Scalable Garbage Collection with Guaranteed MMU

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

Felix S. Klock II
Northeastern University
pnkfelix@ccs.neu.edu

Abstract

Regional garbage collection offers a useful compromise between real-time and generational collection. Regional collectors resemble generational collectors, but are scalable: our main theorem guarantees a positive lower bound, independent of mutator and live storage, for the theoretical worst-case minimum mutator utilization (MMU). The theorem also establishes upper bounds for worst-case space usage and collection pauses.

Standard generational collectors are not scalable. Some real-time collectors are scalable, while others assume a well-behaved mutator or provide no worst-case guarantees at all.

Regional collectors cannot compete with hard real-time collectors at millisecond resolutions, but offer efficiency comparable to contemporary generational collectors combined with improved latency and MMU at resolutions on the order of hundreds of milliseconds to a few seconds.

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

We have designed and prototyped a new kind of scalable garbage collector that delivers a provable fixed upper bound for the duration of collection pauses. This theoretical worst-case bound is completely independent of the mutator (defined as the non-gc portion of an application) and the size of its data.

The collector also delivers a provable fixed lower bound for worst-case minimum mutator utilization (MMU, expressed as the smallest percentage of the machine cycles that are available to the mutator during any sufficiently long interval of time) and a simultaneous worst-case upper bound for space, expressed as a fixed multiple of the mutator's peak storage requirement.

These guarantees are achieved by sacrificing throughput on unusually gc-intensive programs. For most programs, however, the loss of throughput is small. Indeed, our prototype's overall throughput remains competitive with several generational collectors that are currently deployed in popular systems.

Section 5 discusses one near-worst-case benchmark. To reduce this paper to an acceptable length, we defer most discussion of more typical programs, and of throughput generally, to another paper that will also describe the engineering of our prototype in greater detail.

Worst-case performance, both theoretical and observed, is the focus of this paper. Many garbage collectors have been designed to exploit common cases, with little or no concern for the worst case. As illustrated by section 5, their worst-case performance can be quite poor. When designing our new *regional* collector, our main goal was to guarantee a minimal level of performance, independent of problem size and mutator behavior. We exploit common cases only when we can do so without compromising latency or asymptotic performance for the worst case.

1.1 Bounded Latency

Generational collectors that rarely stop the mutator while they collect the entire heap have worked well enough for many applications, but that paradigm breaks down for truly large heaps: even an occasional full collection can produce alarming or annoying delays (Nettles and O'Toole 1993). This problem is evident on 32-bit machines, and will only get worse as 64-bit machines become the norm.

Real-time, incremental, or concurrent collectors can eliminate those delays, but at significant cost. On stock hardware, most bounded-latency collectors depend upon a read barrier, which reduces throughput (average mutator utilization) even for programs that create little garbage. Read barriers and other invariants also increase the complexity of compilers and run-time infrastructure, while impeding use of libraries that were written and compiled without knowledge of the garbage collector's invariants.

Our regional collector is a novel bounded-latency collector whose invariants resemble the invariants of standard generational garbage collectors. In particular, our regional collector does not require a read barrier.

1.2 Scalability

Unlike standard generational collectors, the regional collector is *scalable*: Theorem 1 below establishes that the regional collector's theoretical worst-case collection latency and MMU are bounded by nontrivial constants that are independent of the volume of reachable storage and are also independent of mutator behavior. The theorem also states that these fixed bounds are achieved in space bounded by a fixed multiple of the volume of reachable storage.

Although most real-time, incremental, or concurrent collectors appear to be designed for embedded systems in which they can be tuned for a particular mutator, some (though not all) hard real-time collectors are scalable in the same sense as the regional collector. Even so, we are not aware of any published proofs that establish all three scalability properties of our main theorem for a hard real-time collector.

The following theorem characterizes the regional collector's worst-case performance.

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 $c_2P + c_3$.*

We must emphasize that the constants c_0, c_1, c_2, and c_3 are completely independent of the mutator. Their values do depend upon several parameters of the regional collector, upon details of how the collector is implemented in software, and upon the hardware used to execute the mutator and collector. Later sections will discuss the worst-case constants and report on the performance actually observed for one near-worst-case benchmark.

Major contributions of this paper include:

- a new algorithm for scalable garbage collection
- a proof of its scalability, independent of mutator behavior
- a novel solution to the problem of popular objects
- formulas that describe how theoretical worst-case performance varies as a function of collector parameters
- empirical measurements of actual performance for one near-worst-case benchmark

The remainder of this paper describes the processes, data structures, and algorithms of the regional collector, provides a proof of our main theorem above, estimates worst-case bounds, and summarizes related and future work.

2. Regional Collection

The regional collector resembles a stop-the-world generational collector with several additional data structures, processes, and invariants.

In place of generations that segregate objects by age, the regional collector maintains a set of relatively small regions, all of the same size R. 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 assumes every object is small enough to fit within a region. For justification, see sections 3.4 and section 7.

The regional collector maintains a remembered set, a collection of summary sets, and a snapshot structure. Each component is described in detail below, after an overview of the memory management processes. In short, the remembered set tracks region-crossing references, the summary sets summarize portions of the remembered set that will be relevant to upcoming collections, and the snapshot structure gathers past reachability information to refine the remembered set.

The interplay between regions, the remembered set and the summary sets is an important and novel aspect of our design.

2.1 Processes

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

- a collection process uses the Cheney (1970) algorithm to move a region's reachable storage into some other region(s),
- a summarization process computes summary sets from the remembered set, and
- a snapshot-at-the-beginning marking process marks every object reachable in a snapshot of the object graph.

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

The collection and marking processes serve distinct purposes. The collection process moves objects to prevent fragmentation, and updates pointers from outside the collected region to point to the newly relocated objects; it also reclaims unreachable storage.[1]

The pointers that must be updated during a relocating collection reside in uncollected regions, in the marking process's snapshot structure, and in the mutator stack(s); the latter are discussed in sections 2.6 and 2.8 respectively.

The summarization process constructs *summary sets* in preparation for collections, and is the subject of section 2.3.

The regional collector imposes a fixed constant bound on the duration of each collection. That means that a *popular* region, whose summary set is larger than a fixed threshold, would take too long to collect. Section 3.3 proves that, with appropriate values for the collector's parameters, the percentage of popular regions is so well bounded that the regional collector can afford to leave popular regions uncollected. That is one of the critical lemmas that establish the scalability of regional garbage collection.

The main purpose of the marking process is to limit unreachable storage to a bounded fraction of peak live storage; it accomplishes that by removing unreachable references from the remembered set. The marking process also calculates the volume of reachable storage at the time of its initiation; without that information, the collector might not be able to guarantee worst-case bounds for its storage requirements.

2.2 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*, which contains every location (or object) that points from one region to a different region. A similar structure is a standard component of generational collectors.

