Reducing Pause Times with Clustered Collection

Cody Cutler Robert Morris
MIT CSAIL, USA
ccutler@csail.mit.edu, rtm@csail.mit.edu

Abstract

Each full garbage collection in a program with millions of objects can pause the program for multiple seconds. Much of this work is typically repeated, as the collector re-traces parts of the object graph that have not changed since the last collection. Clustered Collection reduces full collection pause times by eliminating much of this repeated work.

Clustered Collection identifies clusters: regions of the object graph that are reachable from a single "head" object, so that reachability of the head implies reachability of the whole cluster. As long as it is not written, a cluster need not be re-traced by successive full collections. The main design challenge is coping with program writes to clusters while ensuring safe, complete, and fast collections. In some cases program writes require clusters to be dissolved, but in most cases Clustered Collection can handle writes without having to re-trace the affected cluster. Clustered Collection chooses clusters likely to suffer few writes and to yield high savings from re-trace avoidance.

Clustered Collection is implemented as modifications to the Racket collector. Measurements of the code and data from the Hacker News web site (which suffers from significant garbage collection pauses) and a Twitter-like application show that Clustered Collection decreases full collection pause times by a factor of three and six respectively. This improvement is possible because both applications have gigabytes of live data, modify only a small fraction of it, and usually write in ways that do not result in cluster dissolution. Identifying clusters takes more time than a full collection, but happens much less frequently than full collection.

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

Keywords Garbage collection, memory management

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754184

1. Introduction

A major cost in tracing garbage collectors is the need to examine every live object in order to follow its child pointers. If there are millions of live objects this tracing can take multiple 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 [1, 3, 5, 8, 9, 11]. Many of these techniques would benefit if they had to trace fewer objects.

This paper's core idea is to reduce pause times by exploiting the similarity of the garbage collection computation from one run to the next. Successive full collections can potentially skip re-tracing regions of the object graph where no object's liveness has changed since the last collection. One challenge is ensuring completeness and safety despite not tracing all objects, particularly in the face of program writes that change the object graph. Completeness means that garbage collection frees all unreachable objects, and safety means that it frees no reachable objects. The other main challenge is choosing a set of regions whose graph structure ensures that skipping yields substantial tracing speedups.

Clustered Collection addresses these challenges 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 program has not modified any of the cluster's objects, the entire cluster is live and need not be traced. In that case tracing skips over the cluster and resumes at the cluster's "out" pointers.

If the program has modified a cluster's objects, the situation is more complex: such a modification may break the property that reachability of the head implies liveness of all of the cluster's objects, and that the cluster's "out" pointer list include all references to external objects. Simply dissolving a cluster in response to any write to it, while correct, would likely eliminate any performance win from Clustered Collection. Instead, Clustered Collection handles many writes without dissolving clusters. For example, a write that creates a reference to an object outside the cluster is handled by adding

an entry to the cluster's "out" list. The only situation in which a cluster must be dissolved is when a write eliminates (overwrites) a reference to an object inside the same cluster; that may cause the object to be dead, so completeness requires that the cluster be re-traced during the next full collection.

Clustered Collection chooses clusters likely to be helpful in reducing full collection pause times. One 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 cluster boundaries. However, clusters should not be too large, in order to limit the impact of each cluster-dissolving write. Finally, objects recently subject to a cluster-dissolving write are omitted from newly formed clusters.

The paper presents an implementation of Clustered Collection as a modification to Racket's precise single-threaded generational collector [4, 12]. The main challenge in this implementation is Racket's page-granularity write barriers, since Clustered Collection needs to know about writes at object granularity. The implementation creates 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 paper's 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 Twitter-like application handling millions of messages. Clustered Collection reduces full-collection pause times by an average of 2.9× for the former and 6.6× for the latter. The periodic Cluster Analyses take longer than a single full collection, but the evaluation shows that Cluster Analysis needs to occur much less frequently than full collection. 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. Clustered Collection is compatible with generational collection.

This paper's novel contributions include: 1) the idea that full collections can avoid re-tracing large regions of the object graph without sacrificing completeness; 2) techniques to cope with many program writes to clustered objects without having to dissolve the surrounding clusters; 3) heuristics for choosing clusters that yield good performance; and 4) an evaluation exploring the conditions under which Clustered Collection is most beneficial.

