Hinted Collection

by

Philip Reames

A thesis submitted in partial satisfaction of the
requirements for the degree of
Masters of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

George Necula, Chair
Krste Asanović, Co-chair

Spring 2013

Abstract

Hinted Collection

by

Philip Reames

Masters of Science in Computer Science

University of California, Berkeley

George Necula, Chair

Krste Asanović, Co-chair

Garbage collection is widely used and has largely been a boon for programmer productivity. However, traditional garbage collection is approaching both practical and theoretical performance limits. In practice, the maximum heap size and heap structure of large applications are influenced as much by garbage collector behavior as by resource availability.

We present an alternate approach to garbage collection wherein the programmer provides untrusted *deallocation hints*. Usage of deallocation hints is similar to trusted manual deallocation, but the consequence of an inaccurate hint is lost performance not correctness. Our hinted collection algorithm uses these hints to identify a subset of unreachable objects with both better parallel asymptotic complexity and practical performance.

We present two prototype implementations of a stop-the-world hinted collector: one entirely serial and one parallel. We evaluate our implementations by comparing against the Boehm-Demers-Weiser [12] conservative garbage collector for C/C++. We leverage existing free calls in mature C programs to stand in for deallocation hints. On some benchmarks, our serial collector implementation achieves 10-20% pause time reductions over a well-tuned baseline. On four cores, our parallel implementation achieves similar benefits.

We include a discussion of the design trade-offs inherent in our approach, and lessons to be learned from our collectors. We close with a discussion of several design variants which we have not been able to explore in depth, but believe would be worthwhile to explore in future work.

Contents

Chapter 1

Hinted Collection

1.1 Introduction

According to one popular language survey [1], eight of the top ten languages in use today use some form of automatic memory management. Tracing garbage collection - in the form of sophisticated generational, concurrent, or incremental collectors, but in some cases in that of relatively simple stop-the-world collectors - is the most common mechanism used.

Several languages support a mixed model of memory deallocation - with some objects deallocated via automatic mechanisms and others deallocated manually - which highlights an interesting middle ground that is not well explored in the memory management literature. One language, Objective-C, uses a mixture of compiler-assisted reference counting and trusted manual deallocation. Other languages have well accepted best practices for nearly automatic memory management without language support. For example, C++ has the widely used std::shared_ptr template which provides a reference counting abstraction. Both schemes are unsound due to the trusted nature of manual deallocation, but not all combined schemes have to be.

In this thesis, we propose a new variety of garbage collector which relies on *hints* from programmers for performance, but not for correctness. Often, the developer of a program has a mostly accurate mental model of the lifetimes of objects in their program. We use this knowledge to convert the standard reachability problem of a tracing collector into an alternate form where a subset of hinted objects are confirmed as unreachable. We call such a collector a *hinted collector*.

From the user perspective, a hinted collector is a hybrid between a traditional garbage collector and an explicit memory allocator. Unlike a standard garbage collector, program performance can benefit from users' understanding of object lifetimes. The language is extended with a *deallocation hint* construct which - as its name implies - provides an untrusted hint to the runtime that the annotated object will not be reachable during the next collection. An inaccurate deallocation hint is wrong and should be fixed. Unlike in an explicit memory deallocation scheme, the penalty for being wrong is performance, not correctness.

The expectation is that most user-provided deallocation hints are accurate - i.e. the annotated object will be unreachable before the next collection cycle - and that it is feasible for the user to provide hints for most deallocated objects. Given our collective experience with languages like C & C++ with explicit memory allocation, we believe these to be reasonable assumptions. As we have learned the hard way, programmers' mental models of object lifetimes are not *always* correct. The tremendous prevalence of use-after-free, double-free, and uninitialized memory reads are strong evidence of this. However, the fact that we can write large applications in these languages at all is good evidence that developers' mental models are mostly accurate.

These assumptions allows us to restrict the problem we need to solve. Rather than attempting to reclaim *all* unreachable objects, we will only reclaim a subset of unreachable objects. In particular, we will assume that any object not *hinted* with a deallocation hint is live and will not attempt to reclaim it. As in all collectors, any object reachable from an object assumed to be live must also be assumed live. To do otherwise would be unsound.

Traversal Collector

Hinted Collector

Figure 1.1: Minimum number of parallel steps required by a traversal collector and a hinted collector to explore a long linked list.