The mutator can create such region-crossing pointers by allocation or assignment. The collector can create region-crossing pointers by relocating an object from one region to another.

The remembered set is affected by 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 regional collector represents its remembered set using a data structure that records at most one entry for each location in the heap (e.g. a hash table or fine-grain card table suffices). The size of the remembered set's representation is therefore bounded by the size of the heap, even though the remembered set is imprecise.

2.3 Summary Sets

A typical generational collector will scan most (or all) of the remembered set during collections of the younger portions of the heap. In the worst case the remembered set can grow proportional to the heap; hence this technique would not satisfy our pause time bounds, and is not an option for the regional collector.

[1] The collection process is the *only* process permitted to move objects. The summarization and marking processes do not change the correspondence between addresses and objects; hence neither interferes with the other's view of the heap (nor the mutator's view), even if run concurrently.

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.

If an imprecise remembered set were organized as a set of summary sets, one for each region, then the collector would not be scalable: in the worst case, the storage occupied by those summary sets would be proportional to the number of regions times the size of the heap. Since regions are of fixed constant size, the summary sets could occupy storage proportional to the square of the heap size. That is why the regional collector uses a remembered set representation that records pointers that come out of a region instead of pointers that go into the region.

There are two distinct issues to address regarding the use and construction of summary sets.

First, the regional collector *must* compute a region's summary set before it can collect the region. But a naïve construction could take both time and space proportional to the size of the heap, which would violate our bounds.

Second, in the worst case, a summary set for a region may consist of all locations in the heap. That means that a *popular* region, defined as a region whose summary set is larger than a fixed threshold, would take too long to collect.

To address these two issues, and thus keep time and space under control, the summarization process

- amortizes the cost in time by incrementally computing multiple summary sets for a fixed fraction $1/F_1$ of the heap's regions, but
- abandons the computation of any summary set whose size exceeds a fixed wave-off threshold (expressed as a multiple S of the region size R).

Waving off summarization raises the question: when do popular regions get collected? Our answer, inspired by Detlefs et al. (2004), is simple: such regions are not collected.[2] Instead we bound the percentage of popular regions to ensure that the regional collector can afford to leave popular regions uncollected. See sections 3.2 and 3.3.

2.4 Nursery

Like most generational collectors, the regional collector allocates all objects within a relatively small *nursery*. The nursery has little impact on worst-case performance, so our proofs ignore it. For most programs, however, the nursery greatly improves the observed MMU and overall efficiency of the regional collector.

Since the nursery is collected as part of every collection, locations within the nursery that point outside the nursery do not need to be added to the remembered set.

Pointers from a region into the nursery can be created only by assignments. Those pointers are recorded in a special summary set, which is updated by processing of write barrier logs. If the size of that summary set exceeds a fixed threshold, then the regional collector forces a minor collection that empties the nursery, promoting survivors into a region.

2.5 Grouping Regions

Figure 1 depicts how regions are partitioned into five groups: { **ready**, **unfilled**, **filled**, **popular**, **summarizing** }. In the figure, each small rectangle is a fixed-size region, the tiny ovals are objects allocated within a region, and the triangular "hats" atop some of the regions are summary sets. The dotted hats are under construction, while the filled hats are completely constructed.

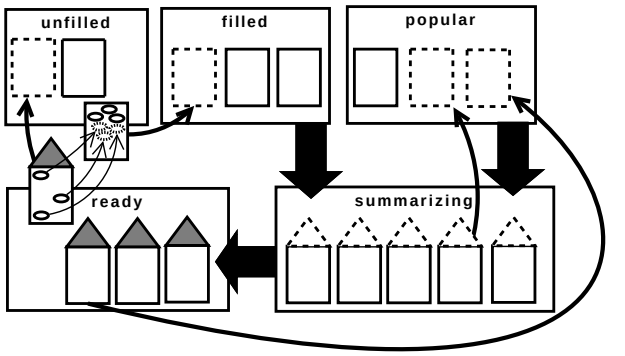

Figure 1. Grouping and transition of regions

The thinnest arcs in the figure, connecting small ovals, represent migration of individual objects during a major collection; that is the *only* time at which objects move from one region to another. Arcs of medium thickness represent transitions of a single region from one group to another, and the thickest arcs represent transitions of many regions at once.

At all times, one of the unfilled regions is the current *to-space*; it may contain some objects, but all other regions in the unfilled group are empty.

Four of the arcs form a cycle that describes the usual transitions of a region:

(ready, unfilled) On each major collection, one region (the *from-space*) is selected from the **ready** group. All of its reachable objects are forwarded to unfilled region(s) via Cheney's algorithm (the thinnest arcs). After object forwarding is complete, the now empty region is reclassified as **unfilled**.

(unfilled, filled) When the collector fills the current to-space region to capacity, it is reclassified as **filled**, and another unfilled region is picked to be the new to-space.

(filled, summarizing) The summarization process starts its cycle by reclassifying a subset of regions *en masse* as **summarizing**, preparing them for future collection.

(summarizing, ready) At the end of a summarization cycle the summarized regions become **ready** for collection.

The remaining three arcs in the diagram describe transitions for **popular** regions:

(summarizing, popular) As the summarization process passes over the remembered set, it may discover that a summary set for a particular region is too large: i.e., the region has too many incoming references to be updated within the pause time bound. The sumarization process will then remove that region from the summarizing group, and deem that region **popular**.

(ready, popular) Mutator activity can increase the number of incoming references to a **ready** region, to the point where it has too many incoming references to be updated within the pause time bound. Such regions are likewise removed from the **ready** group and become **popular**.

(popular, summarizing) Our collector does *not* assume that popular regions will remain popular forever. At the start of a summarization cycle, **popular** regions can be shifted into the **summarizing** group, where their fitness for collection will be re-evaluated by the summarization process.

[2] Our strategy is subtly different from Detlefs et al. (2004); Garbage-First migrates popular *objects* to a dedicated space; that still requires time proportional to the heap size in the worst case. We do not migrate the popular objects at all.

2.6 Snapshots

The remembered set is imprecise. To bound its imprecision, a periodic snapshot-at-the-beginning (Yuasa 1990) marking process incrementally constructs a snapshot of the heap at a particular point in time. The resulting snapshot classifies every object as either unreachable or live/unallocated at the time of the snapshot.

The marking process incrementally traces the snapshot's object graph; objects allocated after the instant the snapshot was initiated are considered live by the snapshot and are not traced by the marking process. Objects relocated by the Cheney algorithm retain their current unreachable/live classification in the snapshot.

When the marking process completes snapshot construction, it removes dead locations from the remembered set. This increases remembered set precision, reducing the amount of floating garbage; in particular, it ensures that cyclic garbage across different regions is eventually removed from the remembered set.