2. Related Work

Hayes [6] 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 (the collector which we present and evaluate in this paper is also generational). 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 [3, 7] partition the heap and are able to collect just one partition at a time, reducing worst-case pause times. The G1 collector [3] 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 [5] garbage collectors reduce pause times for full collections by running the collection on multiple cores. Clustered Collection seems likely to be helpful if added to a parallel collector.

Concurrent/real-time [1, 8, 10, 11] collectors reduce collection pause times by performing most of the collection while the application runs. These collectors have short pause times, but interleaving the collection work with program execution typically reduces program throughput. Informal measurements of HotSpot's Concurrent Mark and Sweep collector suggest that it reduces program speed by 20% to 30% compared to HotSpot's Serial collector. Section 6 shows that, while Clustered Collection yields a smaller reduction in pause time, it also has a much lower impact on program throughput.

Cohen [2] 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.

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 tracing of those objects.

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 take action 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 clusters.

Suppose a program's object graph looks like part A of Figure 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 will imply that all of the cluster's objects are live. Pointers may also enter a cluster to non-head objects, but only a 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 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 tags 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, 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. When the mark phase encounters the "head" object of the remaining cluster, 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

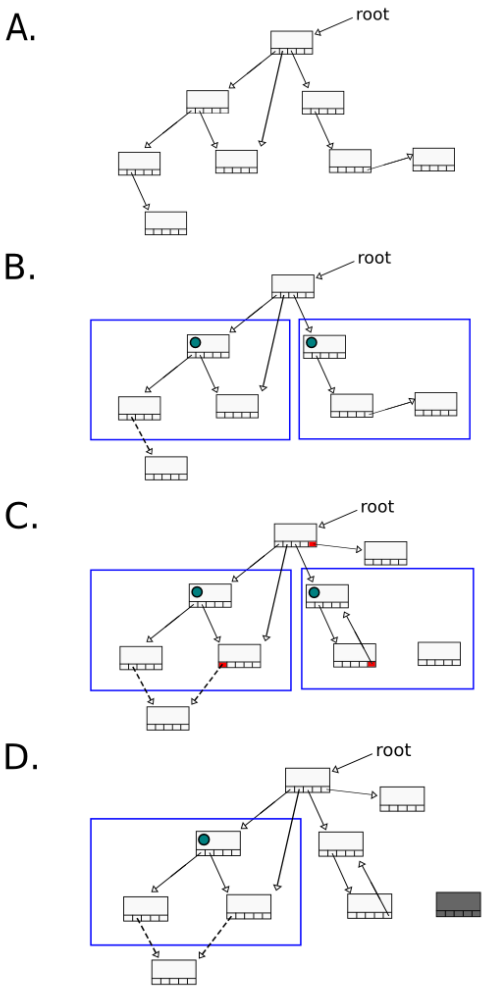

Figure 1. An example clustering. Nodes with green circles are cluster head objects, arrows are pointers, dashed arrows are "out" pointers, and blue boxes enclose clusters. Part A shows an application's live data, part B shows a possible choice of clusters, part C depicts the live data after the program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

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 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

An important aspect of the Clustered Collection design is the way it chooses clusters. This section 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 via intra-cluster pointers; 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 (a "dissolving write"), 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 that have suffered one dissolving write may be likely to suffer another. Thus it is good to omit such objects from future clusters.
- 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 sub-tree's 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 this state:

- `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 object has suffered a dissolving write recently.
- `c.mark`: for each cluster, a flag indicating whether the cluster's head has been reached during the current full collection.
- `c.camark`: for each cluster, a flag indicating whether the cluster's out-pointers have been clustered during Cluster Analysis.
- `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 heap 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 either program writes cause clusters to be dissolved or newly allocated objects are accumulated, until it reaches `run_thresh` and Cluster Analysis runs again.

Cluster Analysis pseudo-code is shown in Figure 2. At any given time, Cluster Analysis has a work stack of (o, c) pairs, each indicating that object o should be considered for inclusion in cluster c. If Cluster Analysis has already visited object o or if `o.cluster.camark` is set, Cluster Analysis ignores o. If `o.cluster.camark` is not set, Cluster Analysis sets `o.cluster.camark` and pushes `o.cluster.out` to the work stack with `o.cluster` as the destination cluster. If `o.written` is set, or o 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. If c is not null, and adding o to 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' onto the work stack.

