Reducing Pause Times With Clustered Collection

by

Cody Cutler

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2014

Signature redacted

Certified by .....
Robert Morris
Associate Professor
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Reducing Pause Times With Clustered Collection

by

Cody Cutler

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Clustered Collection reduces garbage collection pauses in programs with large amounts of live data. A full collection of millions of live objects can pause the program for multiple seconds. Much of this work, however, is repeated from one collection to the next, particularly for programs that modify only a small fraction of their object graphs between collections.

Clustered Collection reduces redundant work by identifying regions of the object graph which, once traced, need not be traced by subsequent collections. Each of these regions, or "clusters," consists of objects reachable from a single head object. If the collector can reach a cluster's head object, it skips over the cluster, and resumes tracing at the pointers that leave the cluster. If a cluster's head object is not reachable, or an object within a cluster has been written, the cluster collector may have to trace within the cluster. Clustered Collection is complete despite not tracing within clusters: it frees all unreachable objects.

Clustered Collection is implemented as modifications to the Racket collector. Measurements of the code and data from the Hacker News web site show that Clustered Collection decreases full collection pause times by a factor of three. Hacker News works well with Clustered Collection because it keeps gigabytes of data in memory but modifies only a small fraction of that data. Other experiments demonstrate the ability of Clustered Collection to tolerate certain kinds of writes, and quantify the cost of finding clusters.

Thesis Supervisor: Robert Morris
Title: Associate Professor

Acknowledgments

Thank you Robert and labmates, both present and past, for teaching me so much.

Contents

List of Figures

Chapter 1

Introduction

A major cost in tracing garbage collectors is the need to eventually examine every live object in order to follow its child pointers. If there are millions of live objects this tracing can easily take a few seconds. Stop-the-world collectors expose this cost directly in the form of pause times, which can be awkward for servers and interactive programs. Many techniques have been developed to reduce or mask tracing cost, such as parallel, concurrent, and generational collection [9, 6, 2, 4, 10, 12]. However, all of these techniques would benefit if they had to trace fewer objects.

The work in this thesis exploits the similarity of the garbage collection computation from one run to the next, particularly for large programs that modify only a fraction of their data. The basic idea is that successive full collections can omit tracing within regions of the object graph that the program does not modify. A design based on this idea must answer some key questions. How to constrain regions so that they can be skipped without sacrificing safety or completeness? How to find regions that are unlikely to be modified by program writes? How to react to any writes that do occur? How to find regions whose pointers mostly stay within the region?

Clustered Collection addresses these problems as follows. A periodic Cluster Analysis identifies non-overlapping clusters, each consisting of a head object along with other objects reachable from the head. Cluster Analysis records, for each cluster, the locations of "out" pointers that leave that cluster. During a full collection, if a cluster's head object is reachable and the cluster's objects have not suffered certain

kinds of writes, the entire cluster is live and need not be traced. In that case tracing resumes at the cluster's "out" pointers (thus maintaining safety). A program write in a cluster that could cause one of the cluster's objects to be unreferenced causes the cluster to be dissolved: full collections until the next Cluster Analysis must trace the ex-cluster's objects (thus maintaining completeness). Other kinds of program writes within clusters can either be ignored or handled by modifying the cluster's out-pointer list.

Much of the design of Clustered Collection focuses on choosing clusters likely to be helpful in reducing full collection pause times. One major consideration is that clusters should be unlikely to suffer program writes that force them to be dissolved. Cluster Analysis omits recently-written objects from any cluster, since such objects may be likely to suffer more writes in the near future. Cluster Analysis adaptively limits the maximum cluster size to reduce the probability that any one cluster suffers a write. Clustered Collection also avoids dissolving a cluster despite many kinds of program writes, when it can establish that the write cannot change any object's liveness.

The other main consideration is that each cluster should have relatively few "out" pointers, to reduce the time that a full collection must spend tracing them. Cluster Analysis tries to form clusters large enough that each holds many "natural clusters" (e.g. sub-trees); this helps reduce out-pointers by reducing the number of natural clusters sliced by each cluster's boundary. Cluster Analysis also detects commonly referenced and effectively immortal objects (e.g. type objects) and avoids creating out-pointers to them.

This work presents an implementation of Clustered Collection as a modification to Racket's precise single-threaded generational collector [11, 5]. The main challenge in this implementation is Racket's page-granularity write barriers. The implementation achieves finer granularity by creating a shadow copy of the old content of each written page in order to observe the specific writes to clustered objects that a program makes.

The Evaluation shows the performance improvement for two programs that are well suited to Clustered Collection: a news aggregator web service with hundreds of

thousands of articles, and a k-nearest-neighbor classification program with 1.25 GB of training data. For the former, Clustered Collection reduces full-collection pause times by an average of 3×. For the latter, somewhat less than 2×. Clustered Collection imposes two costs in return for shorter full collection pause times: total program throughput is 10% and 3% slower, respectively, and the initial Cluster Analysis takes a few times longer than a full collection. The evaluation explores the dependence of performance on various aspects of program behavior, as well as the overheads that Clustered Collection imposes on the program. Clustered Collection is most effective for programs that have large numbers of live objects, and that have locality in their writes – that concentrate their writes in particular parts of the object graph so that large regions are unmodified from one collection to the next.

To summarize, this work's novel contributions include: 1) the idea that full collections can profitably avoid re-tracing regions of the object graph that have not been modified; 2) heuristics for discovering useful instances of such regions; and 3) an evaluation exploring the conditions under which the technique is most beneficial.

Chapter 2

Related Work

Hayes [7] observes that related objects often have the same lifetimes, and in particular die at about the same time. Hayes suggests a "key object opportunism" garbage collection technique, in which a "key" object acts as a proxy for an entire cluster; the collector would only trace the cluster if the key object were no longer reachable. Hayes' work contains the basic insight on which Clustered Collection is built, but is not a complete enough design that its properties can be compared with those of Clustered Collection.

Generational garbage collection [9] is related to Clustered Collection in the sense that, if objects in the old generation don't change much, a generational collector will not trace them very often. However, once a generational collector decides to collect the old generation, it traces every live object; thus a generational garbage collector reduces total time spent in collection but not the pause time for individual full collections. Clustered Collection is compatible with generational collection, and can reduce their full collection pause times. Clustered Collection also borrows the write barrier technique from generational collectors, though it uses barriers to track connectivity changes within and among clusters rather than between the old and new generations.

Generalizations of generational collection [8, 4] partition the heap and are able to collect just one partition at a time, reducing worst-case pause times. The G1 collector [4] keeps track of inter-partition pointers using write barriers, a technique

which Clustered Collection borrows. The main new ideas in Clustered Collection have to do with actively finding parts of the object graph that never need to be traced (modulo program writes); G1 does not do this.