The developing snapshot has a frontier of objects remaining to be processed, called the *mark stack*. The regional collector treats the portion of the mark stack holding objects in the collected region as an additional source of roots. In order to ensure that collection pauses only take time proportional to the size of a region, each regions' substacks are threaded through the single mark stack, and the collector scans *only* the portion of the stack relevant to a particular region.

2.7 Write Barrier

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

The regional collector uses a variant of a Yuasa-style logging write barrier (Yuasa 1990). Our write barrier logs three things: (1) the location on the left hand side of the assignment, (2) its previous contents, and (3) its new contents.

The first is for remembered set and summary set maintenance. The second is for snapshot maintenance (the marker). The third identifies which summary set (if any) needs maintenance for the log entry.

2.8 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 (Appel 1992). Although mixed stack/heap, incremental stack/heap, Hieb-Dybvig-Bruggeman, and Cheney-on-the-MTA strategies are often used (Clinger et al. 1999; Hieb et al. 1990), 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.

3. Collection Policies

This section describes the policies the collector follows to achieve scalability, even in the worst case.

Some of the policies are parameterized by numerical parameters: F_1 (described in Section 2.3), F_2 (3.2), F_3 (3.2), R (3.3), S (3.3), L_{soft} and L_{hard} (3.6). See section 5 for typical values. These parameters provide implementors with valuable flexibility, but we assume that the values of these parameters will be fixed by the implementors of a regional collector, and will not be tailored for particular mutators.

3.1 Minor, Major, Full, and Mark Cycles

The nursery is collected every time a region is collected, but the nursery may also be collected without collecting a region. A collection that collects only the nursery is a *minor collection*. A collection that collects both the nursery and a region is a *major collection*.

The interval between successive collections, whether minor or major, is a *minor cycle*. The interval between major collections is a *major cycle*.

The interval between successive initiations of the summarization process is a *summarization cycle*.

Regions are ordered arbitrarily, and collected in roughly round-robin fashion (see Figure 1), skipping popular and empty (unfilled) regions. When all non-popular, non-empty regions have been collected, a new *full cycle* begins.

The snapshot-at-the-beginning marking process is initiated at the start of a new full cycle. The interval between successive initiations of the marking process is a *mark cycle*.

Our proofs assume that mark and full cycles coincide, because worst-case mutators require relatively frequent marking (to limit the size of the remembered set and to reduce floating garbage). On normal programs, however, the mark cycle may safely be several times as long as a full cycle.

Usually there are F_1 summarization cycles per full cycle, but that can drop to F_1/F_3; see Section 3.3.

The number of major collections per full cycle is bounded by the number of regions N/R, where N is the total size of all regions.

The number of minor collections per major cycle is mostly determined by the promotion rate and by two parameters that express the desired (soft) ratio and a mandatory hard bound on N divided by the peak live storage.

3.2 Summarization Details

If the number of summary sets computed exceeds a fixed fraction $1/(F_1F_2)$ of the heap's regions, then the summarization process can be suspended until one of the regions associated with the newly computed summary sets is scheduled for the next collection.

If on the other hand the summarization process has to wave off the construction of too many summary sets, then the summarization process makes another pass over the remembered set, computing summary sets for a different group of regions. The maximum number of passes that might be needed before $1/(F_1F_2)$ of the heap's regions have been summarized is a parameter F_3 whose value depends upon parameters S, F_1, and F_2; see section 3.3.

Mutator actions can change which regions are classified as popular; popular regions can become unpopular, and vice versa. To prevent this from happening at a faster rate than the collection and summarization processes can handle, the mutator's allocation and assignment activity must be linked to collection and summarization progress (measured by the number of regions collected and progress made toward computation of summary sets).[3] As explained in 4.2, this extremely rare contention between the summarization process and the mutator determines the theoretical worst-case MMU of the collector.

When a region is collected, its surviving objects move and its other objects disappear. Entries for reclaimed objects must be removed from all existing summary sets, and entries for surviving objects must be updated to reflect the new addresses. A good representation for summary sets allows this updating to be done in time proportional to the size of the collected region.

3.3 Popular Regions

Suppose there are N/R regions, each of size R, so the total storage occupied by all regions is N.

Definition 2. *A region is* popular *if its summary set would exceed S times the size of the region itself, where S is the collector's wave-off threshold.*

[3] This leads to a curious property: in a regional collector, allocation-free code fragments containing assignment operations can cause a collection (and thus object relocation).

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

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

Proof. If there were more than $1/S$ popular regions, then the total size of the summary sets for all popular regions would be greater than

$$\frac{1}{S}\frac{N}{R}SR = N$$

That is impossible: there are only N words in all regions combined, so how could more than N words be pointing into the popular regions? □

Example: If $S = 3$, then at most $1/3$ of the regions are popular, and not collecting those popular regions will add at most 50% to the size of the heap.

Corollary 4. *Suppose marking cycles coincide with full cycles, and a new full cycle is about to start. Let P_{old} be the volume of reachable storage, as computed by the marking process, at the start of the previous full cycle, and let A be an upper bound on the storage allocated during the previous full cycle. If $S > 1$, then the fraction of regions that are popular is no greater than*

$$\frac{P_{old} + A}{S}$$

Mutator activity can make previously popular regions unpopular, and can make previously unpopular regions popular, but the number of new pointers into a region is bounded by the number of words allocated plus the number of distinct locations assigned. Furthermore the fraction of popular regions can approach $1/S$ only if there are very few pointers into the unpopular regions. That means the mutator would have to do a lot of work before it could prevent a second or third pass of the summarization process from succeeding, provided of course that the collector's parameters are well-chosen.

Recall that the summarization process attempts to create summary sets for $1/F_1$ of the regions in each pass, and that it keeps making those passes until it has created summary sets for $1/(F_1F_2)$ of the regions.

Lemma 5. *Suppose S, F_1, and F_2 are greater than 1, and F_3 is a positive integer. Suppose also that*

$$c = \frac{F_2F_3 - 1}{F_1F_2}S - 1 > 0$$

and the mutator is limited to cN words allocated plus distinct locations assigned while the summarization process is performing up to F_3 passes. Then F_3 passes suffice.

Proof. We calculate the smallest number of allocations and assignments cN that would be required to leave at least i regions popular at the end of the summarization cycle. If i is less than or equal to the bound given by lemma 3, then no allocations/assignments are needed. Otherwise the smallest number of allocations/assignments occurs when the bound given by lemma 3 is met at both the beginning and end of the summarization cycle.[4] If that bound is met at the beginning of the cycle, then all non-popular regions have no pointers into them, and it takes SR allocations/assignments to create another popular region.

[4] In other words, starting with fewer popular regions *increases* the mutator activity required to end the cycle with large i; we are deriving the *minimum* number of actions required.

