the maximum pause time observed when an application is run. Although maximum pause time is an intuitive and important metric, it can be misleading: To users, a densely packed series of short pauses with hardly any mutator activity between two adjacent pauses may be indistinguishable from a single long pause.

In the garbage collection community, the application program excluding the garbage collector is regarded as a mutator process that runs in coroutining fashion with the garbage collection process. For any fixed interval of time, the *mutator utilization* during that interval is defined as the percentage of that interval in which the mutator has control. For any fixed time resolution, the *minimum mutator utilization* (MMU) is defined as the smallest mutator utilization over a set of time intervals whose length is equal to the resolution.

For any given execution of a program, the MMU so defined will be a function that maps positive time resolutions to percentages. One might think that MMU would be a monotonically increasing function of the length of the resolution interval, but that is not so: Consider a program in which the mutator runs for exactly 1 second, the garbage collector for exactly 1 second, and so on. At a resolution of 2 seconds, the MMU is 50%, which is also the average mutator utilization. At a resolution of 3 seconds, however, the MMU is only 33%: There are 3-second intervals in which the garbage collector runs twice, for a total of 2 seconds. In general, however, the MMU does tend to increase as the interval increases, approaching (but not reaching) the average mutator utilization.

In this work we distinguish between two kinds of MMU. *Observed MMU* is measured empirically during the executions of some specific set of benchmarks, as in Figures 2 and 3. *Theoretical worst-case MMU* is the infimum (greatest lower bound) of ob-

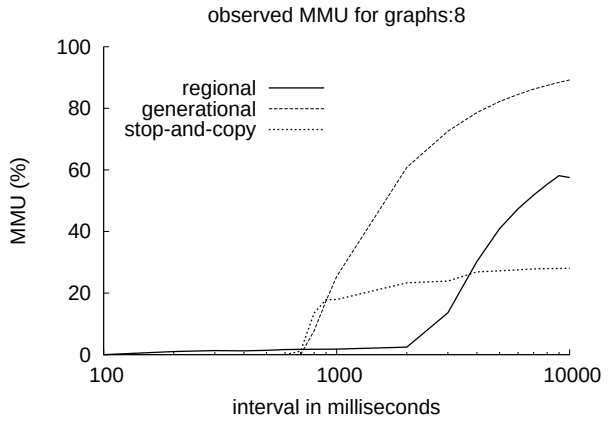

Figure 2. Observed minimum memory utilization (MMU) for an allocation-intensive benchmark of moderate difficulty.

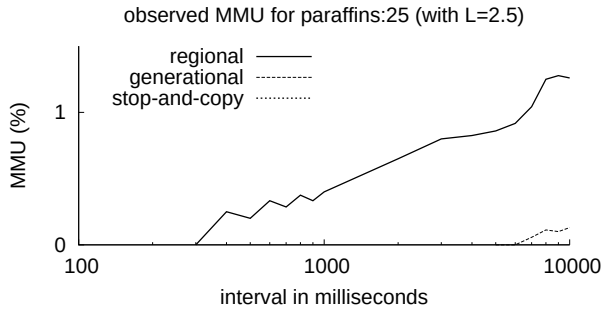

Figure 3. Observed MMU for an extremely gc-intensive benchmark. Larceny's stop&copy collector delivers essentially zero MMU at all intervals through 10 seconds, and Larceny's generational collector does only slightly better.

served MMUs over all possible executions of all possible benchmarks; it cannot be derived via measurement.

A collector is scalable with respect to responsiveness if there exists a resolution at which the theoretical worst-case MMU is non-zero, independent of the amount of live heap storage and mutator behavior.

5. Regional collectors are scalable

We say that a garbage collector is scalable if and only if it guarantees non-trivial theoretical worst-case bounds for space, time, collection pauses, and minimum mutator utilization that are independent of application behavior and problem size.

The following theorem states that the regional collector is scalable. (The non-trivial lower bound for MMU implies a non-trivial lower bound for throughput.)

Theorem 1. *There exist positive constants c_0, c_1, c_2, and c_3 such that, for every mutator, no matter what the mutator does:*

1. *GC pauses are independent of heap size: c_0 is larger than the worst-case time between mutator actions.*
2. *Minimum mutator utilization is bounded below by constants that are independent of heap size: within every interval of time longer than $3c_0$, the MMU is greater than c_1.*
3. *Memory usage is $O(P)$, where P is the peak volume of reachable objects: the total memory used by the mutator and collector is less than $c_2P + c_3$.*

Proof. See our previous paper [17] and the more complete proof in Klock's doctoral dissertation [28]. □

Note that the constants c_0, c_1, c_2, and c_3 are completely independent of the mutator. Hence

Corollary 2. *If a mutator operation executes in $O(1)$ mutator time, then the operation will execute in $O(1)$ elapsed time so long as the worst-case space stated by Theorem 1 is available.*

Prior to our proof of Theorem 1, no general-purpose garbage collector had ever been proved to have all of those non-trivial and mutator-independent bounds on worst-case performance.

Theorem 1 asserts the existence of a set of fixed bounds that apply to all mutators and to all volumes of reachable storage. Its $\exists\forall$ quantifier structure is stronger than the $\forall\exists$ structure of theorems that assert the existence of mutator-dependent bounds: Scalability is stronger than mutator-dependent bounds in exactly the same way that uniform continuity is stronger than pointwise continuity.

With most general-purpose collectors, simple user operations such as clicking in a scroll bar might allocate just enough storage to trigger a full collection, which takes time proportional to reachable storage. Since there is no *a priori* bound on reachable storage, there is no *a priori* bound on the time required to scroll a page; clicking in the scroll bar may give users the impression that the program has locked up.

Corollary 2 and Figure 3 say that can't happen with a regional collector. The program may run very slowly, but it will never give the impression of locking up completely.