Parallel [6] and concurrent/incremental [2, 12, 10] garbage collectors reduce pause times for full collections by running the collection on multiple cores or by performing some of the collection while the program runs. Clustered Collection seems likely to be helpful if added to a parallel collector.

Cohen [3] reduces collection synchronization in a parallel run-time by associating sub-heaps of data with clusters of threads that access that data; different sub-heaps can be collected without disturbing program threads using other sub-heaps.

Real-time collectors [1, 10] ensure that garbage collection pauses do not prevent the application from meeting deadlines by bounding the duration of collection pauses. The goal of our work differs with real-time collectors in that Clustered Collection aims to reduce total full collection pause times while real-time collectors suffer the pause time of a full collection in separate, consistent pauses.

The Mapping Collector [13] exploits the similarity in lifetimes of groups of objects that are allocated at the same time; instead of copying objects to avoid fragmentation, it waits for whole pages of objects to become dead, so that entire pages can be recycled. Avoiding copying allows the Mapping Collector to reduce pause times. Clustered Collection exploits related properties of objects; it does have to copy clustered objects, but then is able to avoid further copies of tracing in those objects.

Chapter 3

Design

3.1 Overview

Clustered Collection is an extension to pointer-tracing collector designs. It has three parts. Cluster Analysis runs periodically to decide which parts of the object graph should form clusters. The Watcher uses write barriers to recognize when the program modifies an object in a cluster. The Tracer modifies the way a full collection's "mark" phase traces pointers, causing it to skip over unmodified clusters.

Suppose a program's live object graph looks like part A of Figure 3-1. When Cluster Analysis examines the object graph, it might choose the two clusters shown in part B. Each cluster has a "head" object (shown with a green dot), from which all other objects in the cluster must be reachable; during a collection, reachability of the head implies that all of the cluster's objects are live. Pointers may also enter a cluster to non-head objects, but only an external reference to the head will cause the Tracer to skip tracing the whole cluster. For each cluster, Cluster Analysis records the set of objects in the cluster that contain "out" pointers referring to objects outside the cluster (there is just one in Part B).

While the program executes, the Watcher uses write barriers to detect program modifications of cluster objects. Part C shows three modifications with red dots: the program has added a new child to the root object, has changed a pointer in the right-hand cluster to point to a different object in the cluster, and has added a pointer

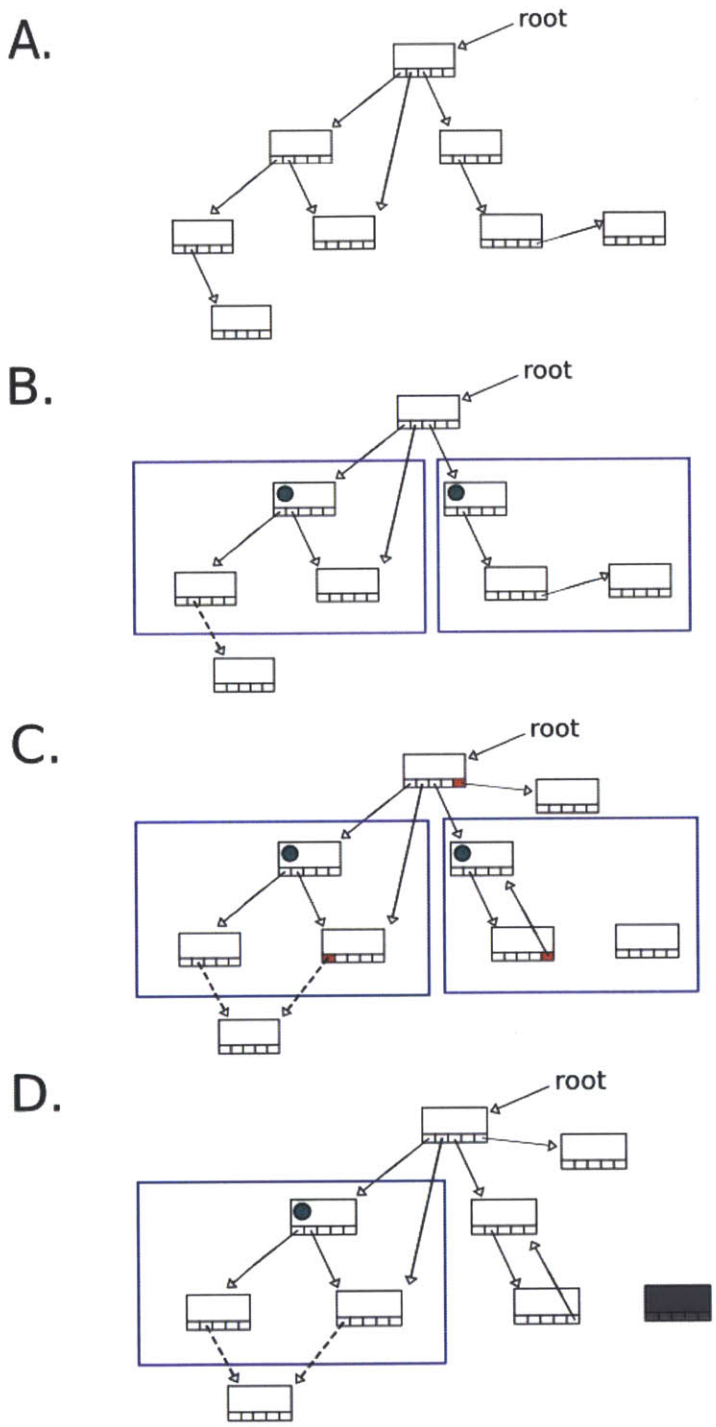

Figure 3-1: An example clustering. Nodes with green circles are cluster head objects, arrows are pointers, dashed arrows are "out" pointers, and blue boxes enclose clusters. Part A shows an application's live data, part B shows a possible choice of clusters, part C depicts the live data after the program has modified some data (changed pointers in red), and part D shows Clustered Collection's response (dissolving the right-hand cluster).

to an outside object to the left-hand cluster. The Watcher responds to the change in the right-hand cluster by dissolving the cluster, since a change to an intra-cluster pointer may cause an object in the cluster to be unreachable (as has happened here); completeness requires that full collections no longer skip that cluster. The Watcher also marks the written object so that later Cluster Analyses will omit it from any cluster.

The Watcher can tell that the program's write to the left-hand cluster in part C could not have changed the liveness of any object within the cluster (by comparing the old pointer with the newly written pointer), so it does not dissolve the cluster. Because the new pointer points outside the cluster, it may affect liveness of outside objects, so the Watcher adds an entry to the cluster's out-pointer set.