This formulation gives fundamentally different scalability limitations than a standard tracing collector. The key advantage of a hinted collector is the removal of a constraint on the order in which edges can be visited. Consider the case of a long linked list which is live during the collection (Figure 1.1). A standard traversal mark algorithm must traverse the list in order from head to tail - likely with low locality and many cache misses - whereas a hinted collector can traverse the edges in the list in any desired serial or parallel order. As a result, heap shape - the structure of references connecting objects in the heap - is largely irrelevant to the performance of a hinted collector. Given that long data structures are not uncommon in real world programs, heap shape has been widely identified as a limit to the parallel performance of tracing collectors [33, 6, 5]. Removing this ordering dependency is a potentially profound change.

We present two working prototypes to illustrate the feasibility of hinted collection and to highlight the interesting properties of such a design. We co-opt the existing free calls in C and C++ programs to act as deallocation hints; this allows us to evaluate the feasibility of hinted collection on large programs. Our serial prototype achieves pause times which are 40-60% faster than a standard tracing collector on some microbenchmarks, and 10-20% faster on some of the SPEC 2006 benchmarks and one case study. Our parallel collector slightly outperforms the parallel traversal collector baseline across our benchmarks. Worth highlighting is that on one benchmark it achieves nearly a 75% reduction in maximum observed pause time. We explore the limits of such a design and highlight opportunities for future exploration.

A hinted collector does need to be paired with a backup collector to recover objects which become unreachable without being hinted. As we show, the actual leak rate of such objects is low in C and C++ programs - around 5% in the case study we considered. When paired with either a low frequency stop-the-world tracing collector or a concurrent collector, our collector would likely achieve better average pause times and overall throughput.

Interestingly, such a combined system would provide a clear path - by inserting additional hints or improving the accuracy of those already present - for performance tuning *without having to sacrifice memory safety*. We consider this to be one of the most exciting potential applications of hinted collection.

The key contribution of this work are:

- We introduce the concept of hinted collection, and frame the implicit graph problem - establishing unreachability for a subset of hinted objects - for comparison with the reachability formulation of standard garbage collectors.
- We present an algorithm for this problem which - when given exact hints - is asymptotically faster in parallel settings. When given inexact hints, the algorithm reduces to a standard reachability traversal over the inaccurately hinted objects.
- We present a mostly serial implementation of a hinted collector which outperforms a well tuned mark implementation by between 40-60% on microbenchmarks, and between 10-20% for some of the SPEC benchmarks and one real world case study.

- We present a parallel implementation with proven performance with four marker threads. On the same set of benchmarks, we match or slightly exceed the performance of a parallel traversal mark algorithm. On one benchmark (`perlbench`), we outperform the baseline by nearly 4x.
- We highlight lessons learned with the current collector and propose a modified hinted collector design so inspired. We reflect on the implications of hinted collection, and close with a extensive discussion of possible future research direction.

1.2 Background

At its most fundamental, a garbage collector is an engine for soundly identifying dead objects which can be reclaimed to recycle their allocated memory for future allocation. Garbage collection - as opposed to the more general term automatic memory management - is specifically the use of the reachability abstraction to arrive at such a sound approximation. At their heart, tracing garbage collectors use some traversal algorithm for solving graph reachability. All of the practical garbage collectors of which we are aware are tracing garbage collectors.

Reachability

The graph reachability problem is the following: given a graph consisting of objects (vertices), directed edges connecting objects, and a set of root objects assumed a priori to be live, mark all objects which are transitively reachable along any path from the root set.

Throughout this thesis we will use $V_{reachable}$ and $V_{unreachable}$ to describe the set of vertices reachable and unreachable by a traversal. These sets are not known a priori, but are useful for analysis purposes. An edge is reachable if the source vertex is reachable. When needed, we will use P to denote the number of processors available. When a parallel complexity is presented without an explicit reference to P, this should be read as the limit as P approaches infinity.

A property of reachability which will be useful in reasoning about the correctness of our hinted collector is the following: The set of objects reachable from an initial set of roots must be a (non-strict) superset of the set of objects reachable from a subset of those roots. Slightly more formally, this could be stated:

$$\forall R_1 \subseteq V, R_2 \subseteq V.(R_1 \subseteq R_2 \implies V_{reachable}(R_1) \subseteq V_{reachable}(R_2))$$

In this thesis, we compare against the class of traversal based algorithms (such as breadth-first-search, or depth-first-search). In principle, other classes of reachability algorithms could be used for a garbage collector, but we are not aware of a collector that does so. The closest might be the optimistic marking of [6]. Standard traversal algorithms for solving reachability have a serial complexity of $O(|V_{reachable}| + |E_{reachable}|)$ and a parallel complexity of $O(D)$, where D is the depth of the graph[1].