The summarization process will compute usable summaries for at least $1/(F_1F_2)$ of all N/R regions if

$$\frac{1}{F_1F_2}\frac{N}{R} \leq \frac{F_3}{F_1}\frac{N}{R} - \frac{1}{S}\frac{N}{R} - \frac{cN}{SR}$$

Equivalently

$$\begin{aligned} c &\leq \left(\frac{F_3}{F_1} - \frac{1}{S} - \frac{1}{F_1F_2}\right)S \\ &= \frac{F_2F_3 - 1}{F_1F_2}S - 1 \end{aligned}$$

□

That lemma, when combined with an upper bound for the duration of a collection, basically determines the theoretical worst-case MMU. See section 4.2.

For simplicity, we will henceforth assume that F_1/F_3 is an integer.

The following lemma bounds the number of regions that will not be collected during a full cycle.

Lemma 6. *Within any full cycle, the fraction of regions whose summary sets are not computed by the summarization process is no greater than*

$$1 - \frac{1}{F_2F_3}$$

Proof. Each summarization cycle makes up to F_3 passes, summarizing $1/F_1$ of the regions in each pass over the remembered set, to obtain at least $1/(F_1F_2)$ usable summary sets. In the worst case, there are F_1/F_3 summarization cycles in a full cycle. The largest possible fraction of unusable summary sets is therefore

$$\frac{F_1}{F_3}\left(\frac{F_3}{F_1} - \frac{1}{F_1F_2}\right) = 1 - \frac{1}{F_2F_3}$$ □

Each major collection consumes one summary set. The worst-case MMU is calculated by assuming each summary cycle yields only

$$\frac{1}{F_1F_2} \cdot \frac{N}{R}$$

usable summaries. The worst-case MMU is therefore unaffected by starting each summarization cycle when the number of summary sets has been reduced to the value used to calculate the worst-case MMU.

Corollary 7. *The space occupied by summary sets is never more than*

$$\frac{SF_3}{F_1}N$$

Proof. During any summarization cycle, the space occupied by the summary sets being computed is bounded by $N + cN$. Hence the total space occupied by all summary sets is bounded by

$$\begin{aligned} &\left(\frac{1}{F_1F_2} \cdot \frac{N}{R}\right)SR + N + cN \\ = &\frac{SN}{F_1F_2} + N + \left(\frac{F_2F_3 - 1}{F_1F_2}S - 1\right)N \\ = &\frac{SF_3}{F_1}N \end{aligned}$$

□

3.4 Fragmentation

As was mentioned in section 2 and justified in section 7, 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. Our calculations assume that ratio is negligible.

3.5 Work-Based Accounting

The regional collector performs work in proportion to a slightly peculiar accounting of mutator work. The peculiarities reflect our focus on worst cases, which occur when the rate of promotion out of the nursery is nearly 100% and the mutator spends almost all of its time allocating storage and performing assignments.

The mutator's work is measured by the volume of storage that survives to be promoted out of the nursery and the number of assignments that go through the write barrier. If we ignore the nursery (which has little effect on the worst case) then promoted objects are, in effect, newly allocated within some region.

The collector's work is measured by the number of regions collected. A full cycle concludes when all nonempty, non-popular regions have been collected, so the number of regions collected also measures time relative to the current full cycle. That notion of time drives the scheduling of marking and summarization processes.

The marking and summarization processes are counted as overhead, not work. Our calculations assume their cost is evenly distributed (at the fairly coarse resolution of one major cycle) over the interval they are active, using mutator work as the measure of time. That makes sense for worst cases, and overstates the collector's relative overhead when the mutator does things besides allocation and assignments (because counting those other things as work would increase the mutator utilization).

3.6 Matching Collection Work to Allocation

At the beginning of a full cycle, the regional collector calculates the amount of storage the mutator will allocate (that is, promote into regions) during the full cycle.

Almost any policy that makes the mutator's work proportional to the collector's work would suffice for the proof of our main theorem, but the specific values of worst-case constants are sensitive to details of the policy. Furthermore, several different policies may have essentially the same worst-case performance but radically different overall performance on normal programs.

We are still experimenting with different policies. The policy stated below is overly conservative, but allows simple proofs of this section's lemmas because A is a monotonically increasing function of the peak live storage, and does not otherwise depend upon the current state of the collector.

Outside of this section, nothing depends upon the specific policy stated below. The proof of our main theorem relies only upon its properties as encapsulated by lemmas 9 and 10.

The following policy computes a hard lower bound for the amount of free space that will become available as regions are collected during this full cycle, and divides that free space equally between this full cycle and the next. If promoting that volume of storage might exceed the desired bound on heap size, then the promotion budget for this full cycle is reduced accordingly.

Policy 8. *The promotion to be performed during the coming full cycle is*

$$A = \min\left(\frac{1}{2}((1-k)L_{hard} - 1)P_{old}, (L_{soft} - 1)P_{old}\right)$$

where

- *k is any fixed upper bound for the fraction of nonempty regions that go uncollected within a full cycle. (Lemma 6 calculates a specific value for k.)*
- *P_{old} is the peak live storage, computed as the maximum value of N_{old} (see below).*
- *N_{old} is the volume of reachable storage at the beginning of the previous full cycle, as measured by the marking process during that cycle; if this is the first full cycle, then N_{old} is the size of the initial heap plus some headroom.*
- *L_{soft} is the desired ratio of N to peak live storage.*
- *$L_{hard} > 1/(1-k)$ is a fixed hard bound on the ratio of N to peak live storage at the beginning of a full cycle.*

The two lemmas below express the only properties that A must have.

Lemma 9. *If the collector parameters are consistent, then A is in $\Theta(P_{old})$.*

The following lemma states the regional collector's most critical invariant, and establishes that this invariant is preserved by every full cycle.

The critical insight of its proof is that the Cheney collection process reclaims all storage that was unreachable as of the beginning of the previous full cycle, except for the bounded fraction of objects that lie in uncollected regions. Furthermore there is no fragmentation among the survivors of collected regions, so the total storage in all regions at the end of a full cycle, excluding free space recovered by the cycle, is the sum of the total storage occupied by the survivors, the regions that aren't collected, and the storage that was promoted into regions during the cycle.

Lemma 10. *Let N_0 be the volume of storage in all regions, including live storage and garbage but not free space, at the beginning of a full cycle. Then $N_0 \le N \le L_{hard}P_{old}$.*

Proof. The lemma is true at the beginning of the first full cycle.