The Tracer is part of the full garbage collector's "mark" phase. Ordinarily, a mark phase follows ("traces") all pointers from a set of roots to discover all live objects; the mark phase sets a "mark" bit in each discovered object's header to indicate that it is alive. The Tracer modifies this behavior. At the start of a collection, the Tracer forgets about clusters that the Watcher dissolved, leaving the situation in Part D. If the mark phase encounters the "head" object of one of the remaining clusters, the Tracer marks the entire cluster as live, and causes the mark phase to continue by tracing the cluster's "out" pointers. If the Tracer encounters an object inside a cluster before encountering that cluster's head object, it postpones the object in order to increase the chance of the cluster's head being traced during the remainder of the mark phase, enabling the Tracer to skip the postponed object when it is later re-examined. If the cluster containing the postponed object is still unmarked when the postponed object is re-examined, it will be traced in the ordinary way.

In the example in Figure 3-1, it is good that Cluster Analysis created two clusters instead of one; that allowed the left-hand cluster to survive despite the writes in Part C.

A cluster is similar to an object: one can view it as a single node in the object graph, with pointers to it and pointers out of it. The cost to trace a cluster is little more than the cost of tracing a single object; thus the more objects Clustered

Collection can hide inside clusters, the more it can decrease collection pauses.

3.2 Clusters

The most important aspect of the Clustered Collection design is the way it chooses clusters. This chapter explains the properties that clusters must have in order to be correct and useful.

In order that Clustered Collection be safe (never free reachable objects) and complete (free all unreachable objects), the collector includes these design elements:

- Head objects: In order to be complete despite not tracing inside clusters, there must be a way to decide whether all of a cluster's objects are reachable. Clustered Collection does this by choosing clusters that have a head object from which all other objects in the cluster are reachable; if the head object is reachable from outside the cluster, every object in the cluster is live.

- Out pointers: In order to be safe, the collector must trace all pointers that leave each cluster. Clustered Collection does this by recording the set of cluster objects that contain "out" pointers, which it traces during collections.

- Write barriers: In order to maintain the invariants that all of a cluster's objects are reachable from the head, and that the "out" set contains all external pointers, Clustered Collection must be aware of all program writes to cluster objects. Some writes may force the cluster to be dissolved, or may require additions to the cluster's "out" set.

Among the possible correct clusters, some choices lead to greater reduction in full collection time than others:

- Objects written in the recent past may be likely to be written in the near future, forcing the containing cluster to be dissolved. Thus it is good to omit recently-written objects from any cluster.

- Large clusters are good because they can reduce the number of "out" pointers that each full collection must trace. For example, if the head object is an interior node of a tree, it's good to allow the cluster to be large enough to encompass the entire sub-tree below the head, since then all the tree links will point within the cluster. A cluster smaller than the entire sub-tree will have to have "out" pointers leading to sub-sub-trees.

- On the other hand, a large cluster is likely to have a higher probability of receiving at least one write than a small cluster. Since even a single write may force dissolution of the entire cluster, there is an advantage to limiting the size of clusters.

- Very small clusters, and clusters with a high ratio of "out" pointers to objects, may have costs that exceed any savings.

3.3 State

Clustered Collection maintains these data structures:

- `o.cluster`: for each object, the number of the cluster it belongs to, if any.

- `o.head`: for each object, a flag indicating whether the object is the head of its cluster.

- `o.written`: for each object, a flag indicating that the program has written the object since the last Cluster Analysis execution.

- `c.mark`: for each cluster, a flag indicating whether the cluster's head has been reached during the current full collection.

- `c.out`: for each cluster, its "out" set: the set of objects within the cluster that contain "out" pointers.

3.4 Cluster Analysis

The job of Cluster Analysis is to form clusters in accordance with the considerations explained in Section 3.2.

Cluster Analysis gets a chance to run before each full collection. It runs if two conditions are met: 1) the program size has stabilized, and 2) the ratio of unclustered objects to clustered objects is above a threshold `run_thresh`. The first condition suppresses Cluster Analysis during program initialization; it is true as soon as a full collection sees that that the amount of live data has increased by only a small fraction since the last full collection. For the second condition, the ratio of unclustered objects to clustered objects is low immediately after Cluster Analysis runs, then grows as program writes cause clusters to be dissolved, until it reaches `run_thresh` and Cluster Analysis runs again. Cluster Analysis leaves existing clusters alone, and forms new clusters out of unclustered objects.

Cluster Analysis pseudo-code is shown in Figure 3-2. At any given time, Cluster Analysis has a work stack of pairs of objects and cluster identifiers. It processes a work stack item o/c as follows. If Cluster Analysis has already visited object o, it ignores it. If o has been written since the last Cluster Analysis execution (`o.written` is set), or is a "sink" object (see below), Cluster Analysis adds o to no cluster, and pushes o's children on the work stack, each with a null c. o's children are likely to become cluster heads. Otherwise, if c is not null, and adding o to cluster c would not cause c to contain more than `max_size_thresh` objects, Cluster Analysis adds o to c and pushes o's children with c. Otherwise Cluster Analysis creates a new cluster c' with o as head, and pushes o's children with c'.

Cluster Analysis then moves each cluster's objects to a separate region of memory, so that the cluster's objects are not intermingled with any other objects. This has several benefits. First, until the entire cluster is freed or dissolved, the cluster's objects will not need to be moved since they are not fragmented. Second, to the extent that frequently-written objects are successfully omitted from clusters, cluster memory will be less likely to suffer Racket write-barrier page faults and shadow page

```
function CLUSTER_ANALYSIS:
    if heap size is not yet stable then
        return
    if #unclustered/#clustered < run_thresh then
        return
    if n_w > n_o then
        max_size_thresh /= 2
    else
        max_size_thresh *= 2
    for r in roots do
        CLUSTER1(r, nil)
    clear all o.written
    move objects to per-cluster separate memory
    fix up pointers to clustered objects
    for c in clusters do
        calculate c.out
    for c in clusters do
        np = NUMOUTPOINTERS(c)
        if np / SIZE(c) > out_thresh then
            DELETE(c)
        else if SIZE(c) < min_size_thresh then
            DELETE(c)
function CLUSTER1(object o, cluster c):
    if LIVE_CLUSTER(o.cluster) then
        if o.cluster.mark then
            return
        o.cluster.mark = True
        for o_1 in o.cluster.out do
            CLUSTER1(o_1, nil)
        return
    if o.mark then
        return
    o.mark = True
    sizeok = SIZE(c)+1 < max_size_thresh
    if o.written or IS_SINK(o) then
        c = nil
    else if c ≠ nil and sizeok then
        o.cluster = c
    else
        c = new cluster
        o.cluster = c
        o.head = True
    for o_1 in CHILDREN(o) do
        CLUSTER1(o_1, c)
```

Figure 3-2: Cluster Analysis pseudo-code. The implementation uses a work stack rather than recursive calls.

copying. Third, the cluster's "out" set can be compactly represented by a bitmap with a bit for each word in the cluster's memory area. Finally, since collections don't move a cluster's objects, intra-cluster pointers need not be fixed up during garbage collection.