Garbage Collection

In the garbage collection literature, the application - which is ideally ignorant of all memory management details - is known as the mutator. The reachability problem described above is referred to as the mark phase of a tracing collector. Much of the work on garbage collection has been focused on improving two metrics: throughput (the number of dead objects collected per unit time), and pause time (the time during which the mutator can not run). Both of these metrics are usually dominated by time spent in the mark phase. A number of options have been explored for accelerating the reachability traversal including:

---

[1] In practice, contention on concurrent updates to make bits and synchronization between processors introduce further terms. We considered using an Concurrent Read Exclusive Write model to adjust for this, but found the results to be more complicated and no more insightful. We note that other classes of algorithms can compute reachability in less than $O(D)$ for specific graph structures; we are not aware of a practical algorithm for program heap graphs.

- Ordering the traversal to improve cache locality is not addressed in the asymptotic results, but is in practice a major concern. The difference between having an item in cache vs not can be roughly two orders of magnitude. As a result, numerous traversal orders have been explored [23, 14, 11].
- Executing the traversal using multiple hardware threads greatly decreases average pause times. As hinted by the asymptotic results, performance does not continue to scale forever. Even ignoring the costs of coordination, program heaps contain a finite degree of parallelism with deep data structures not being uncommon [6, 5, 26].
- Sub-dividing the heap into sections (as in generational and region-based collectors such as [34, 4, 10, 9]) which can be collected mostly independently greatly reduces the average pause time, at the cost of requiring some edges between regions to be tracked by the mutator. Worst case pause times are still determined by the overall heap structure and may even be worsened by a poor division.
- Splitting the mark phase into a series of smaller steps which are interwoven with the mutator (as in incremental or concurrent collectors) reduces the average pause time, but often reduces throughput. The fundamental issue is that the mutator is essentially racing with the collector; if the mutator ever exhausts the pool of reclaimed memory before the collector can refill it with unreachable objects, the mutator must block on the collector for an amount of time bounded only by that required to perform a full collection cycle.

Despite their limitations, such collectors are very widely used. Production collectors succeed in reducing pause times to levels that do not impact most programs, and - with the help of some wasted space - achieve "good enough" throughput rates. Current technology breaks down when applications have little tolerance for pauses, extremely high turnover rates, or heaps measured in GB rather than MB[2]. Painfully long pause times have been seen with nearly every production collector of which we are aware.

As we will explore, hinted collection has a parallel asymptotic complexity favorably comparable to reachability based collection - particularly when given exact deallocation hints.

Related Work

The only work we know of that directly addresses the fundamental scalability of parallel collection is that of Barabash & Petrank [6, 5]. They propose two techniques. The first is based on inserting shortcut links into the heap dynamically, but does not include any mechanism for keeping links updated between collections. The second uses optimistic marking from randomly chosen heap nodes with spare threads. The issue identified (and not addressed) is a high rate of floating garbage caused by the optimistic marking. Often, a practical responses to heap shape controlled pause times is to simply change the data structure used. We are not aware of research which investigates this approach.

In the realm of systems which combine manual and automatic memory management, probably the best well known is the line of conservative collectors for type-unsafe languages pioneered by Boehm and Weiser [12]. The Boehm-Demers-Weiser collector can be used to improve reliability by reclaiming leaks, avoid temporal safety bugs by handling all deallocations, or for reporting leaks during debugging.

In the most recent edition of the C++ standard, support for referencing counting (std::shared_ptr) and unique pointers (std::unique_ptr) has been added to the standard library (but not the language). In recent years, Objective-C has moved from being a language with only manual memory deallocation to a primarily reference counted language (with compiler support) where the use of manual deallocation is strongly discouraged.