Cluster Analysis then moves each cluster's objects to a separate region of memory, so that the cluster's objects are not intermingled with other objects. This has several

```
cluster_analysis():
  for r in roots:
    cluster1(r, nil)
  move objects to per-cluster separate memory
  fix up pointers to clustered objects
  for c in clusters:
    calculate c.out
  for c in clusters:
    op_ratio = numoutpointers(c) / size(c)
    if op_ratio > out_thresh:
      destroy c
    else if size(c) < min_size_thresh:
      destroy c

cluster1(object o, cluster c):
  if live_cluster(o.cluster):
    if o.cluster.camark
      return
    o.cluster.camark = true
    for o1 in o.cluster.out:
      cluster1(o1, o.cluster)
    return
  if o.mark:
    return
  o.mark = true
  size_ok = size(c) + 1 < max_size_thresh
  if o.written or is_sink(o):
    c = nil
  else if c!=nil and size_ok
    o.cluster = c
  else:
    c = new cluster
    o.cluster = c
    o.head = true
  for o1 in children(o):
    cluster1(o1, c)
```

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

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 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 sets the corresponding bit in the out pointer bitmap if the object 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 cluster 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 dissolving writes.

`max_size_thresh` is initialized to 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, the danger from too-large clusters is evidently greater than that from too-small ones, so Cluster Analysis halves `max_size_thresh`. Otherwise it doubles `max_size_thresh`.

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 clusters' "out" sets will become "sink" objects if the number of references exceeds a threshold, `sink_thresh`.

To preserve completeness, the immortal regions have a limited life-time – until the next Cluster Analysis. The next Cluster Analysis dissovles the pre-existing immortal regions and treats all objects in the immortal region as normal objects (which can themselves be again added to a new immortal region or a new cluster).

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 that have recently suffered dissolving writes. Clustered Collection can use the write barriers that are usually required for a generational garbage collector (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 an object o has a slot that used to point to o_o, and that the program changes it to point 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 example, 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 intra-cluster tracing. If the head of an object's cluster is traced after the 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 pointer slot in 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.

4. Implementation

We implemented Clustered Collection as a modification to the precise collector in Racket [4, 12] 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` only for clustered objects.

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 3. 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 3 (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.

5. Applications

We use two applications to evaluate Cluster Collection.

5.1 Hacker News

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.

The application's database is populated with the most recent 500,000 news items from the real Hacker News[2]. The software keeps active news items in memory, and the 500,000 news items 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*`.

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

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

A new comment is added to a list of children attached to the commented-on news item.

5.2 Squawker

Squawker is a Twitter-like service written in Racket. Each user can post messages, subscribe to other users' messages, unsubscribe, and read messages from the users to whom they subscribe. Users talk to the service over TCP.

For each user, the implementation maintains a subscription hash table, a message list, and a message list tail pointer. The implementation maintains three hash tables indexed by user IDs to hold this data. The keys of User A's subscription hash table are the user IDs of all users that have ever subscribed to A and each value indicates whether the user is currently subscribed. User A's message list contains the messages that A should see, i.e. the posts of all users that A subscribes to. When user A subscribes to user B, A's identity is added to B's subscription hash table. When B posts a message, a message object is appended to the message list of every user for which an identity exists in B's subscription hash table, and each subscribing user's message list tail pointer is updated.

The Squawker implementation works well with Clustered Collection because the primary operation (posting new messages) does not dissolve the clusters holding the bulk of the data (the per-user message lists). Each new message is appended to each subscribing user's message list. This causes the null pointer at the end of the list to be changed to point to a new list element. The rules described in Section 3.7 allow this update to proceed without requiring dissolution of the cluster containing the message list, though an entry will be added to the cluster's "out" set.

6. Evaluation

This section 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, `out_thresh` is 2.5, `sink_thresh` is 100, and `run_thresh` is 1.1, and `max_size_thresh` is set to the number of objects.

6.1 Hacker News Pause Times

This experiment quantifies Clustered Collection's effect on full collection pause times for a real application, Hacker News. It compares Clustered Collection with the stock Racket garbage collector.

The client program runs on a separate machine and issues HTTP requests over TCP. The client is fast enough that the server is the bottleneck, not the client. The client issues 1.5 million requests. 99% of them read one of the most recent 500 news items; the other 1% add a new comment to a recent item. There is initially 2,760 MB of live data, and 3,068 MB by the end of the experiment.