6. Regional collection: how it works

The regional collector resembles a stop-the-world generational collector. In place of generations that segregate objects by age, the regional collector maintains a set of relatively small regions, all of the same size. There is no strict correlation between an object's region and the object's age. Only one region is collected at a time. (In most generational collectors, collecting a generation implies the simultaneous collection of all younger generations.)

The regional collector maintains a remembered set, a collection of summary sets, and a snapshot structure. The remembered set keeps track of region-crossing references, summary sets list all elements of the remembered set that will be relevant to upcoming collections, and the snapshot structure accumulates reachability information that will be used to eliminate unreachable references from the remembered set and summary sets.

6.1 Processes

The regional collector adds three distinct computational processes to those of the mutator:

- A snapshot-at-the-beginning marking process marks every object that was reachable when the snapshot was initiated.
- A summarization process computes summary sets from the remembered set.
- A collection process uses Cheney's algorithm [13] to copy a region's reachable storage into some other region(s).

The marking and summarization processes run concurrently or interleaved with the mutator processes. When the collection process is executing, however, all other processes are suspended.

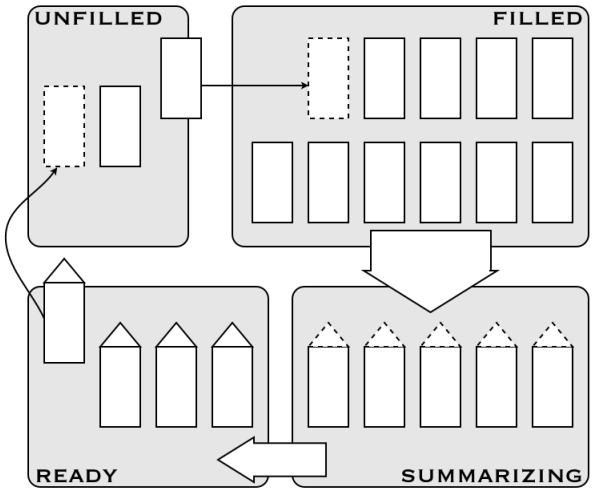

Figure 4. Regions cycle through a four-stage circular pipeline.

The collection process moves objects (to eliminate fragmentation) and updates pointers from outside the collected region to point to the newly relocated objects. It also reclaims unreachable storage.

Because the collection process moves objects, all other processes must be suspended during a collection. Those processes are free to proceed only at the end of collection, after the collection process has updated all pointers to point to the new locations of objects.

The collection process is the *only* process that moves objects. Because the mutator, summarization, and marking processes do not move objects, they cannot interfere with other processes' views of the heap, even if they execute concurrently.

When a region is collected, its summary set contains all of the locations within uncollected regions that hold pointers into the collected region. (See Figure 5 and Section 6.8.) Those locations must be updated after the collection so they will point to the surviving objects' new addresses. The summary sets for uncollected regions must be updated as well, because any locations they may have contained that lie within the collected region now correspond to new locations within relocated objects. Similarly, the marking process's mark stack and all of the mutator's call stacks must be updated. The mark stack is discussed in Section 6.6. Mutator stacks are discussed in Section 6.15.

6.2 Classification of regions

Figure 4 shows the regions' normal pipeline in which an empty or partially empty region is used as part of the to-space for a Cheney collection, becomes full, is eventually selected for summarization, becomes ready for collection after its summary set has been computed, and becomes empty once again after its reachable objects have been copied into one or more unfilled regions.

The small solid rectanges represent regions. The thin arrows represent an individual region's transition from one state to another. The thin curved arrow shows a ready region being emptied by a collection, after which the now-empty region is reclassified as unfilled. The thin straight arrow shows a region being reclassified after it has been filled by objects evacuated from a collected region.

The triangular hats on some regions represent summary sets. A summary set that is still under construction has a dashed outline. The broad arrows represent transitions of multiple regions at the beginning or end of a summarization cycle. At the beginning of a summarization cycle, some of the filled regions (usually the ones that were filled least recently) become candidates for summarization. Their summary sets are computed by scanning the remembered set for the entire heap, which takes a while. When their summary sets are complete, those regions become ready for collection.

While the summary sets are being computed, ready regions are collected one at a time, at a rate determined by mutator activity. The summarization process must complete its computation of the summary sets before or soon after the last ready region has been collected. Sections 6.12 and 6.13 discuss the scheduling constraints in more detail.

6.3 Remembered set

We bound the pause time by collecting one region independently of all others. To enable this, the mutator and collector collaboratively maintain a *remembered set* that contains every location (or object) that points from one region to a different region. (Standard generational collectors maintain a similar invariant, but their remembered sets don't have to keep track of locations that point from younger to older objects.)

The mutator can create region-crossing pointers by allocation (as when an argument to `cons` resides within a different region from the resulting pair) or by assignment. The collector can create region-crossing pointers by moving an object from one region to another.

The remembered set may suffer from two distinct kinds of imprecision:

- The remembered set may contain entries for locations or objects that are no longer reachable by the mutator.
- The remembered set may contain entries for locations or objects that are still reachable, but no longer contain a pointer that points from one region to a different region.

The remembered set shown in Figure 5 is precise.

6.4 Remembered set representation

The regional collector represents its remembered set using a card table, hash table, or some other data structure that records at most one entry for each location in the heap.

The size of the remembered set's representation is therefore bounded by the size of the heap, even though the remembered set is imprecise. Without that bound, the collector would not be scalable.

6.5 Write barrier

Assignments and other stores into pointer fields of objects must go through a *write barrier* that updates the remembered set to account for the assignment.

To support snapshot-at-the-beginning marking, the regional collector uses a Yuasa-style write barrier [42] that logs three things: (1) the location on the left hand side of the assignment, (2) its previous contents, and (3) its new contents. Because the write barrier must log the previous contents, our regional collector cannot use a write barrier that writes directly to a card table (even if the remembered set happens to be represented by a card table). In our benchmark results, the cost of that write barrier is charged to the mutator, but it is a hidden cost of regional collection.