At the beginning of the second full cycle, N_0 consists of

- storage that was reachable at the beginning of the first full cycle (bounded by N_{old})
- storage in uncollected regions (bounded by kN)
- storage promoted into regions during the previous full cycle (bounded by A)

At the beginning of subsequent full cycles, N_0 consists of

- storage that was reachable at the beginning of the full cycle before the previous full cycle and is still reachable (bounded by N_{old})
- storage in uncollected regions (bounded by kN)
- storage promoted into regions during the previous full cycle (bounded by A)
- storage promoted into regions during the cycle before the previous full cycle (bounded by A, because A is nondecreasing)

Therefore

$$\begin{aligned} N_0 &\le N_{old} + kN + A + A \\ &= N_{old} + kN + ((1-k)L_{hard} - 1)P_{old} \\ &\le P_{old} + kL_{hard}P_{old} + ((1-k)L_{hard} - 1)P_{old} \\ &= L_{hard}P_{old} \end{aligned}$$

□

4. Worst-case Bounds

The subsections below sketch proofs for the three parts of our main theorem, which was stated in section 1.2.

We use asymptotic calculations because we cannot know the hardware- and software-dependent relative cost of basic operations such as allocations, write barriers, marking or tracing a word of memory, and so on. Constant factors are important, however, so we make a weak attempt to estimate some constants by assuming that all basic operations have the same cost per word. That is roughly true, but only for appropriate values of "roughly". The constant factors calculated for space may be more trustworthy than those calculated for time.

4.1 GC Pauses

It's easy to calculate an upper bound for the duration of major collections. The size of the region to be collected is a constant R. The size of its summary set is bounded by SR. The summary and mark-stack state to be updated is bounded by $O(R)$. A Cheney collection of the region therefore takes time $O(R + SR) = O(R)$.

4.2 Worst-case MMU

For any resolution Δt, the minimum mutator utilization is the infimum, over some set of intervals of length Δt, of the mutator's CPU time during that interval divided by Δt (Cheng and Blelloch 2001). The MMU is therefore a function from resolutions to the interval $[0, 1]$.

The obvious question is: What set of intervals are we talking about? In most cases, an MMU is defined over the intervals recorded during some specific execution of some specific benchmark on some specific machine. We'll call that an *observed MMU*.

Our main theorem uses a very different notion of MMU, which can be regarded as the infimum of observed MMUs over all possible executions of all possible benchmarks. We have been referring to that notion as the *theoretical worst-case MMU*.

The theoretical worst-case MMU is the notion that matters when we talk about worst-case guarantees or scalable algorithms.

The theoretical worst-case MMU is easily bounded above using observed MMUs; for example, an observed MMU of zero implies a theoretical worst-case MMU of zero. On the other hand, we cannot use observed MMUs to prove that a regional collector's theoretical worst-case MMU is bounded below by a non-zero constant. Our only hope is to prove something like our main theorem.

Some programs reach a storage equilibrium, which allows us to define the inverse load factor L as the ratio of heap size to reachable heap storage. Although some collectors can do better on some programs, it appears that, for any garbage collector, the theoretical worst-case ratio of allocation to marking is less than or equal to $L - 1$, from which it follows that there must be resolutions at which the worst-case MMU is less than or equal to

$$\frac{L-1}{(L-1)+1} = \frac{L-1}{L}$$

For a stop-and-collect collector, the worst-case MMU is zero for intervals shorter than the duration of the worst-case collection. For collectors that occasionally perform a full collection, taking time proportional to the reachable storage, the theoretical worst-case MMU is therefore zero at all resolutions. If there is some finite bound on the worst-case gc pause, however, then the theoretical worst-case MMU may be positive for sufficiently large resolutions.

Our main theorem claims this is true for a regional collector at resolutions greater than $3c_0$, where c_0 is a bound on the worst-case duration of a gc pause. At that resolution and above, the worst case occurs when two worst-case gc pauses surround a mutator interval in which the mutator performs a worst-case (small) amount of work. The two gc pauses take $O(R)$ time, so we need to show that the mutator will perform $\Omega(R)$ work between every two major collections.

The regional collector performs $\Theta(N/R)$ major collections per full cycle, and the scheduling of those collections is driven by mutator work. Between two successive major collections, the mutator performs $\Omega(AR/N)$ work, where A, the promotion per full cycle as defined in section 3.6, is in $\Theta(P_{old})$ and therefore in $\Omega(N)$.

If the regional collector had no overhead outside of major collections, the paragraph above would establish that the theoretical worst-case MMU at that resolution is bounded below by a constant. Since the regional collector does have overhead from the marking and summarization processes, we have yet to establish that (1) the overhead per major cycle of those processes is $O(R)$ and (2) their overhead is distributed fairly evenly within the interval; that is, there are no subintervals of duration $3c_0$ or longer that have an overly high concentration of overhead or overly low fraction of mutator work.

The marking process's overhead per full cycle is $O(N)$, and standard scheduling algorithms suffice to ensure that its overhead per major cycle is $O(R)$, with that overhead being quite evenly distributed when observed at the coarse resolution of $3c_0$.

The summarization process, as described in sections 2.3 and 3.3, is more complicated. The summarization process performs up to F_3 passes over the remembered set per summarization cycle. Each pass takes $O(N)$ time to scan the remembered set, while creating

$$O\left(\frac{1}{F_1}\frac{N}{R}SR\right)$$

entries in the summary sets. There are between F_1 and F_1/F_3 summarization cycles per full cycle, distributed as evenly as those tight constant bounds allow. In conclusion, the summarization process has $O(N)$ overhead per full cycle and $O(R)$ overhead per major cycle.

That would complete the proof of part 2, except for one nasty detail mentioned in section 2.3 and lemma 5: The mutator's work during summarization is limited to cN, where c is the constant defined in lemma 5.

That doesn't interfere with the proof of part 2, because the mutator is still performing $\Theta(N)$ work per summarization cycle, but it does lower mutator utilization. If we assume that all basic operations have about the same cost per word, then the theoretical worst-case MMU at sufficiently large resolutions is a constant of which we have some actual knowledge.

Lemma 11. *When regarded as a function of the collector's parameters, the regional collector's theoretical worst-case MMU is roughly proportional to*

$$\frac{SF_2F_3 - S - F_1F_2}{(S+1)(F_2F_3+2) + F_1F_2F_3}$$

Proof. The worst-case MMU is proportional to the worst-case mutator work accomplished during a major cycle, divided by the worst-case cost of the marking and summarization processes during a major cycle plus the worst-case cost of the two major collections that surround the mutator work. We assume that work and costs are spread evenly across the relevant cycles; any bounded degree of unevenness can be absorbed by the constant of proportionality.

The number of regions collected during a worst-case summarization cycle is

$$d = \frac{1}{F_1F_2}\frac{N}{R}$$

- The worst-case mutator work per major cycle is cN/d.
- The worst-case cost of summarization per major cycle is

$$F_3N + \frac{F_3}{F_1}\frac{N}{R}SR = (F_3 + \frac{F_3}{F_1}S)N$$

divided by d.

- The worst-case cost of the marking process during a major cycle is F_2F_3R, which is N divided by the worst-case number of major collections during a full cycle (as given by lemma 6).
- The worst-case cost of a major collection is $R + SR$.