Cluster Analysis then builds the "out" set for each cluster by scanning the cluster's objects. It places an object in the out set if it contains a pointer to a non-sink object that is outside the cluster.

Finally, Cluster Analysis looks for clusters whose out-pointer-to-object ratios are greater than `out_thresh`, or whose size is less than `min_size_thresh`, and destroys them. Such clusters do not save enough collector work to be worth the book-keeping overhead. More importantly, destroying these clusters feeds back into the adaptive maximum size computation; see Section 3.5 below.

3.5 Cluster Size Threshold

Cluster Analysis adjusts `max_size_thresh` (the target cluster size) adaptively. Clusters should be large enough that many of a cluster's objects' pointers point within the cluster, in order to reduce the number of out-pointers. Clusters should also be small enough that many clusters will not suffer any program writes, and thus won't have to be dissolved.

`max_size_thresh` is initially one sixteenth of the number of objects at the time Cluster Analysis first executes. On each subsequent execution, Cluster Analysis calculates the number of objects in clusters that were dissolved due to writes since the last execution (n_w), and the number of objects in clusters that the previous execution dissolved because they had too many out-pointers (n_o). If n_w is greater than n_o, `max_size_thresh` is halved. Otherwise it is doubled.

This algorithm causes `max_size_thresh` to oscillate. This oscillation causes no problems as long as the threshold is considerably larger than the "natural" size of the program's clusters.

3.6 Sink Objects

Some objects are so pervasively referenced that they would greatly inflate cluster out-pointer sets if not handled specially; type-descriptor objects are an example. Cluster Analysis detects long-lived objects with large numbers of references, declaring them "sink" objects. It omits them from cluster out-pointer sets, moves them to "immortal" regions of memory, does not include them in any cluster, and arranges for them not to be moved by subsequent collections. "Sink" objects are detected while building the "out" sets; the objects most referenced by a cluster's "out" set will become "sink" objects if the number of references exceeds a threshold, `sink_thresh`.

3.7 Watcher

Clustered Collection needs to know about program writes for three reasons. First, a write to one of a cluster's objects may cause violation of the invariant that the cluster's "out" set contains all pointers that leave the cluster. Second, a write to one of a cluster's objects may cause violation of the invariant that all of the cluster's objects are reachable from the head object. Third, Cluster Analysis should not include objects written in the recent past in any cluster. Clustered Collection can use the write barriers that are usually required for a generational garbage collector (but object granularity precision is needed; see Section 4).

For each object the program writes, the Watcher decides whether the write forces dissolution of the object's enclosing cluster. Suppose the program modifies o so that a slot that used to point to o_o now points to o_n. If o_o is inside the same cluster as o, then the modification forces dissolution because o_o may now be unreachable; the Watcher sets o's `o.written` and marks the cluster as dissolved. If o_o is a "sink" object or an immediate value such as a small integer, and o_n is outside the cluster, the Watcher adds o to the cluster's "out" set. Otherwise, if none of o's slots reference objects outside o's cluster, o is removed from the cluster's "out" set.

The above strategy avoids cluster dissolution for many program writes. For exam-

ple, suppose both o_o and o_n are outside the cluster. Changing o to point to o_n rather than o_o does not affect liveness of objects inside the cluster, though it may affect liveness of o_o or o_n. Because the collector will trace all of the cluster's out-pointers, and because the out-pointer set contains the locations of the out-pointers (rather than their values), the collector will notice if the modification changed the liveness of either o_o or o_n. Thus the Watcher can safely ignore the write in this example.

3.8 Tracer

Clustered Collection requires modifications to the underlying garbage collector, as follows.

Before the mark phase, the collector discards information about clusters dissolved due to writes since the last collection. These clusters' objects are then treated as ordinary (non-clustered) objects.

During the collector's mark phase:

- If the mark phase encounters cluster c's head object, and `c.mark` is not set, the mark phase sets `c.mark` and traces c's "out" pointers.
- If the mark phase encounters a non-head object o in cluster c, and `c.mark` is set, the mark phase ignores o. If `c.mark` is not set and o's mark is not set, the mark phase postpones the marking of o and its children.
- If the mark phase encounters an unmarked object that is not in a cluster, it proceeds in the ordinary way (by setting its mark bit and tracing its children).
- The mark phase processes the postponed objects once only postponed objects remain. For each postponed object o in cluster c, if `c.mark` is set, the mark phase ignores o. Otherwise the mark phase sets o's mark and traces its child pointers.

The purpose of postponing non-head objects is to avoid inter-cluster tracing. If the head of an object's cluster is traced after an object has been postponed but before

that postponed object is re-inspected, the postponed object can be ignored.

After the mark phase completes, all non-live clusters are dissolved. All objects in each live cluster are live. A non-cluster object (including any object in a dissolved cluster) is live if its mark bit is set.

The collector's copy or compaction phase does not move objects in live clusters.

The collector must "fix up" pointers to any objects it moves. Typically the fix-up phase considers every child pointer of every object. Since a collection doesn't move objects that are in clusters, the fix-up phase only needs to consider pointers in objects in a cluster's "out" set; other pointers in cluster objects don't need fixing, since they point either to objects within the same cluster, or to "sink" objects that never move.

3.9 Discussion

Cluster Analysis uses depth-first search to build clusters, with `max_size_thresh` limiting the size of each cluster. This strategy works well for lists and for tree-shaped data: it yields clusters with a high object-to-out-pointer ratio. For a list of small items, the effect is to segment the list into clusters; each cluster has just one out-pointer (to the head of the next segment). For a tree, `max_size_thresh` is expected to be much larger than the tree depth, so that each cluster encompasses a sub-tree with many leaves; thus there will be considerably fewer out-pointers than objects. For tables or lists where the individual elements contain many objects, Cluster Analysis will do well as long as `max_size_thresh` is larger than the typical element size.

Some object graphs may contain large amounts of unchanging data, but have topologies that prevent that data from being formed into large clusters. For example, consider a large array that is occasionally updated, and whose elements are small and read-only. The only way to form a cluster with more than one element is to include the array object in the cluster, probably as the head. However, the program's writes to the array object may quickly force the cluster to be dissolved. If such situations are common, Cluster Collection must treat them specially by splitting up the large object; Section 4 describes this for Racket's hash tables.

Cluster Analysis' adaptive choice of `max_size_thresh` responds to both program writes and graph structure. If too many clusters are destroyed by writes, Cluster Analysis will reduce `max_size_thresh`, so that each write dissolves a cluster containing fewer objects. If `max_size_thresh` shrinks too much, many clusters will have out-pointer-to-object ratios exceeding `out_thresh`, so that Cluster Analysis will itself destroy them; this will prompt the next execution of Cluster Analysis to increase `max_size_thresh`.