6.6 Snapshots

To keep the remembered set's imprecision within fixed bounds, a periodic snapshot-at-the-beginning marking process incrementally constructs a snapshot of the heap at a particular point in time.

Incremental snapshot construction is a standard technique [42]. It classifies every object as (1) unreachable at the time of the snapshot, (2) reachable at the time of the snapshot, or (3) allocated after the time of the snapshot. Objects in category (3) are considered reachable, but their fields need not be traced by the marker because those fields cannot affect reachability as of the time of the snapshot.

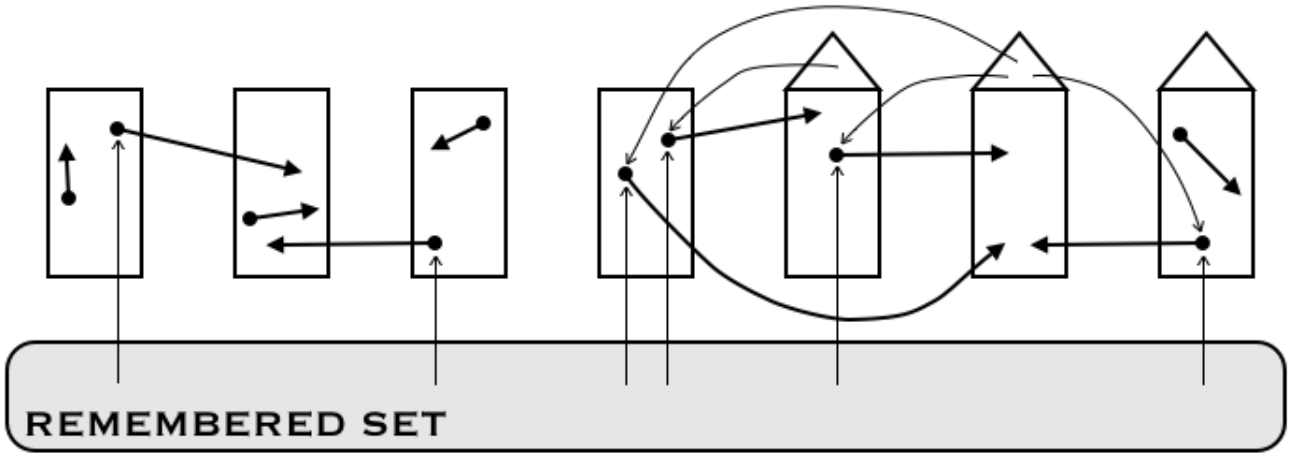

Figure 5. Relationship between the points-out-of remembered set and the points-into summary sets.

When the marking process completes, it removes unreachable locations from the remembered set. This reduces floating garbage. In particular, it ensures that the nodes of any cyclic structures that become unreachable will be removed from the remembered set.

While the marking process is constructing the snapshot, it maintains a *mark stack* of objects that have been marked but have not yet been traced.

6.7 Mark stack and snapshot

The marking process marks every location it pushes, and never pushes a marked location, so no location appears more than once in the mark stack, and the size of the mark stack is bounded by the size of the heap.

The number of pointers in the mark stack that need to be updated after a collection is bounded by the size of the collected region. To update those pointers in time proportional to region size, the mark stack is divided into per-region substacks, those substacks are threaded through the single mark stack, and the collector updates *only* the substack that's relevant to the collected region.

Two more subtle issues arise with the mark stack and snapshot:

1. Collections must not change the state of objects within the snapshot. In particular, unmarked objects must remain unmarked.
2. Even if an object on the mark stack has become globally unreachable (so one might think the collector could reclaim its storage), that object may hold the only reference the marking process will encounter to a different object that remains globally reachable via some path that was introduced by assignments performed after the snapshot was initiated. (For an example, see Section 4.6.1 of Klock's dissertation [28].)

To resolve that second issue, the regional collector uses the mark substack for the collected region as an additional source of roots for that collection. The collector does *not* use the entire mark stack or snapshot as roots; if it did, the collector would not be scalable, and garbage that was live when the snapshot was initiated would not be collected.

6.8 Summary sets

A typical generational collector will scan most (or all) of its remembered set when collecting one of its younger generations. In a worst case, the size of that remembered set can be proportional to the size of the heap, so that approach is not scalable.

To collect a region independently of other regions, the collector must know all locations in uncollected regions that may hold pointers into the collected region. This set of locations is the *summary set* for the collected region.

Figure 5 shows a heap with 7 regions, an object graph with 6 region-crossing pointers, and 3 summary sets (the triangular hats).

6.9 Popular regions

In the worst case, a region's summary set could consist of all locations outside the region. If that region were collected by moving its reachable objects, then it would take too long to update all pointers to those objects, and the collector would not be scalable.

To solve that problem, the regional collector defines a *popular* region as a region whose summary set is larger than S times the region size R, and never collects a popular region.[2]

It is impossible for all regions to be more popular than average. That mathematical observation generalizes to the following lemma.

Lemma 3. *If $S > 1$, then the fraction of regions that are popular is no greater than $1/S$.*

Example: If $S = 8$, then at most 12.5% of the regions are popular, and not collecting those popular regions adds at most 12.5% to the size of the heap.

That is the key insight that allows regional collection to be scalable. To achieve scalability, it is also necessary to keep the imprecision of the remembered set within fixed bounds (which is accomplished by the marking process) and to abandon computation of any summary sets whose size exceeds the threshold set by S.

Popularity is not permanent. Regions that go uncollected because of their current popularity will still be considered for collection in some later full cycle. Regions that lose their popular status will eventually be collected.