There is a wide range of literature on detecting and debugging various classes of deallocation errors (i.e. temporal memory errors) in C and C++ programs [15, 28, 13]. The most relevant for our own work are attempts to create memory allocators which can transparently tolerate deallocation errors in production

---

[2] For non-relocating collectors, one must also add long running applications impacted by fragmentation to this list. This is out of the scope of this work.

environments through over-provisioning with randomized object placement [8, 29, 25], type-specific pool allocation [21, 2, 20], checkpointing and environmental perturbation [31], or runtime patching with probability based identification [30]. We do not address spatial memory errors (such as array bounds violations) in this work; our collector would be complemented by approaches such as baggy bounds checking [3] for detecting and tolerating such errors.

1.3 The Hinted Collector Algorithm

In this section, we present an idealized hinted collector. We focus on the mark phase of the collector, which must establish the invariant that any unmarked object can be safely reclaimed. A standard sweeping phase (either eager or lazy [11]) can follow the mark to actually do the reclamation. Following the discussion of the algorithm, we explore the fundamental scaling limits of such a collector (both in serial and parallel versions) and then close with a discussion of certain key properties of the algorithm.

The Problem

The graph problem posed to our collector is slightly different from the standard reachability problem solved by standard collectors. We still have a directed graph consisting of a set of objects (vertices), a set of directed edges, and a set of root objects that are assumed live. However, in addition, some of those objects are *hinted* - meaning the user has given a deallocation hint for that object since the last collection, whereas others are *unhinted*. Rather than seeking to identify all unreachable objects, the collector is only asked to identify a *subset of the hinted objects which are in fact disconnected from the roots - i.e. unreachable*. Another way to view the modified problem is to consider the set of hinted objects as an approximate solution to the standard reachability problem. The task at hand is to refine that approximate solution into a subset of objects which are, in fact, unreachable. It is this slightly modified problem statement that allows us to improve parallel scalability over a standard tracing collector. (See Section 1.3)

We term an individual hint *accurate* if the object so hinted is unreachable. We describe an unhinted unreachable object as having a *missing* hint. We term a set of hints where every hint is accurate and with none missing to be *exact*. We will occasional use the informal terms *mostly accurate* and *nearly exact* for sets of objects. We expect that in the common case, hints will be mostly accurate and few hints will be missing.

The Abstract Algorithm

The hinted collector marking algorithm (given in Figure 1.2) conceptually has three main phases: marking unhinted objects, marking objects directly reachable from unhinted objects, and a reachability traversal to locate objects which were hinted, but are actually reachable. When all hints are exact, only the first two execute. We note that the version presented in this section is organized for ease of discourse and clarity, not efficiency of implementation.

The key assumption made by our hinted collector algorithm is that the sets of hinted objects and unhinted objects can be efficiently tracked and objects within those sets can be cheaply iterated. We discuss one means of achieving this in Section 2.1. As with a standard collector, we associate a mark bit with each object that is set if that object is assumed to be live. The collector starts with all objects unmarked. When the algorithm completes, any reachable object will have been marked. We note that the algorithm may also mark objects unreachable in a standard traversal due to missing hints.

Phase 1 In the first phase, any unhinted objects are marked. Hinted objects in the root set are also marked. A key observation is that there are no restrictions on the iteration order of unhinted objects; iteration can be done in any serial or parallel order.

Phase 2 In the second phase, all outbound references from unhinted objects are traced and their target marked. The net effect of this phase is to mark any hinted object which is directly reachable from an unhinted

```
phase 1: /* unhinted objects and roots */
for o in unhinted objects:
  mark(o)
mark all roots

phase 2: /* hinted, directly reach. from unhinted */
exact = (are all roots unhinted?)
for o in unhinted objects:
  for e in o.outbound_edges:
    if not e.target.marked:
      mark e.target
      exact = false

if exact:
  exit with marking done

phase 3: /* hinted, reachable from hinted only */
for o in hinted objects:
  if o.marked:
    push(o)
while( mark stack not empty ):
    o = pop
    for e in o.outbound_edges:
      if not e.target.marked:
         mark e.target
         push e.target
```

Figure 1.2: Hinted collection algorithm discussed in Section 1.3.

object. If no new objects are marked during this phase, we have established that no objects were marked inaccurately and do not need to execute phase 3 at all. Note that this is a sufficient, but not necessary, condition; missing hints may trigger the execution of phase 3 even when all given hints were accurate.

As with the first phase, the iteration order is completely arbitrary. Care must be taken to assure that the marking of an object is idempotent, but once this is true, marking can occur in any order. The structure and connectivity of the unhinted subgraph is irrelevant.