It is possible for there to be no good equilibrium size threshold: consider a program whose data is a randomly connected graph, and that continuously adds and deletes pointers between randomly selected objects. The program will modify a relatively high fraction of unpredictable objects between collections, which means that clusters must be small in order to escape dissolution. On the other hand, the object graph is unlikely to contain "natural" clusters of small size with mostly internal pointers. Clustered Collection won't perform well for this program; it is targeted at programs which leave large portions of the object graph unmodified, and which exhibit a degree of natural clustering.

The Cluster Analysis strategy of omitting recently written objects is a prediction that writes in the near future will affect the same objects that were written in the recent past. If that prediction is largely accurate, Cluster Analysis will eventually form clusters that aren't written, and thus aren't dissolved, and that therefore save time during full collections. If the prediction isn't accurate, perhaps because the program has little write locality, many clusters will be dissolved and thus won't be skippable during full collections.

Some performance could be gained by sacrificing or deferring completeness. For example, the Watcher could temporarily ignore writes that change one internal pointer to another, which is safe but might delay freeing of the object referred to by the old pointer.

Chapter 4

Implementation

We implemented Clustered Collection as a modification to the precise collector in Racket [11, 5] version v5.90.0.9. The Racket collector is a single-threaded generational copying collector. It detects modifications to objects in the old generation by write-protecting virtual memory pages.

Clustered Collection uses 24 bits in each object's header; these bits are taken from the 43 bits holding each object's hash value. 20 of the bits hold the object's cluster number, one bit holds the "head" flag, one holds the "written" flag, and one holds a "sink" flag. In all our experiments, this reduction in hash bits had no noticeable effect.

The Watcher needs to discover which cluster (if any) owns the page a write fault occurs on (it cannot easily tell from the faulting pointer where the containing object starts). It does this with a table mapping address ranges to cluster numbers; this table is implemented as an extension of the Racket collector's "page table."

In order to know exactly which objects the program has modified, Clustered Collection needs object-granularity write barriers. Racket only detects which pages have been written. Clustered Collection copies each page to a "shadow copy" on the page's first write fault after each collection. During the next full collection, each page and its shadow copy are compared to find which objects were modified; each written object may have its `o.written` flag set, and may dissolve the containing cluster. The implementation maintains `o.written` for all objects, not just objects in clusters.

Phase name	Cluster analysis + GC	Stock
Cluster ID assignment	Yes	No
Mark+Copy	Yes	Yes
Out pointer discovery	Yes	No
Pointer fix-up	Yes	Yes

Figure 4-1: The passes over the live data made by Cluster Analysis, compared with the full collection passes made by Racket's stock collector.

Racket implements hash tables as single vectors, so that a hash table with millions of entries is implemented as a very large object. If not treated specially, such an object, if written, might not be eligible for clustering; this in turn would likely mean that each item in the hash table would have to be placed in its own small cluster. To avoid this problem, Clustered Collection splits large hash tables, so that each "split" and its contents can form a separate cluster. This allows reasonably large clusters, while also causing a program write to dissolve only the cluster of the relevant split.

The Cluster Analysis implementation makes four passes over the objects, as shown in Figure 4-1 (marking and copying are interleaved). Two of these passes are shared with an associated full collection. Combining Cluster Analysis with a full collection saves work since both need to move live data, and thus both need to fix up pointers. A more sophisticated implementation could assign cluster IDs during the Mark+Copy phase, saving one pass; similarly, out-pointer discovery could be combined with pointer fix-up.

Chapter 5

Evaluation

This chapter measures how much Clustered Collection reduces full collection pause times, how much time Cluster Analysis takes, how much other overhead Clustered Collection imposes, and how sensitive it is to program writes.

The experiments run on a 3.47 GHz Intel Xeon X5960 with 96 GB of memory. The mutator and the garbage collector are single-threaded and stay on the same core throughout each experiment. `min_size_thresh` is 4096, `max_size_thresh` is initially set to one sixteenth of the number of objects, `out_thresh` is 2.5, `sink_thresh` is 100, and `run_thresh` is 1.1.

5.1 The Hacker News Application

Hacker News is a social news aggregation web site. We use the publicly available source[1], which runs on Racket. Most activity consists of viewing comments on articles, submitting articles, and submitting comments on both articles and other comments ("news items"). Hacker News is sensitive to full collection pause times since user requests cannot be served during a collection, resulting in user-perceivable delays of multiple seconds. We modified the code to disable per-user request rate limiting and to allow comments on news items of any age.

The application's database is populated with the most recent 500,000 news items

[1] http://arclanguage.org/install

from the real Hacker News[2]. The software keeps active news items in memory, and the 500,000 consume approximately 3 GB of memory. A single large hash table called `items*` contains an entry for each news item; each news item is implemented as a small hash table. When a new news item is submitted, a new hash table is allocated and populated with the item's contents. A reference to the new hash table is then inserted into `items*`. A new comment is added to a list of children attached to the commented-on news item.

We drive Hacker News with a client program that runs on a different machine and issues HTTP requests over a TCP connection. The client and server are connected with gigabit Ethernet and the round-trip time is approximately 100 microseconds. The client first discovers the set of 500 most recent news items; call them the "active set". The client then issues 600k requests, one after the other in a closed loop. 99% of the requests are to read a randomly chosen item from the active set. 1% of requests are to add a new comment to an item chosen randomly from the active set.

5.2 Effect of Cluster Collection on Pause Times

The purpose of the Hacker News experiment is to quantify the reduction in full collection pause times and the overhead of Clustered Collection on a realistic application. We run Hacker News and the client program twice: once with the stock Racket garbage collector and once with Clustered Collection.

Figure 5-1 shows the results. There is one numbered row for each full collection, showing the wall-clock times for Clustered Collection and the stock collector. Collections before the first Cluster Analysis are omitted; they take an average of 1.6 seconds for Clustered Collection, and 1.4 seconds for the stock collector. The client program halts after collection 24. Cluster Analysis runs twice, at the points indicated by the bold rows. The time shown on each Cluster Analysis row includes the time for the associated full collection.

Figure 5-1 shows that Clustered Collection reduces full collection pause times by

[2]http://news.ycombinator.com

n	Cluster (s)	Stock (s)	Ratio
Cluster Analysis	**11.7**	-	-
15	-	4.0	-
16	0.6	3.9	6.5
Cluster Analysis	**3.1**	-	-
17	-	3.3	-
18	0.8	3.3	4.1
19	1.7	3.4	2.0
20	0.8	3.0	3.7
21	1.5	3.5	2.3
22	0.8	3.3	4.1
23	1.4	3.3	2.3
24	0.8	3.3	4.1

Figure 5-1: The pause times in seconds for full collections and Cluster Analysis (**CA**) for the Hacker News application. Each Cluster Analysis time includes the time for the associated full collection. The Ratio column shows Stock time divided by Clustered Collection time. The average ratio (counting the first Cluster Analysis as 0.34) is 3.0.