6.10 Amortized summary set construction and wave-off

Recall that the portion of the remembered set associated with a region consists of all locations within that region that may point outside the region. (In other words, we use a *points-out-of* remembered

[2] Oracle's garbage-first collector doesn't collect its popular regions either, but it does move popular objects to a dedicated space [20]. In the worst case, updating all pointers to a popular object during that migration can take time proportional to the heap size, so the garbage-first collector is not scalable. Our regional collector does not move popular objects at all.

set. An imprecise *points-into* remembered set, such as the one used in Oracle's garbage-first collector, would not be scalable because its worst-case size is quadratic in the heap size.)

To compute a region's summary set, the summarization process must scan the points-out-of remembered set for the entire heap, which takes time proportional to the size of the heap. Just-in-time computation of individual summary sets would not be scalable.

To keep both time and space under control, the summarization process

- amortizes the cost in time by attempting to compute summary sets for a fixed fraction $1/F_1$ of the heap's regions during a full pass over the remembered set, but
- abandons the computation of any summary set whose size exceeds the fixed wave-off threshold S.

If the number of usable summary sets computed by a summarization cycle exceeds a fixed fraction $1/(F_1F_2)$ of the number of regions in the heap, then the summarization process can be suspended until the number of ready regions drops below that fraction.

6.11 Parameters

The key parameters (and their values in Larceny v0.98b1):

- R (5 megabytes) is the fixed region size.
- S (8) is the wave-off threshold.
- $1/F_1$ ($1/2$) is the fraction of the heap for which the summarization process attempts to build summary sets during each summarization cycle.
- $1/F_2$ ($1/2$) is the fraction of those attempts that must succeed.

The values of these parameters are hard-wired into the regional collector. If they were allowed to vary depending on the mutator or heap size, then the collector would not be scalable. They are parameters only in the sense that several different sets of parameter values are known to yield a scalable collector [28].

6.12 Proof-driven scheduling

If the mutator weren't executing concurrently (or interleaved) with the summarization pass, then the number of abandoned summary sets would be limited by Lemma 3, and the fraction $1/(F_1F_2)$ would be guaranteed by choosing the fixed parameter F_2 so that $1/(F_1F_2) < 1/F_1 - 1/S$. Unfortunately, mutators can perform allocations and assignments that increase or reduce the number of pointers that point into a region being summarized, making previously unpopular regions popular or previously popular regions unpopular. To achieve scalability, the regional collector must therefore arrange for the summarization process to finish before the mutator can perform enough allocation and/or assignments to prevent summary sets from being computed for $1/(F_1F_2)$ of the regions.

After the summary sets have been computed, they must be kept up to date by scanning newly allocated objects and by processing assignments logged by the write barrier.[3] If an updated summary set were to grow too large, then it would take too long to update all of its locations when the associated region is collected, and the collector would not be scalable. The regional collector must therefore discard any summary set whose size grows to exceed a fixed threshold $S' \geq S$. To retain scalability, the regional collector must consume summary sets (by collecting regions) fast enough to prevent too many summary sets from being discarded.

[3] Keeping the summary sets up to date with only $O(1)$ amortized time per allocation/assignment requires sophisticated data structures combined with fixed bounds on the size of a summary set and on the number of updates that are necessary. For details, see Klock's dissertation [28].

Scalability therefore requires careful scheduling. The regional collector must compute summary sets and collect regions at a sufficiently high rate, but must not compute summary sets and collect regions so rapidly that the mutator's share of the machine drops below the minimum mutator utilization guaranteed by Theorem 1. The regional collector's scheduling of summarization and collection is constrained by the same lemmas that were used to prove its scalability:

Lemma 4. *If N is the total heap size and R the size of each region (so N/R is the number of regions), and a summarization cycle finishes before the mutator's activity during the cycle can exceed cN, where c satisfies*

$$0 < c \leq \frac{F_2 - 1}{F_1 F_2} S - \frac{S}{N/R} - 1$$

then the summarization cycle will compute usable summary sets for at least

$$\frac{1}{F_1 F_2} \frac{N}{R}$$

regions.

Lemma 5. *Let D be the total size of the previously constructed summary sets at the start of a new summarization cycle. If the summarization cycle finishes before the mutator's activity during the cycle can exceed cN, then the number of summary sets that remain undiscarded throughout the entire cycle (because their size never exceeds $S'R$) is at least*

$$\left(\frac{1}{F_1 F_2} - \frac{D}{NS'} - \frac{c}{S'} \right) \frac{N}{R}$$

Both lemmas above are proved in Chapter 5 of Klock's dissertation [28].

These lemmas establish fixed upper bounds on mutator activity that must be enforced by scheduling. To achieve scalability, the scheduler must also enforce a fixed lower bound on mutator activity. The proof of Theorem 1 establishes that these (and other) fixed upper and lower bounds can be satisfied simultaneously by careful scheduling of mutator, marking, summarization, and collection processes.

6.13 Mutator activity

Allocation and assignment are the only operations that can change the object graph, so mutator activity is reckoned as the sum of words allocated and word-sized assignments performed.

During each *full cycle*, the regional collector collects every region that existed at the beginning of the cycle unless the region is empty or its summary set would be too large.

The collector's overhead during each full cycle is $\Theta(P)$,[4] where P is the peak live storage. Furthermore the collector's overhead can be spread fairly evenly over the collections that are performed during that full cycle.

To achieve scalability, the regional collector must arrange for the mutator activity within each full cycle to be $\Theta(P)$ (hence proportional to the collector's overhead) and for that mutator activity to be spread fairly evenly over the collections that are performed during that full cycle.