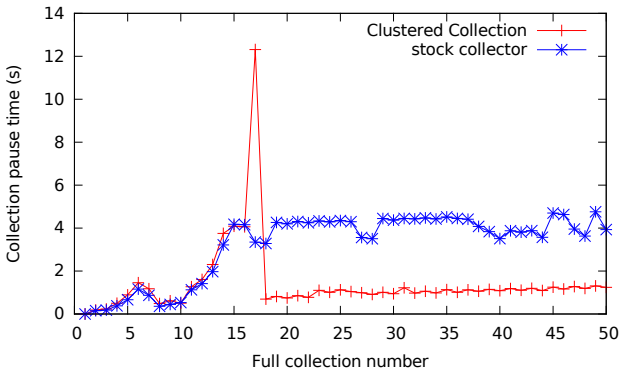

Figure 4. Hacker News pause times for full collections and Cluster Analysis, which occurs during collection 17.

Phase	Cluster (ms)	Stock (ms)
Set written bits	7	-
Mark sink objects	0	-
Mark+Copy	302	2,695
Pointer fix-up	173	1,544
Reset cluster marks	118	-

Figure 5. The most costly phases during full collection 18 in the Hacker News experiment.

Figure 4 shows the results. It contains one data point per full collection, with the collection's pause time on the y axis. The size of the live data fluctuates at collections 1 through 7 while the server loads the 500,000 news items and finishes the initial loading at collection 15. The server then begins to service the client's requests.

The peak at full collection 17 in Clustered Collection's pause times corresponds to the execution of Cluster Analysis. After the Cluster Analysis, Clustered Collection's full collection pause times drop to an average of 0.25× as long as the stock collector. Figure 5 confirms that most of the reduction is in pointer tracing (Mark+Copy): Cluster Collection does not need to trace within clusters. This multi-second reduction in pause time is the main benefit of Clustered Collection.

The Cluster Analysis in Figure 4 takes 12.2 seconds; Figure 6 breaks down this time. While expensive, Cluster Analysis is infrequent: the next one would occur during full collection 87. As Section 6.5 shows, subsequent Cluster Analyses are typically faster than the first one.

Figure 7 contains details describing the clusters found during the Hacker News experiment. 95% of the 44 million objects are clustered into 458 clusters. Less than 2% of the clustered objects are lost due to pointer modifications. Out

	Cluster (ms)	Stock (ms)
Cluster ID assignment	3,109	-
Mark+copy	4,296	2,134
Out pointer discovery	2,476	-
Pointer fix-up	1,583	1,076
Misc.	367	83

Figure 6. Time in milliseconds for each phase of the Cluster Analysis during collection 17 of Hacker News; the total is 12.2 seconds. The second column shows the times for the phases of the full collection at the same point for the stock collector. Cluster Analysis copies much more than the stock collector; the former copies all clustered data, while the latter rarely copies old-generation objects.

Total objects	44,425,018
Pct. of objects clustered	95%
Total clusters	458
Avg. object count per cluster	93,147
Clustered obj. lost due to writes	557,196
Out pointer per clustered object	0.006
Total sink objects	164

Figure 7. Statistics for the clusters found in the first Cluster Analysis in the Hacker News experiment.

	Cluster	Stock
Runtime (s)	5,493	5,715
Requests/second	273	262
Avg. young GC pause (ms)	79	79
Average live data size (MB)	2,903	2,967
Peak shadow page use (MB)	7	-
Full GC peak memory (MB)	3,317	3,703
CA peak memory (MB)	5,688	-

Figure 8. Run-time information for the Hacker News experiment. Times are wall-clock time.

pointers in these clusters are rare: the ratio of out pointers to clustered objects is 0.006. It is good that nearly all the live data is clustered and out pointers are few: the result is that the Mark+copy phase has a reduced workload since the tracing of the clustered objects will be skipped and few out pointers will be traced.

Figure 8 summarizes some of Clustered Collection's effects on run-time and memory use. Hacker News with Clustered Collection serves requests 4% faster than with the stock collector. The maximum amount of memory used by shadow pages is 7 MB, less than 1% of the live data size.

Old	New	Number	Dissolved?
in pointer	any	37	Yes
nil/value	nil/value	814	No
nil/value	in/out pointer	1,380	No
Total		2,231	

Figure 9. 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.

Figure 8 shows that the two collectors have similar peak memory use during typical full collections. Cluster Analysis requires more memory than a full collection: roughly 2× the size of the live data, since Cluster Analysis copies all the clustered objects. While in this experiment the stock collector has a lower peak memory use than Clustered Collection, in fact they have roughly the same worst-case memory requirements. The reason is that the stock collector sometimes copies live data in order to de-fragment memory. Whether it copies each page of memory depends on whether the page is more than 25% empty; given enough time, at some point most of the pages will be simultaneously empty enough, and a single stock collection will end up copying most or all of the live data. Thus the worst-case memory requirement for the stock collector is also 2× the size of the live data. For this reason the comparison is fair with respect to the memory available to each collector.