Phase	Cluster (s)	Stock (s)
Set written bits	0.02	-
Mark sink objects	0.02	-
Mark+Copy	0.30	2.5
Pointer fix-up	0.20	1.4
Reset cluster marks	0.06	-

Figure 5-2: The most costly phases during full collection 16 in the Hacker News experiment.

more than a factor of two, and often by a factor of four. Figure 5-2 shows that, as expected, the biggest reduction is in the pointer tracing part (Mark+Copy) of full collection, since Cluster Collection does not need to trace within clusters.

The first Cluster Analysis takes 11.7 seconds. This is longer than a full collection due to the extra passes made by the Cluster Analysis implementation (see Figure 5-3). On the other hand, the second Cluster Analysis takes only 3.1 seconds, and if the client kept running subsequent ones would take about the same amount of time. The subsequent Cluster Analyses are less expensive than the first because most of the live data is still clustered, allowing Cluster Analysis to skip most objects.

	Cluster	Stock
Cluster ID assignment	3.1	-
Mark+copy	4.3	2.6
Out pointer discovery	2.4	-
Pointer fix-up	1.9	1.4

Figure 5-3: Time in seconds for each phase of the first Hacker News Cluster Analysis; the total is 11.7 seconds. The second column shows the times for the phases of the full collection at the same point in the Stock experiment. Cluster Analysis copies much more than the stock collector; the former copies all clustered data, while the latter rarely copies old-generation objects.

A second Cluster Analyses is needed in Figure 5-1 because of program writes that dissolve clusters, in particular writes caused by the client submitting new comments. Adding a new comment to `items*` does not dissolve any clusters, but does cause a new out-pointer to be added to the cluster of the affected split of `items*`. However, adding the new comment to the commented-on item's list of child comments does cause dissolution of the commented-on item's cluster. There are relatively few clusters (see Figure 5-4), so even though the client program only comments on the active set, this causes a significant fraction of clusters to be dissolved. As a result, a second Cluster Analysis is needed at the start of full collection 17. At this point, the client program has already commented on almost all of the active set, so that the hash tables implementing those news items have `o.written` set. The second Cluster Analysis omits those hash tables from its clusters, so that the client won't cause the new set of clusters to be dissolved. Thus after the second Cluster Analysis, no further analysis is needed in order to maintain good pause time reduction.

The ratios in the last column of Figure 5-1 fluctuate because the two collectors run at somewhat different times in the program execution, and thus with somewhat different amounts and structure of live data.

Figure 5-5 summarizes some overall Clustered Collection costs. Despite reducing full collection pauses, Clustered Collection causes the program to run slower overall, mostly because young collections take about 34% longer. Young collections are slower because Clustered Collection adds fields to the entries of Racket's "page" table, which

Total objects	44,425,018
Pct. of objects clustered	95%
Total clusters	415
Avg. object count per cluster	102,110 ±139,710
Clustered obj. lost due to writes	2,643,470
Out pointer per clustered object	0.003
Total sink objects	164

Figure 5-4: Statistics for the clusters found in the first Cluster Analysis in the Hacker News experiment.

	Stock	Cluster
Runtime (s)	2685	2968
Requests/second	223	202
Avg. young GC pause	69 ± 41 ms	93 ± 47 ms
Peak mem. use	3,866 MB	6,734 MB

Figure 5-5: Run-time information for the Hacker News experiment.

is consulted for every object during collection; these fields increase the amount of data that must be read from RAM. A second significant cost is the comparison of modified pages with their shadow copies. Clustered Collection uses two times as much memory as the stock collector while it is moving clustered objects to separate heap pages; old copies of the evacuated objects cannot be freed until all cluster objects have been copied. This momentary 2× memory increase is a limitation of the implementation – a more sophisticated implementation would use compaction to evacuate objects to private heap regions, requiring 1.3× memory overhead at most. Clustered Collection also allocates shadow pages, though these do not affect its peak memory use.

To summarize, Clustered Collection reduces full collection pause times by an average of 3× for Hacker News, while decreasing overall throughput by about 10%.

Old	New	Number	Dissolved?
in pointer	any	38,606	Yes
nil/value	nil/value	261,634	No
nil/value	in/out pointer	161,623	No
Total		461,863	

Figure 5-6: The number of different kinds of program writes to clustered objects during the Hacker News experiment. "In" pointer modifications force dissolution of the containing cluster while other writes do not.

5.3 Tolerating Writes

Some program writes force Clustered Collection's Watcher to dissolve the surrounding cluster, while others can be tolerated without dissolution (see Section 3.7). This chapter explores how effective the Watcher is at tolerating Hacker News' program writes.

We recorded the number of writes to clustered objects during the Hacker News experiment; Figure 5-6 shows the results.

Only writes where the old value is an "in" pointer (pointing to an object in the same cluster) force a cluster to be dissolved. These are rare in Hacker News: 38,606 out of 461,863 writes to a field in a clustered object alter an "in" pointer. These writes occur when a new comment is added to a news item's child comment list; the 16 clusters that suffer this kind of write are dissolved.

The other writes do not force dissolution. The most common writes were to change a `nil` to an integer value, or change one integer to another (e.g., increment a counter). Modifications that change a `nil` to an "in" or "out" pointer are also common with a count of 161,623. Most of these occur when a new comment is added to `items*`. If the new pointer is an "out" pointer, the write may cause a new entry to be added to the cluster's out-pointer set. The watcher ignores these modifications since they cannot change any object's liveness.

The Watcher significantly reduces the number of cluster-dissolving writes; only 8% of all modifications to clustered objects result in cluster dissolution.