At the beginning of each full cycle, the regional collector calculates the mutator activity to be performed within that full cycle, and uses that calculated value to schedule its own activities as well as those of the mutator throughout that full cycle. That calculation involves an inverse load factor L, which expresses the desired ratio

[4] $\Theta(P)$ means the overhead is proportional to P to within fixed lower and upper bounds for the constant of proportionality. To achieve scalability, those fixed bounds must be independent of the mutator.

between heap size and peak reachable storage. For $1 < L \leq 3$ and collector parameters $S = 8$, $F_1 = 2$, and $F_2 = 2$, the mutator activity per full cycle is $(L-1)P$, where the peak reachable storage P is estimated by taking the maximum over all completed marking cycles. (By bounding the mutator's activity since completion of the previous marking cycle, the collector ensures that the current amount of live storage cannot be greater than a fixed constant times that estimate [17, 28].)

Because the mutator activity per full cycle is $\Theta(P)$, and the collector's overhead per full cycle is also $\Theta(P)$, scalability is achieved by any reasonably even distribution (to within fixed bounds) of both mutator activity and collector overhead amongst the region-sized collections that will be performed within the full cycle.

6.14 Nursery

Like most generational collectors, the regional collector allocates all objects within a relatively small *nursery*. Small nurseries have little impact on worst-case performance, and the scalability asserted by Theorem 1 can be achieved without using a nursery. For most programs, however, the nursery greatly improves the observed MMU and overall efficiency of the regional collector.

6.15 Mutator stacks

The regional collector assumes mutator stacks are constructed from heap-allocated objects of bounded size, as though all stack frames were allocated on the heap [1]. Although mixed stack/heap, incremental stack/heap, Hieb-Dybvig-Bruggeman, and Cheney-on-the-MTA strategies are often used [16, 25], their bounded stack caches can be regarded as special parts of the nursery. That allows a regional collector to deal with them as though the mutator uses a pure heap strategy.

6.16 Fragmentation

As justified in Section 10, the regional collector assumes objects are limited to some size $m < R$. The Cheney algorithm ensures that worst-case fragmentation in collected regions is less than m/R.

7. Larceny's regional collector

We have implemented a regional collector for Larceny, an implementation of Scheme [23].[5] For Larceny v0.98b1, the regional collector's nursery size is 1 MB, and its other parameter values are stated in Section 6.11.

Larceny's regional collector does not exploit the potential concurrency of our design: The marking and summarization processes *could* run concurrently with the mutator but do not.

The marking and summarization processes *are* incremental. Incremental marking is performed before most minor collections. The summarization process scans the outgoing pointers for only one or two regions at a time, and those scans are scheduled at the fine grain of Larceny's software timer interrupts.

8. Performance

For this paper, we compared Larceny's regional collector to two of Larceny's other garbage collectors on a standard set of Scheme benchmarks. We also compared Larceny's collectors to those of Oracle's OpenJDK Server VM (build 19.0-b09), in 32-bit mode, on four near-worst-case synthetic benchmarks that were designed to provide a fair comparison between Scheme and Java. Larceny's collectors were run with an inverse load factor of $L = 2.5$, and the JVM's collectors were run with heap sizes that approximated the peak memory used by Larceny. We ran these benchmarks on an otherwise unloaded Intel Core 2 Duo (with two processor cores) running at 3 GHz, with 3.8 gigabytes of RAM, under Linux Ubuntu 9.10.

The numbers reported here are for a single run. For the regional collector, that run was selected by taking the run whose maximum pause time was the median of five consecutive runs. For those five runs, the pause times ranged from 83 to 129 ms (`gcbenchJ`), 114 to 192 ms (`permJ`), 81 to 114 ms (`queueJ`), and 211 to 214 ms (`pueueJ`). With all of the collectors, overall timings and memory usage were consistent across multiple runs to within a few per cent. Pause times and MMU varied quite a bit more, especially with collectors that performed some but not many full collections. With Java's concurrent mark/sweep collector, full collections can be induced by reducing the heap size, resulting in at least one pause of 10 to 32 seconds on every benchmark. For all other collectors, the pause times and MMU for the runs reported here were typical and close to the median and average.

8.1 Representative benchmarks

To estimate performance on typical Scheme programs, we benchmarked three of Larceny's garbage collectors on the 68 R6RS benchmarks that are distributed with Larceny's source code.[6] These benchmarks were written by various people, collected by Will Clinger and Marc Feeley, and translated into R6 Scheme by Abdulaziz Ghuloum and Will Clinger.

For those standard benchmarks, Larceny runs about 12% slower overall (geometric mean) with the regional collector than with Larceny's default generational collector. About half of that slowdown is attributable to the regional collector's use of a 1-megabyte nursery instead of the generational collector's 4-megabyte nursery.

When Larceny's stop&copy collector is used instead of its generational collector, Larceny runs about 23% slower overall. We therefore conclude that, for a typical mix of Scheme programs, the regional collector is likely to have better throughput than the stop&copy collector.

The most comparable figures that we have seen are for the Gambit implementation of Scheme. Compared to Gambit's non-generational collector, Gambit's real-time collector had more overhead on every benchmark [29]. From that, we conclude that Larceny's regional collector has better throughput than Gambit's real-time collector. On the other hand, the worst-case pause times observed for Larceny's regional collector are nowhere near as short as the 15 millisecond maximum observed for Gambit's real-time collector.

On the standard benchmarks, the regional collector's longest pause time was .24 seconds. For that benchmark, the generational collector reported a pause of .72 seconds. Across all 68 benchmarks, the generational collector's longest pause was 1.88 seconds.

8.2 GC-intensive synthetic benchmarks

To compare the near-worst-case performance of Larceny's collectors against collectors available for Java, we used four extraordinarily gc-intensive synthetic benchmarks. All four benchmarks were carefully written to provide fair comparisons between the two languages. (Among other things, this involved using Scheme vectors with two elements when lists or pairs would be more idiomatic. In Java, an object that encapsulates two reference fields is likely to occupy 4 words of memory, as does a 2-element vector in Larceny, but Larceny's pairs occupy only 2 words each.)