To summarize, Clustered Collection's pause times reduction over all collections (including the Cluster Analysis) is 2.9× and increases throughput by 4% for Hacker News.

6.2 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 section explores how effective the Watcher is at tolerating Hacker News' writes.

Figure 9 shows the numbers of different kinds of Hacker News writes as classified by the Watcher. 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: 37 out of 2,231 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 4 clusters that suffer this kind of write are dissolved.

The other writes do not force dissolution. The most common writes were modifications that change a `nil` to an "in" or "out" pointer with a count of 1,380. 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. Writes to change a `nil` to an integer value, or change one integer to another (e.g., increment a counter) are also common with 814 occurances.

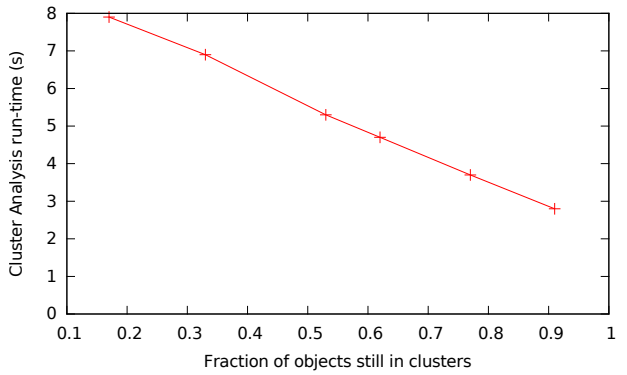

Figure 10. The effect on Cluster Analysis run-time of the fraction of objects still in clusters since the previous Cluster Analysis. The time includes both the Cluster Analysis and the associated full collection.

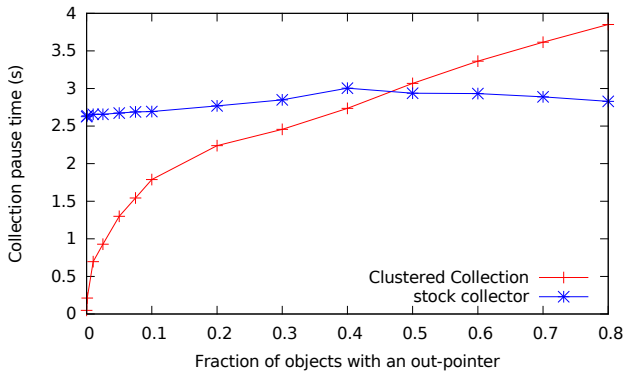

Figure 11. The effect of the fraction of objects that contain out-pointers on Clustered Collection's ability to reduce pause times.

The watcher ignores these modifications since they cannot change any object's liveness.

The Watcher avoids dissolving the cluster for all but 1.6% of program writes.

6.3 Later Cluster Analyses

Cluster Analysis preserves existing clusters and forms new clusters only from previously unclustered objects. This section explores how much this technique speeds up Cluster Analysis.

The experiment uses Hacker News with the same setup as in Section 6.1, 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 10 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. This experiment shows that preserving existing clusters significantly reduces Cluster Analysis run-times.

6.4 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. Once the binary tree is built, Cluster Analysis is run, followed by a final full collection which is timed. Cluster Analysis finds 716 clusters on each run.

Figure 11 presents the results. The x-axis is the fraction of objects that reference the special object and the y-axis is the pause time of the final full collection. When there are no out-pointers, Clustered Collection's final collection takes only 51 ms compared to the stock collector's 2.6 seconds. Because each out-pointer is traced and repaired during a full collection, Clustered Collection's pause times increase as the live data gains out-pointers. The break-even point occurs when about 40% of objects have out-pointers; at that point, the cost of tracing the out-pointers out-weighs the benefits of not tracing within the clusters. This experiment shows that Clustered Collection reduces pause times more effectively when clusters have fewer out-pointers.

6.5 Squawker Pause Times

This experiment explores how general Clustered Collection is by measuring its performance on Squawker (see Section 5.2), an application significantly different than Hacker News.

The client runs on a separate machine and submits requests over TCP. The client initially creates 5,000 users, creates 43,357 subscriptions with Pareto popularity distribution (the most popular user has 4,950 subscribers), and creates one million posts from random users. The client then issues requests for 5400 seconds. The request types have these probabilities: fetch 10 latest messages 0.992, 256-byte post 0.005, subscribe 0.002, unsubscribe 0.001. There is 1,987 MB of live data after the initial posts, and 2,792 MB by the end of the experiment.