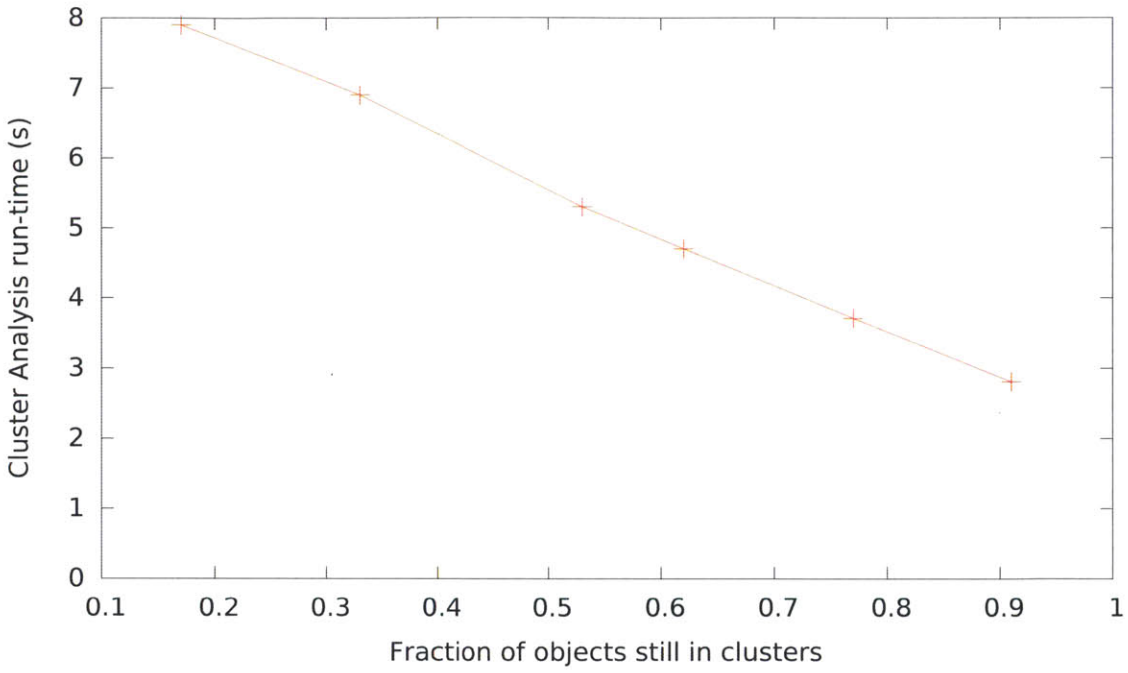

Figure 5-7: The effect on Cluster Analysis run-time of the fraction of objects still in clusters since the previous Cluster Analysis. The time includes both the Cluster Analysis and the associated full collection.

5.4 Later Cluster Analyses

Cluster Analyses after the first skip tracing within clusters that are still valid (haven't been dissolved). This chapter explores how much this technique speeds up Cluster Analysis.

The experiment uses Hacker News with the same setup as in Section 5.2, except that the client program comments on news items chosen randomly from the entire loaded set of 500,000. The reason for this change is to allow control over the number of clusters dissolved between one Cluster Analysis and the next: that number will be close to the number of random comments created, since each comment will dissolve the cluster that contains the commented-on news item. Each experiment loads the 500,000 news items, runs Cluster Analysis, lets the client program insert a given number of comments, runs Cluster Analysis a second time, and reports the second Cluster Analysis' run time.

Figure 5-7 shows the results. The graph depicts the time taken in seconds for the

second Cluster Analysis as a function of the fraction of live objects that are still in non-dissolved clusters when the second Cluster Analysis runs. Cluster Analysis with 91% of the data still clustered is more than twice as fast as when only 17% of the data is still clustered. The implication is that Clustered Collection will be most effective for programs that leave much of their data untouched.

5.5 Effect of Cluster Out-Pointers

Clusters with fewer out pointers are likely to yield faster full collections. This experiment explores the effect of out pointers on collection pause times.

The benchmark builds a binary tree of 32 million nodes. Every node in the tree also contains a pointer that either references a special object, or is null. The special object is not a member of any cluster. We vary the fraction of objects that refer to the special object in order to vary the number of out pointers. Cluster Analysis finds 716 clusters on each run. The number of out pointers between clusters are few and are included in the listed percentage of out pointers in the table.

Figure 5-8 presents the results. The break-even point occurs when about 40% of objects have out-pointers; at that point, the cost of tracing the out-pointers outweighs the benefits of not tracing within the clusters.

5.6 k-Nearest Neighbors

The purpose of the k-Nearest Neighbors experiment is to evaluate the overhead and pause time reductions achieved by Clustered Collection on a program whose live data and operation are much different from Hacker News.

The k-Nearest Neighbors program used in the experiment was downloaded from the Internet[3]. The program starts by loading a training data set and a target data set to be classified from disk. Each data set is stored in a list. Each list item contains five pointers: four reference a real number object and one references a string. There

[3]http://spin.atomicobject.com/2013/05/06/k-nearest-neighbor-racket/

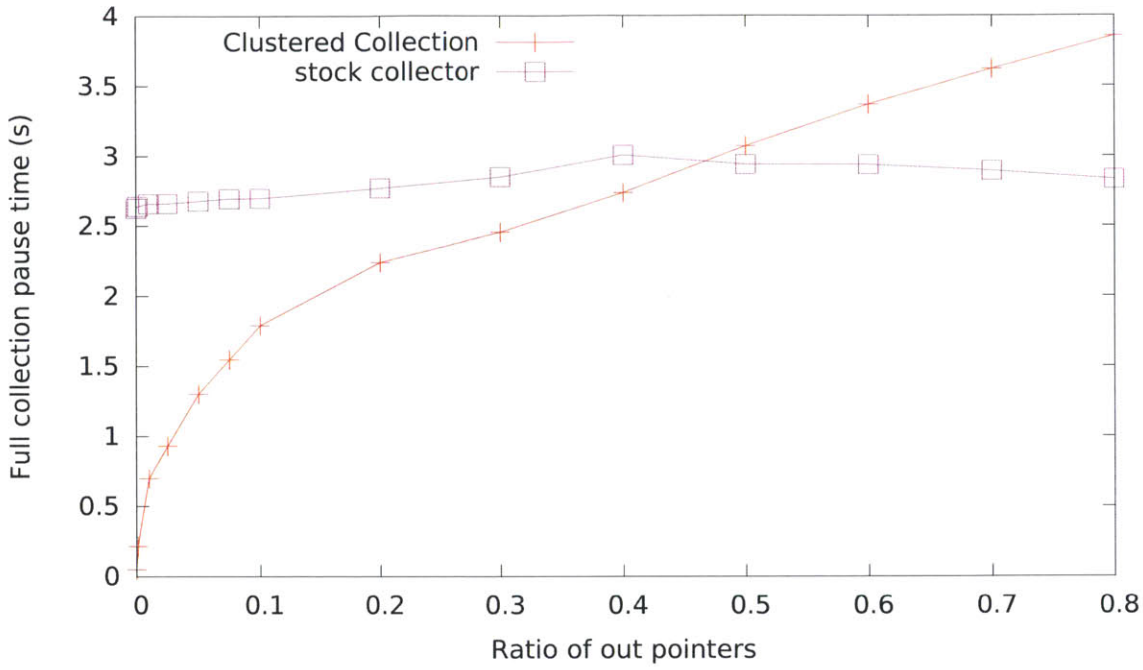

Figure 5-8: The effect of the fraction of objects that contain out-pointers on Clustered Collection's ability to reduce pause times.

are 3.2 million training items and 50 target items. The program classifies each target item; each requires two full passes over the training data. The training and target data sets are randomly generated. The training data uses about 1.25 GB of memory, while the target data uses about 10 KB.