`gcbenchJ` is a scalable version of the synthetic benchmark originally written in Java by John Ellis, Pete Kovac, and Hans Boehm. That original benchmark corresponds to `1gcbenchJ:18` of the scalable benchmark. We benchmarked `5gcbenchJ:24`, which

[5] `http://www.larcenists.org/`

[6] `http://www.larcenists.org/benchmarks2009.html`

language	gc technology	cpu time (sec) total	cpu time (sec) mark+summ+gc	elapsed time	memory (MB) total	memory (MB) heap	# of gc pauses total	# of gc pauses full	pause time (sec) avg full	pause time (sec) max
Scheme	regional	726	121+34+287	757	1871	1775	158800	0		.12
Scheme	generational	372	188	380	1867	1863	39756	276		3.13
Scheme	stop&copy	419	233	428	1881	1877	711	711	.33	2.94
Java	default	133	67	116		1751	541	20	1.13	3.02
Java	parallel	131	65	114		1838	530	21	1.00	2.78
Java	garbage-first	245	128	154		2147	1514	5	1.44	2.13
Java	concurrent m/s	672	526	450		2144	6228	5	13.64	15.45

Figure 6. Timings and memory usage for the `5gcbenchJ:24` benchmark.

language	gc technology	cpu time (sec) total	cpu time (sec) mark+summ+gc	elapsed time	memory (MB) total	memory (MB) heap	# of gc pauses total	# of gc pauses full	pause time (sec) avg full	pause time (sec) max
Scheme	regional	160	41+5+87	164	2272	2106	8239	0		.13
Scheme	generational	73	54	74	1632	1628	2059	27		3.23
Scheme	stop&copy	54	38	55	1633	1629	26	26	1.45	3.44
Java	default	136	130	90		2097	71	22	2.21	3.07
Java	parallel	131	125	82		2097	68	22	2.14	2.93
Java	garbage-first	198	183	112		2334	726	5	4.19	4.68
Java	concurrent m/s	221	141	148		2356	430	0		.50

Figure 7. Timings and memory usage for the `400permJ:9:30:1` benchmark.

language	gc technology	cpu time (sec) total	cpu time (sec) mark+summ+gc	elapsed time	memory (MB) total	memory (MB) heap	# of gc pauses total	# of gc pauses full	pause time (sec) avg full	pause time (sec) max
Scheme	regional	186	63+1+102	191	1925	1826	15260	0		.09
Scheme	generational	98	86	101	2044	2039	3815	47		4.28
Scheme	stop&copy	85	72	88	2051	2047	48	48	1.49	4.62
Java	default	239	233	180		1771	124	42	2.49	3.00
Java	parallel	238	232	176		1771	124	42	2.49	3.24
Java	garbage-first	345	334	205		1992	1049	12	4.27	4.29
Java	concurrent m/s	456	270	326		1989	667	0		.45

Figure 8. Timings and memory usage for the `1000queueJ:1000000:50` benchmark.

language	gc technology	cpu time (sec) total	cpu time (sec) mark+summ+gc	elapsed time	memory (MB) total	memory (MB) heap	# of gc pauses total	# of gc pauses full	pause time (sec) avg full	pause time (sec) max
Scheme	regional	320	97+48+156	326	1924	1829	15260	0		.21
Scheme	generational	107	95	110	2044	2039	3815	47		4.45
Scheme	stop&copy	90	76	92	2054	2049	48	48	1.58	4.74
Java	default	254	248	189		1771	124	42	2.71	3.31
Java	parallel	251	245	190		1771	124	42	2.73	3.32
Java	garbage-first	445	432	254		1992	1151	12	5.32	5.84
Java	concurrent m/s	444	275	318		1989	667	0		5.94

Figure 9. Timings and memory usage for the `1000pueueJ:1000000:50:50` benchmark.

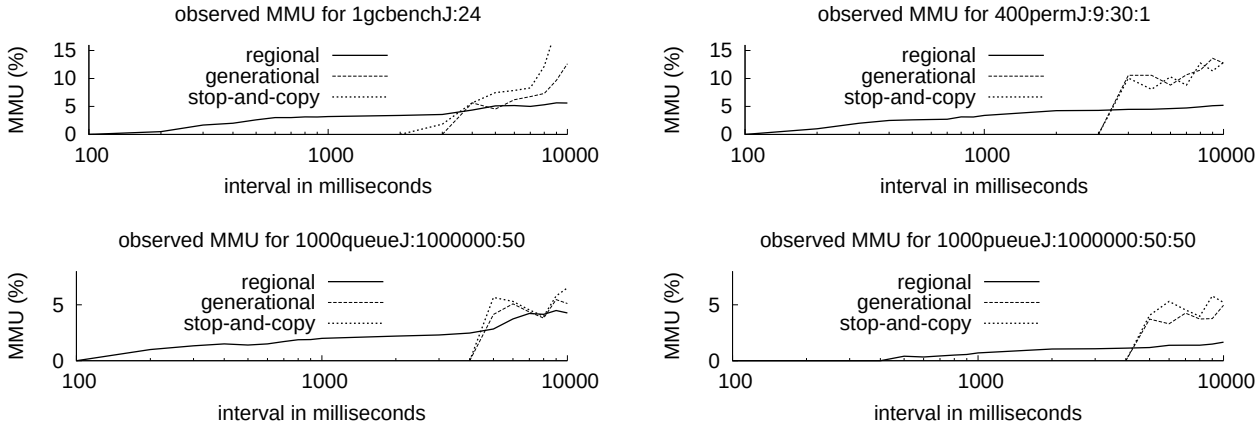

Figure 10. Observed MMU for four near-worst-case synthetic benchmarks.

performs 5 iterations scaled for a 2 gigabyte heap (the original 32 megabytes times 2^{24-18}).