The theoretical worst-case MMU is therefore roughly proportional to

$$\begin{aligned} & \frac{F_1F_2cR}{2(1+S)R + F_1F_2(F_3 + SF_3/F_1)R + F_2F_3R} \\ = \quad & \frac{SF_2F_3 - S - F_1F_2}{(S+1)(F_2F_3+2) + F_1F_2F_3} \end{aligned}$$

□

That calculation was pretty silly, but gives us quantitative insight into how much we can improve the theoretical worst-case MMU by choosing good values for the collector's parameters or by designing a more efficient summarization process.

4.3 Worst-case Space

The regional collector allocates a new region only when the current set of regions does not have enough free space to accomodate all of the objects that need to be promoted out of the nursery. Lemmas 9 and 10 therefore establish that N, the total storage occupied by all regions, is in $\Theta(P_{old})$ (where P_{old} is a lower bound for the peak live storage).

The remembered set is $O(N)$. The set of previously computed summary sets that have not yet been consumed by a major collection is $O(N)$. The set of summary sets currently under construction is $O(N)$. The mark bitmap is $O(N)$. Each mark stack (one per region) is $O(R)$, so the total size for all mark stacks is $O(N)$.

The total space required by the regional collector is therefore $\Theta(P_{old})$. The specific constants of proportionality depend upon collector parameters L_{hard}, S, F_1, and F_2 as well as details of the collector's data structures; for example, the size of the mark bitmap might be N, $N/2$, $N/4$, $N/8$, $N/32$, or $N/64$ depending on object alignment, granularity of marking, and number of bits per mark. With plausible assumptions about data structures, the theoretical worst-case space is about

$$\left(\left(\frac{5}{4} + \frac{SF_3}{F_1}\right)L_{hard} + \frac{1}{2}\right)P$$

where P is the peak reachable storage.

No program can reach theoretical worst-case bounds for all of the collector's data structures simultaneously. For example, the mark stack's worst case is achieved when the heap is filled by a single linked structure of objects with only two fields. That means half the pointers are perfectly distributed among regions, which halves the worst-case number of popular regions; it also removes the factor of L_{hard}, because all objects that get pushed onto the mark stack are reachable. On gc-intensive benchmarks, our prototype uses about the same amount of storage as stop-and-copy or generational collectors.

4.4 Floating Garbage

Floating garbage is storage that is reachable from the remembered set but is not reachable from mutator structures (and will not be marked by the next snapshot-at-the-beginning marking process).

In the calculations above, the peak reachable storage P does not include floating garbage, but the theoretical worst-case bounds do include floating garbage. In this section, we calculate a bound for how much of the worst-case space can be occupied by floating garbage.

When bounding the space used by collectors that never perform a full collection, the hard part is to find an upper bound for floating garbage. The regional collector is especially interesting because

- When a region is collected, its objects that were unreachable as of the beginning of the most recently completed marking cycle will be reclaimed.
- The regional collector does not guarantee that all unreachable objects will eventually be collected.
- The regional collector does guarantee that the total volume of unreachable objects is always bounded by a small constant times the total volume of reachable objects.

Suppose some object x, residing in some region r, becomes unreachable. If there are no references to x from outside r, then x will be reclaimed the next time r is collected.

If there are references to x from outside r, then those references will be removed from the remembered set at the end of the first marking cycle that begins after x becomes unreachable (because all references to an unreachable object are from unreachable objects). Then x will be reclaimed by the first collection of r that follows the completion of that marking cycle.

On the other hand, there is no guarantee that r will ever be collected. r will remain forever uncollected if and only if the summarization process deems r popular on every attempt to construct r's summary set.

Lemma 3 proves that the total volume of popular regions is no greater than N/S. Lemma 10 proves that $N \leq L_{hard}P$, where P is the peak live storage. Hence the total volume of perpetually uncollected garbage is no greater than L_{hard}/S times the peak live storage.

4.5 Collector Parameters

Most of the collector's parameters can be changed at the beginning of any full cycle. If the parameters change at the beginning of a full cycle, then it will take at most two more full cycles for the collector to perform within the theoretical worst-case bounds for the new parameters.

5. Near-Worst-Case Benchmarks

We have implemented a prototype of the regional collector, and will provide a more detailed report on its engineering and performance in some other paper. For this paper, we compare its performance to that of several other collectors on a very simple but extremely gc-intensive benchmark (Clinger 2009).

The benchmark repeatedly allocates a list of one million elements, and then stores the list into a circular buffer of size k. The number of popular objects (used as list elements) is a separate parameter p; with $p = 0$, the list elements are small integers, which are usually represented by non-pointers that the garbage collector does not have to trace.

To illustrate scalability and the effect of popular objects, we ran three versions of the benchmark:

- with $k = 10$ and $p = 0$
- with $k = 50$ and $p = 0$
- with $k = 50$ and $p = 50$

All three versions allocate exactly the same amount of storage, but the peak storage with $k = 10$ is about one fifth of the peak storage with $k = 50$. The third version, with popular objects, is the most challenging benchmark we have been able to devise for the regional collector. The queue-like object lifetimes of all three versions make them near-worst-case benchmarks for generational

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	192	170	.07	.60	386
Gambit	v4.4.3	stop&copy	63	44		.52	493
Ypsilon	0.9.6-update3	mostly concurrent	265	$\geq$ 53	.64	?	711
Sun JVM	1.5.0	generational	175	?		.78	333
Larceny	prototype	generational	109	88	.80	.88	555
Sun JVM	1.5.0	parallel	275	?		.91	511
Larceny	prototype	stop&copy	76	55	.90	.94	518
Chicken	4.0.0	Cheney-on-the-MTA	87	36		1.	490
PLT	v4.1.4	generational	227	211		1.	617
Ikarus	0.0.3	generational	264	242		2.25	1055
Sun JVM	1.5.0	incremental mark/sweep	409	?		3.41	530

Figure 2. GC-intensive performance with about 160 MB of live storage.

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	212	187	.11	.7	1808
Ypsilon	0.9.6-update3	mostly concurrent	24971	$\geq$ 24818	2.4	?	2067
Gambit	v4.4.3	stop&copy	68	47		2.5	2363
Chicken	4.0.0	Cheney-on-the-MTA	118	62		4.	1955
Sun JVM	1.5.0	parallel	311	?		4.2	1973
Larceny	prototype	generational	149	128	4.2	4.3	2073
Larceny	prototype	stop&copy	119	95	4.5	4.5	2058
Sun JVM	1.5.0	generational	212	?		4.9	1497
PLT	v4.1.4	generational	286	273		5.	2109
Ikarus	0.0.3	generational	419	371		11.6	2575
Sun JVM	1.5.0	incremental mark/sweep	457	?		15.8	2083

Figure 3. GC-intensive performance with about 800 MB of live storage.