In the experiment the program is executed twice: once with Clustered Collection and once with the stock collector. In both runs there are 33 full collections. Clustered Collection performs one Cluster Analysis after the k-Nearest Neighbors program has started classifying the target data. The pause times of every collection are recorded, along with total application execution time and peak memory use.

Figure 5-9 shows the pause times of the Clustered and stock collections after Cluster Analysis. The pause times of both collectors during execution of k-Nearest Neighbors are shown in ascending order of pause time, because the Stock and Clustered collections do not occur at the same points in the program. Sorting allows the distributions of pause-times to be compared. The elbow in the graph is due to the size of the live data fluctuating by 25% between collections. The average Clustered

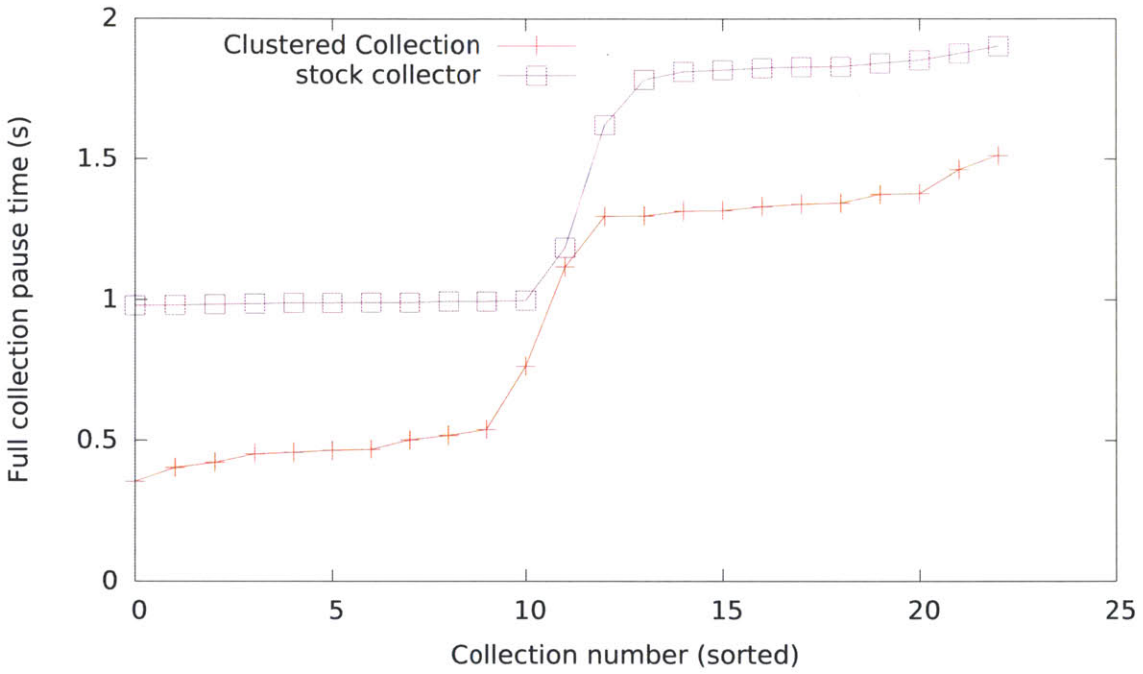

Figure 5-9: k-Nearest Neighbors full collection times for Clustered Collection (red) and the stock collector (purple). The collections of each type are sorted in ascending order by pause time. The average Clustered Collection takes 0.6 times as long as the average stock collection.

	Stock	Cluster
Total run-time	329 secs	339 secs
Avg. young GC pause	31 ± 17 ms	37 ± 25 ms
Young GC count	2287	2283
Peak mem. use	2603 MB	5360 MB

Figure 5-10: Run-time information for the k-Nearest Neighbors experiment.

Total objects	26,161,936
Pct. of objects clustered	88%
Total clusters	47
Avg. object count per cluster	$492,818 \pm 109,830$
Clustered obj. lost due to writes	0
Out pointer per clustered object	.1
Total sink objects	108

Figure 5-11: Statistics for the clusters found in the k-Nearest Neighbors experiment.

Collection takes 0.6 times as long as the average stock collection.

Figure 5-11 shows statistics about the clusters found by Cluster Analysis. Cluster Analysis took 1.7 seconds to execute and the discovered clusters are equally-sized segments of the training data and target data lists making up 88% of the programs live data. No clusters were dissolved because the training and input data are read-only. The clusters contain almost no "out" pointers because elements in the lists that make up the live data are shallow.

The full collection pause times for the Clustered Collector average 0.6 times as long as those of the stock collector. The cost, as shown in Figure 5-10, is that overall program run-time increases from 329 seconds to 339 seconds. Thus Clustered Collection reduces pause time significantly while increasing run-time by just 3%.

Chapter 6

Conclusion

Clustered Collection significantly reduces full collection pause times for applications with large amounts of mostly read-only data whose writes have locality in the object graph. Collection pause times are reduced by finding clusters of objects that can be skipped without sacrificing safety or completeness. Writes that may violate the invariants required for safety or completeness are handled correctly. An evaluation of Clustered Collection in Racket shows that pause times are significantly reduced at some cost in memory and application throughput.

Bibliography

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New Orleans, Louisiana, USA, 2003. ACM.

[2] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978.

[3] Myra Cohen. Clustering the heap in multi-threaded applications for improved garbage collection. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1901–1908, Seattle, WA, USA, 2006. ACM.

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, Vancouver, BC, Canada, 2004. ACM.

[5] Robert Bruce Findler and PLT. Drracket: Programming environment. Technical Report PLT-TR-2010-2, PLT Design Inc., 2010. http://racket-lang.org/tr2/.

[6] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[7] Barry Hayes. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 33–46, Phoenix, Arizona, USA, 1991. ACM.

[8] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 359–373, Anaheim, California, USA, 2003. ACM.

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

[10] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for multiprocessors. In *Proceedings of the 6th*

International Symposium on Memory Management, ISMM '07, pages 159–172, Montreal, Quebec, Canada, 2007. ACM.

[11] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise Garbage Collection for C. In *Proceedings of the 9th International Symposium on Memory Management*, ISMM '09, Dublin, Ireland, June 2009. ACM.

[12] 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, San Jose, California, USA, 2011. ACM.

[13] Michal Wegiel and Chandra Krintz. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 91–102, Seattle, WA, USA, 2008. ACM.