Figure 12 shows the results, with full collection number on the x-axis, and collection pause time on the y-axis. The peaks at 10 and 53 correspond to Cluster Analysis executions.

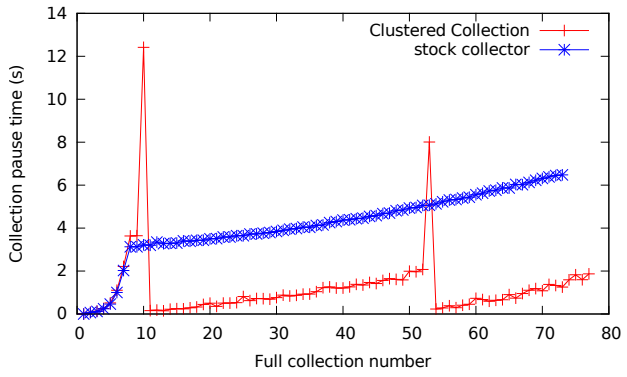

Figure 12. Pause times for full collections and Cluster Analysis for Squawker. Cluster Analysis occurred during collections 10 and 53.

	Cluster	Stock
Requests/second	13,004	13,428
Avg. young GC pause (ms)	4.6	2.2
Average live data size (MB)	2,256	2,276
Peak shadow page use (MB)	82	-
Full GC peak memory (MB)	2,964	3,006
CA peak memory (MB)	3,981	-

Figure 13. Run-time information for the Squawker experiment.

Collections 1 through 7 occurred while the client program was adding the initial million posts.

Collection pause times are over 20× shorter for Clustered Collection than the stock collector immediately after the first Cluster Analysis, because nearly all objects are included in clusters. This advantage decreases as new posts create unclustered objects. The second Cluster Analysis clusters these new posts, again yielding a nearly 20× full collection speedup. The second Cluster Analysis is faster than the first because most of the live data (approximately 74%) is still clustered. The average ratio of stock collector pause time to Clustered Collection pause time (including Cluster Analyses) is 6.6.

Figure 13 reports the stock and Clustered Collection average young generation collection pause times and requests completed per second. Squawker running under Clustered Collection had throughput 3% lower than under the stock collector. Much of this difference is due to slower young generation collections. Young generation collections under Clustered Collection are slower by 2.4 ms on average because the Watcher inspects all pointer modifications to old-generation objects during young generation collections.

For Squawker, it is important that Cluster Analysis extends existing clusters during the second Cluster Analysis instead of creating new clusters. After the first Cluster Analysis, each cluster holds multiple users' message lists. If a subsequent Cluster Analysis created new clusters to hold the new tails of the message lists, each tail would need its own cluster, since different users' tails would not share a head object. These clusters would all have an "out" pointer per message, referring to the shared string holding the message content. Tracing this large number of "out" pointers would eliminate any performance gain from clustering. Because Cluster Analysis instead grows existing clusters, each new message list tail is added to the cluster containing the existing message list; this preserves the property that each cluster contains many message lists that all share the same set of "out" pointers to message content strings, much reducing the total number of "out" pointers.

7. 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 it reduces full collection pause times by a factor of three to six times.

Acknowledgments

Thanks to Matthew Flatt and the anonymous reviewers for their feedback. We gratefully acknowledge the support of the National Science Foundation under awards 0964106, 1301934, and 0915164.

References

[1] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978. ISSN 0001-0782.

[2] M. 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.

[3] D. Detlefs, C. Flood, S. Heller, and T. 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.

[4] R. B. Findler and PLT. DrRacket: Programming Environment. Technical Report PLT-TR-2010-2, PLT Design Inc., 2010. `http://racket-lang.org/tr2/`.

[5] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985. ISSN 0164-0925.

[6] B. 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.

[7] M. Hirzel, A. Diwan, and M. 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.

[8] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: A Wait-free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, Beijing, China, 2012. ACM.

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

[10] B. McCloskey, D. F. Bacon, P. Cheng, and D. Grove. Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors. Technical report, IBM, 2008.

[11] F. Pizlo, D. Frampton, E. Petrank, and B. 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.

[12] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise Garbage Collection for C. In *Proceedings of the 9th International Symposium on Memory Management*, ISMM '09, Dublin, Ireland, June 2009. ACM.

[13] M. Wegiel and C. 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.