`400permJ:9:30:1` consists of 400 iterations that each generate a list (built out of 2-element vectors in Scheme) of all permutations of 9 items, with much shared structure, without generating any garbage, storing the results into a circular buffer of capacity 30; garbage is created only when the oldest list in the buffer is replaced by the most recent copy.

`1000queueJ:1000000:50` allocates 1000 lists (built out of 2-element vectors in Scheme), with one million elements per list, storing them in a circular buffer that holds 50 lists. `pueueJ` is the same as `queueJ` except that every element of the lists is a popular object. With `1000pueueJ:1000000:50:50`, there are 50 popular objects, one for each list.

Figures 6 through 9 show the results. Larceny's regional collector has the shortest maximum pause times, by far, mostly because it's the only collector that never performs a full collection.

The regional collector normally uses about as much memory as Larceny's generational collector, but it uses more memory on the `permJ` benchmark.

Incremental and/or concurrent collection comes at a price: On these extremely gc-intensive benchmarks, the regional collector delivers roughly half the throughput of Larceny's generational collector. Oracle's garbage-first and concurrent mark/sweep collectors also provide less throughput than the JVM's default collector.

Oracle's JVM outperforms Larceny on `gcbenchJ` because that benchmark rewards large nurseries and fast write barriers. The other three benchmarks penalize large nurseries. (Nurseries improve average-case performance, but large nurseries just waste space in the worst case because they don't help with the long-lived objects that dominate truly extreme benchmarks.) On those three benchmarks, Larceny with its regional collector consumes less cpu time than Java with the garbage-first or concurrent mark/sweep collectors. If the regional collector's marking and summarization processes were executed in a separate thread, then the regional collector's total elapsed time for two of those benchmarks would be better than the elapsed time for the garbage-first and concurrent mark/sweep collectors, and its elapsed time for the third would be comparable.

8.3 Observed MMU

As shown by Figures 3 and 10, the regional collector delivers positive minimum mutator utilization at resolutions for which the JVM collectors and Larceny's other collectors have zero MMU.

The MMUs seen in Figures 3 and 10 are appallingly low, but these graphs are for near-worst-case benchmarks. For `pueueJ`, the JVM's default collector delivers an *average* mutator utilization of only 2.5%. The minimum mutator utilization is always less than the average, and the MMU observed over short intervals is usually much less than the average. Larceny's generational and stop&copy collectors are able to achieve an MMU of 5% at 10 seconds for `pueueJ` only because Larceny's compiler generates machine code that's about twice as slow as the machine code generated by the JVM.

The `paraffins` benchmark may be even more severe than `pueueJ`. With the inverse load factor of $L = 2.5$ that was used to generate Figure 3, Larceny's stop&copy collector delivers an average mutator utilization of only 0.25%, while the generational and regional collectors come in at about 4%.

The regional collector's pause times and MMU are more predictable than the pause times and MMU of the other collectors. With the other collectors, you can't be sure they won't perform a full collection at some inopportune moment.

General-purpose garbage collectors have poor worst-case MMU. Although the regional collector's worst-case MMU is not as good as we would like, it improves upon the worst-case MMU of previous collectors.

9. Related work

9.1 Generational garbage collection

Generational collection was introduced by Lieberman and Hewitt [31]. A simplification of that design was first implemented by Ungar, who also introduced a nursery [41]. Most modern generational collectors resemble Ungar's, but our regional collector's design is more similar to that of Lieberman and Hewitt.

9.2 Heap partitioning

Our regional collector partitions the heap and collects the parts independently.

Bishop's collector allows single areas to be collected independently; his work targets Lisp machines and requires hardware support [8].

The *Garbage-First* collector inspired many aspects of our regional collector [20]. Unlike the garbage-first collector, which uses a points-into remembered set that could grow very large in a worst case, we use a points-out-of remembered set with points-into summaries that are bounded in size. The garbage-first collector moves popular objects; we do not. The garbage-first collector is not scalable in the sense defined by Sections 4 and 5: It does not offer worst-case bounds on space usage, pause times, or MMU.

The *Mature Object Space* (a.k.a. *Train*) algorithm uses a fixed policy for choosing which regions to collect [26]. To ensure completeness, it migrates objects across regions until a complete cycle is isolated to its own train and then collected. This gradual migration can lead to significant problems with floating garbage. Our marking process eliminates floating garbage in collected regions, while our handling of popular regions provides an elegant and novel solution that bounds the worst-case storage requirements. The Mature Object Space collector is not scalable.

The *Beltway* collector uses heap partitioning and clever infrastructure to enable flexible selection of collection policies via command line options [9]. Its policy selection is expressive enough to emulate the behavior of semi-space, generational, renewal-older-first, and deferred-older-first collectors. Appropriate choices for mutator-specific policy parameters improved performance by 5%, 10%, and up to 35% over a fixed generational collection policy. The Beltway system forces users to choose between incremental or complete collection, so the Beltway collector is not scalable.

The *MarkCopy* collector breaks the heap down into fixed sized *windows* [36]. During a collection pause, it builds up a remembered set for each window and then collects each window in turn. An extension interleaves the mutator process with individual window copy collection; one could see our design as taking the next step of moving the marking process and remembered set construction off of the critical path of the collector.

The Parallel Incremental Compaction algorithm also has similarities to our approach [7]. It selects an area of the heap to collect, and then concurrently builds a summary for that area. Its points-into summary set is constructed by tracing the whole heap, rather than by maintaining a points-out-of remembered set as in our implementation of the regional collector. (That technique would work with the regional collector as well, and might be a welcome simplification. Klock evaluated that alternative in his dissertation [28].) Their goals are also different from ours: Their technique adds incremental compaction to a mark-sweep collector, while we provide utilization and space guarantees in a copying collector.

9.3 Older-first garbage collection

Our regional collector, like older-first collectors, tends to give objects more time to die before attempting to collect them [23, 37].