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	618	592	.35	2.9	1865
Gambit	v4.4.3	stop&copy	72	51		2.7	2363
Ypsilon	0.9.6-update3	mostly concurrent	28366	$\geq$ 28212	2.89	?	1772
Sun JVM	1.5.0	parallel	314	?		4.1	1918
Larceny	prototype	generational	162	141	4.5	4.6	2064
Larceny	prototype	stop&copy	120	96	4.8	4.8	2060
Chicken	4.0.0	Cheney-on-the-MTA	127	69		5.	1955
Sun JVM	1.5.0	generational	216	?		5.0	1497
PLT	v4.1.4	generational	339	320		5.	2089
Ikarus	0.0.3	generational	427	409		10.7	2588
Sun JVM	1.5.0	incremental mark/sweep	479	?		18.1	2083

Figure 4. GC-intensive performance with 800 MB live storage and 50 popular objects.

collectors in general, and their simplicity and regularity make the results easy to interpret.

To eliminate pair-specific optimizations that might give Larceny (and some other systems) an unfair advantage, the lists are constructed from two-element vectors. Hence the representation of each list in Scheme is likely to resemble the representation used by Java and similar languages. In Larceny and in Sun's JVM, each element of the list occupies four 32-bit words (16 bytes), and each list occupies 16 megabytes.

The benchmarks allocate one thousand of those lists, which is enough for the timing to be dominated by the steady state but small enough for convenient benchmarking.

We benchmarked a prototype fork of Larceny with three different collectors. The regional collector was configured with a 1-megabyte nursery, 8-megabyte regions (R), a waveoff threshold of $S = 8$, and parameters $F_1 = 2$, $F_2 = 2$, and $F_3 = 1$; these parameters have worked well for a wide range of benchmarks, and were not optimized for the particular benchmarks reported here. To make the generational collector more comparable to the regional collector, it was benchmarked with a nursery size of 1 MB instead of the usual 4 MB.

For perspective, we benchmarked several other systems as well. We ran all benchmarks on a MacBook Pro equipped with a 2.4 GHz Intel Core 2 Duo (with two processor cores) and 4 GB of 667 MHz DDR2 SDRAM. Only three of the collectors made use of the second processor core: Ypsilon, Sun's JVM with the parallel collector, and Sun's JVM with the incremental mark/sweep collector. For those three systems, the total cpu time was greater than the elapsed times reported in this paper.

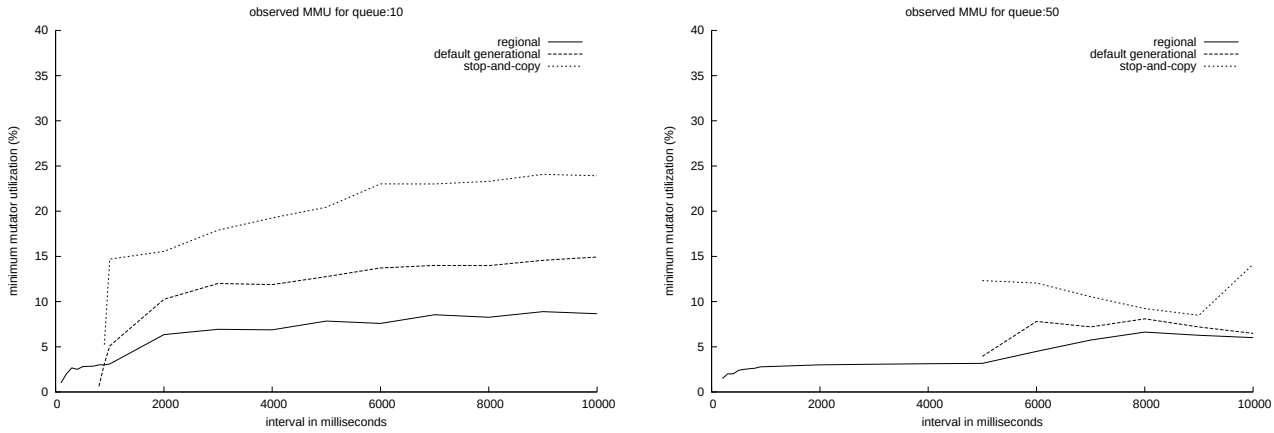

Figure 5. Observed MMU for $k = 10$ and $k = 50$.

Figures 2, 3, and 4 report the elapsed time (in seconds), the total gc time (in seconds), the duration of the longest pause to collect garbage (in seconds), the maximum variation (calculated by subtracting the average time to create a million-element list from the longest time to create one of those lists), and the maximum RSIZE (in megabytes) reported by `top`.

For most collectors, the maximum variation provides a good estimate of the longest pause for garbage collection. For the regional collector, however, most of the maximum variation is caused by uneven scheduling of the marking and summarization processes. With no popular objects, the regional collector's total gc time includes 51 to 54 seconds of marking and about 1 second of summarization. With 50 popular objects, the marking time increased to 104 seconds and the summarization time to 152 seconds. It should be possible to decrease the maximum variation of the regional collector by improving the efficiency of its marking and summarization processes and/or the regularity of their scheduling.

Figure 5 shows the MMU (minimum mutator utilization as a function of time resolution) for the three collectors implemented by our prototype fork of Larceny.

Although none of the other collectors were instrumented for MMU, their MMU would be zero at resolutions up to the longest gc pause, and their MMU at every resolution would be less than their average mutator utilization (which can be estimated by subtracting the total gc time from the elapsed time and dividing by the elapsed time).

As can be seen from figures 2 and 3, simple garbage collectors often have good worst-case performance. Gambit's non-generational stop&copy collector has the best throughput on this particular benchmark, followed by Larceny's stop&copy collector and Chicken's Cheney-on-the-MTA (which is a relatively simple generational collector).

Of the benchmarked collectors, Sun's incremental mark/sweep collector most resembles a soft real-time collector; it combines low throughput with inconsistent mutator utilization. Ypsilon performs poorly on the larger benchmarks, apparently because it needs more than 2067 megabytes of RAM, which is the largest heap it supports; Ypsilon's representation of a Scheme vector may also consume more space than in other systems.

The regional collector's throughput and gc pause times are degraded by popular objects, but its gc pause times remain the best of any collector tested, while using less memory than any system except for Sun's default generational collector.

The regional collector's scalability can be seen by comparing its pause times and MMU for $k = 10$ and $k = 50$. The maximum pause time increases only slightly, from .07 to .11 seconds. For all other systems whose pause times were measured with sub-second precision, the pause time increased by a factor of about 5 (because multiplying the peak live storage by 5 also multiplies the time for a full collection by 5). The regional collector's MMU is almost the same for $k = 10$ as for $k = 50$; for all other collectors, the MMU degrades substantially as the peak live storage increases.