9.4 Bounding collection pauses

There is a broad body of research on bounding the pause times introduced by garbage collection [2, 5, 6, 11, 12, 24, 32, 42]. In particular, Blelloch and Cheng proved worst-case bounds for pause times and space usage (but not MMU) [10].

Bounding individual pause times is not enough; one must also ensure that the mutator can accomplish a sufficient work between the pauses, keeping the processor utilization as high as possible. Cheng and Blelloch addressed this issue by inventing the MMU metric [14]. Their paper presented an *observed* MMU for a parallel real-time collector, not a theoretical worst-case MMU.

9.5 Collection scheduling

Metronome is a hard real-time collector [4]. It can use either time- or work-based collection scheduling, and is mostly non-moving, but will copy objects to reduce fragmentation. Metronome also requires a read barrier. Although the average overhead of the read barrier is only 4%, mutator utilization is said to be limited to about 50% [15]. More significantly, Metronome's guaranteed bounds on utilization and space usage depend upon the accuracy of application-specific parameters. The original set of parameters has been extended to provide tighter bounds on collection time and space overhead [3]. Because its parameters depend upon the mutator, Metronome is not scalable in the sense defined by Sections 4 and 5.

Similarly, Robertz and Henriksson described a collector that depends on a supplied schedule to provide real-time collector performance [35]. Unlike Metronome, it schedules work according to collection cycle times rather than finer grained quanta. Like Metronome, it provides a proven bound on space usage (that depends on the accurary of application-specific parameters).

In contrast to those designs, our regional collector is scalable: It provides worst-case guarantees independent of mutator behavior. On the other hand, our regional collector cannot guarantee pause times or MMU in the millisecond range. Our regional collector is mostly copying, has no read barrier, and uses work-based accounting to drive the collection policy.

9.6 Incremental and concurrent collection

There are many treatments of concurrent collectors, including an algorithm described in 1978 [21].

The Continuously Concurrent Compacting Collector (C4) is a generational form of The Pauseless GC Algorithm [15, 38]. Both of those collectors have goals similar to ours, but attack the problem differently. Where we avoid read barriers entirely, those collectors implement a read barrier in custom hardware or by replacing Linux's virtual memory system with a custom virtual memory system that greatly reduces the cost of using stock memory protection hardware to trap reads from problematic locations. It is unclear whether those collectors are scalable in the sense defined by Sections 4 and 5. Their observed pause times and MMU are impressive, but no proofs have been published and no claims have been made concerning their theoretical worst-case MMU or space requirements.

In our collector, reclamation of dead object state is not performed concurrently with the mutator, but the activity of the summarization and marking processes could be.

Our summarization process was inspired by the performance of Detlefs' implementation of a concurrent thread that refines data within the remembered set to reduce the effort spent towards scanning older objects for roots during a collection pause [19].

Interleaving the summarization and marking processes with the mutator requires a write barrier, which we piggy-back onto a barrier that was already in place to support generational collection. This is similar to how Printezis and Detlefs, building on the work of Boehm *et al.*, merged the overhead of maintaining concurrency related invariants with the overhead of maintaining generational invariants [11, 34].

10. Future work

As implemented in Larceny v0.98b1, the regional collector interleaves the marking and summarization processes with the mutator. Summarization is scheduled at the fine grain of Larceny's software timer interrupts, while marking is scheduled for minor collections and the processing of write barrier logs. Both marking and summarization could be done concurrently with the mutator, which would improve throughput on programs that do not fully utilize all processor cores.

The regional collector's Cheney collections can themselves be parallelized, but that is essentially independent of the design.

We assume object sizes are bounded, so every object will fit into a region. Because we control both the compiler and the runtime representations of objects, we can choose representations that break extremely large objects into pieces of bounded size. We have not yet done that, but expect it to be routine.

11. Conclusions

Regional garbage collection is scalable, with theoretical worst-case bounds for gc latency, MMU, and throughput that are independent of mutator behavior and the volume of reachable storage.

Regional collection improves upon the worst-case pause times and MMU seen in most other general-purpose collectors, including the garbage-first and concurrent mark/sweep collectors. That improvement involves some sacrifice of throughput, but regional collection still tends to deliver better throughput than Larceny's non-generational collector.

12. Acknowledgements

We thank our reviewers for their comments. We thank the program chair for giving us enough space to incorporate the reviewers' suggestions.

References

[1] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

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

[3] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In LCTES [30], pages 81–92.

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[5] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[6] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[7] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International*

Symposium on Memory Management, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.

[8] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[9] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [33], pages 153–164.

[10] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN 1999 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

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

[12] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[13] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[14] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[15] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM.

[16] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, April 1999.

[17] William D. Clinger and Felix S. Klock. Scalable garbage collection with guaranteed MMU. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, pages 14–25, 2009. Online at `http://www.cesura17.net/~will/Professional`.

[18] David Detlefs. A hard look at hard real-time garbage collection. In *Proceedings of 7th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, May 2004.

[19] David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[20] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Amer Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

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

[22] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In PLDI [33], pages 141–152.

[23] Lars Thomas Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 247–258, Pittsburgh, PA, 2002. ACM Press.

[24] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[25] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, 1990.

[26] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[27] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[28] Felix S. Klock. *Scalable Garbage Collection via Remembered Set Summarization and Refinement*. PhD thesis, Northeastern University, January 2011. Online at `http://www.larcenists.org/research.html`.

[29] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 1–9, New York, NY, USA, 1998. ACM.

[30] *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[31] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[32] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[33] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[34] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[35] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In LCTES [30], pages 93–102.

[36] Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[37] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. Mckinley, J. Eliot, and B. Moss. Older-first garbage collection in practice: Evaluation in a java virtual machine. In *In Memory System Performance*, pages 25–36. ACM Press, 2002.

[38] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[39] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.

[40] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997.

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

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