6. Related Work

6.1 Generational garbage collection

Generational collection was introduced by (Lieberman and Hewitt 1983). A simplification of that design was first implemented by (Ungar 1984). Most modern generational collectors are modeled after Ungar's, but our regional collector's design is more similar to that of Lieberman and Hewitt.

6.2 Heap partitioning

Our regional collector is centered around the idea of partitioning the heap and collecting the parts independently. (Bishop 1977) allows single areas to be collected independently; his work targets Lisp machines and requires hardware support.

The *Garbage-First* collector of (Detlefs et al. 2004) inspired many aspects of our regional collector. Unlike the garbage-first collector, which uses a points-into remembered set representation with no size bound, we use a points-outof remembered set representation and points-into summaries which are bounded in size. The garbage-first collector does not have worst-case bounds on space usage, pause times, or MMU. According to Sun, the garbage-first collector's gc pause times are "sometimes better and sometimes worse than" the incremental mark/sweep collector's (Sun Microsystems 2009).

The *Mature Object Space* (a.k.a. *Train*) algorithm of (Hudson and Moss 1992) uses a fixed policy for choosing which regions to collect. To ensure completeness, their policy 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 *Beltway* collector of (Blackburn et al. 2002) uses heap partitioning and clever infrastructure to enable flexible selection of collection policies via command line options. Their policy selection is expressive enough to emulate the behavior of semi-space, genera-

tional, renewal-older-first, and deferred-older-first collectors. They demonstrate that having a more flexible policy parameterization can introduce improvements of 5%, 10%, and up to 35% over a fixed generational collection policy. Unfortunately, in the Beltway system one must choose between incremental or complete collection. The Beltway collector does not provide worst-case guarantees independent of mutator behavior.

The *MarkCopy* collector of (Sachindran and Moss 2003) breaks the heap down into fixed sized *windows*. 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 of (Ben-Yitzhak et al. 2002) also has similarities to our approach. They select an area of the heap to collect, and then concurrently build a summary for that area. However, they construct their points-into set by tracing the whole heap, rather than maintaining points-outof remembered sets. 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.

6.3 Older-first garbage collection

Our design employs a round-robin policy for selecting the region to collect next, focusing the collector on regions that have been left alone the longest. Thus our regional collector, like older-first collectors (Stefanović et al. 2002; Hansen and Clinger 2002), tends to give objects more time to die before attempting to collect them.

6.4 Bounding collection pauses

There is a broad body of research on bounding the pause times introduced by garbage collection, including (Baker 1978; Brooks 1984; Appel et al. 1988; Yuasa 1990; Boehm et al. 1991; Baker 1992; Nettles and O'Toole 1993; Henriksson 1998; Larose and Feeley 1998). In particular, (Blelloch and Cheng 1999) provides proven bounds on pause-times and space-usage.

Several attempts to bring the pause-times down to precisions suitable for real-time applications run afoul of the problem that bounding an individual pause is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high. (Cheng and Blelloch 2001) introduces the MMU metric to address this issue. That paper presents an *observed* MMU for a parallel real-time collector, not a theoretical worst-case MMU.

6.5 Collection scheduling

Metronome (Bacon et al. 2003b) is a hard real-time collector. 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%. More significantly, Metronome's guaranteed bounds on utilization and space usage depend upon the accuracy of application-specific parameters; (Bacon et al. 2003a) extends this set of parameters to provide tighter bounds on collection time and space overhead.

Similarly, (Robertz and Henriksson 2003) depends on a supplied schedule to provide real-time collector performance. 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 provides worst-case guarantees independent of mutator behavior, but cannot provide millisecond-resolution guarantees. Our regional collector is mostly copying, has no read barrier, and uses work-based accounting to drive the collection policy.

6.6 Incremental and concurrent collection

There are many treatments of concurrent collectors dating back to (Dijkstra et al. 1978). 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 (Detlefs et al. 2002).

The summarization and marking processes require a write barrier, which we piggy-back onto the barrier in place to support generational collection. This is similar to how (Printezis and Detlefs 2000), building on the work of (Boehm et al. 1991), merges the overhead of maintaining concurrency related invariants with the overhead of maintaining generational invariants.

7. Future Work

Our current prototype interleaves the marking and summarization processes with the mutator, scheduling at the granularity of minor cycles and the processing of write barrier logs. Both the marking and summarization processes could be concurrent with the mutator, which would improve throughput on programs that do not fully utilize all processor cores. The marking process was actually implemented as a concurrent thread by one of our earlier prototypes, but the current single-threaded prototype makes it easier to measure every process's effect on throughput.

The collections performed by the regional collector can themselves be parallelized, but that is essentially independent of the design.

We assume that object sizes are bounded, so every object will fit into a region. Because we have implemented our prototype in Larceny, we can change both the compiler and the run-time representations of objects, choosing representations that break extremely large objects into pieces of bounded size.

The regional collector's nursery provides most of the benefits associated with generational garbage collection. Although the regional collector sacrifices some throughput on extremely gc-intensive programs, its performance on more normal programs can and does approach that of contemporary generational collectors. We will offer a more complete report on our prototype's observed performance in a separate paper.

8. Conclusions

We have described and prototyped a regional collector, which is a new kind of generational garbage collector.

We have proved that the regional collector is scalable: It guarantees worst-case bounds for gc latency, minimum mutator utilization, and space usage, independent of the peak live storage and mutator behavior.

Such guarantees remain rare. Although our proof is not the first of its kind, it may be the first to guarantee worst-case bounds for MMU as well as latency and space.[5]

The regional collector incorporates novel and elegant solutions to the problems presented by popular objects and floating garbage.

[5] For example, Cheng and Blelloch proved that a certain hard real-time collector has nontrivial worst-case bounds for both gc latency and space, but they had not yet invented the concept of MMU (Blelloch and Cheng 1999).

We have prototyped the regional collector, using a near-worst-case benchmark to illustrate its performance.

References

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

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

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 *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, pages 81–92, San Diego, CA, June 2003a. ACM Press.

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 2003b. ACM Press.

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.

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

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.

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.

Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press. ISBN 1-58113-463-0.

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.

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

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.

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

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.

William D. Clinger. Queue benchmark for estimating worst-case gc pause times. Website, 2009. `http://www.ccs.neu.edu/home/will/Research/SW2009/`.

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.

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.

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.

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.

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.

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

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.

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.

Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 1–9, Vancouver, October 1998. ACM Press. ISBN 1-58113-114-3.

Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983. ISSN 0001-0782.

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.

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. ISBN 1-58113-263-8.

Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, pages 93–102, San Diego, CA, June 2003. ACM Press.

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.

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.

Sun Microsystems. Java HotSpot garbage collection. Website, 2009. `http://java.sun.com/javase/technologies/hotspot/gc/g1_intro.jsp`.

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.

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