Phase 3 In the third phase, any objects reachable from the previously marked hinted objects are marked. The purpose is to prevent objects which were inaccurately hinted, but are only reachable from other inaccurately hinted objects, from being incorrectly reclaimed. This phase may also mark accurately hinted objects due to the existing of missing hints.

As in a standard collector, a stack-based depth-first-search algorithm is used. A mark stack holds references to objects which have been marked, but not yet scanned for outbound references. For now, we will assume an infinite mark stack to avoid overflow issues; we will return to this in Section 2.1. The first step is to scan the set of hinted objects and push any that have been marked onto the stack. A standard traversal is then initiated. When processing an object from the stack, references to objects which have already been marked are ignored since they have either already been traced, or are currently on the mark stack. Once the depth-first-search has terminated, any reachable object must by definition be marked.

	Sequential	Parallel
Phase 1	$\lvert V_{unhinted}\rvert$	1
Phase 2	$\lvert E_{unhinted}\rvert$	1
Phase 3 (Exact)	n/a	n/a
Phase 3 (General)	$\lvert V_{hinted}\rvert + (\lvert V_{hinted}\rvert + \lvert E_{hinted}\rvert)$	D_{hinted}
Overall (Exact)	$\lvert V_{reachable}\rvert + \lvert E_{reachable}\rvert$	1
Overall (General)	$\lvert V\rvert + \lvert E\rvert + \lvert V_{hinted}\rvert$	D_{hinted}
Standard Traversal	$\lvert V_{reachable}\rvert + \lvert E_{reachable}\rvert$	D

Table 1.1: Summary of asymptotic complexity results for hinted collector mark algorithm using an ideal Parallel Random Access Machine (PRAM). The "Exact" results are for the case where all unreachable objects are hinted, and all reachable objects are unhinted. The "General" results allow both inaccurate and missing hints.

Asymptotic Scalability

Extending the definitions introduced previously, we introduce terms V_{hinted}, and $V_{unhinted}$ with the expected meanings. E_{hinted}, and $E_{unhinted}$ are the set of edges leaving each vertex set respectively. Similarly, we will use the terms $V_{missing}$ and $E_{missing}$ for the set of objects with missing hints and the set of edges leaving such objects. We note that all of these terms are usually incomparable with the terms for reachability.

Extending this, the set of objects with inaccurate hints is merely $V_{inaccurate} \equiv V_{hinted} \cap V_{reachable}$. In the case where all hints are accurate then $V_{hinted} \subseteq V_{unreachable}$ and $V_{inaccurate} = \emptyset$. We say that the set of hints is mostly accurate if $|V_{inaccurate}| << |V_{reachable}|$. We expect that in the common case, hints will be mostly accurate and few hints will be missing.

We'll begin by summarizing the results for the entire algorithm, and then exploring the analysis of each phase in isolation. Where appropriate, we will note both general results and constrained results for when hints are exact. A summary of these results in Table 1.1.

Summary Taking all three phases together, we are left with a sequential complexity of $O(|V| + |E| + |V_{hinted}|)$. When the hints are exact, this reduces to $O(|V_{reachable}| + |E_{reachable}|)$ since phase 3 does not execute and, by assumption, the unhinted and reachable sets are equivalent. Worth noting, this is exactly the complexity of the standard traversal algorithm.

When it comes to the parallel complexity, the hinted collector comes out ahead. In the general form, the asymptotic complexity is $O(D_{hinted})$, where D_{hinted} is the depth of the hinted subgraph. In the case where the hints are exact, this drops to a mere $O(1)$ since traversal of the hinted subgraph is not required. With an infinite number of processors, the running time of a hinted collector is only influenced by the subset of the graph which consists of *inaccurately hinted objects* or *hinted objects reachable from objects with missing hints*. This subset is in turn bounded by the set of hinted objects.

Phase 1 Phase 1 has a serial complexity of $O(|V_{unhinted}| + |V_{roots}|)$ since it must consider every unhinted vertex and every root. Since there is no aliasing of mark bits, the parallel complexity scales inversely with P (the number of processors). As P goes to infinity, the complexity drops to $O(1)$.

We assume that V_{roots} is small compared to $V_{unhinted}$. For clarity of presentation, we omit this term from the remainder of our results. In the worst case, V_{roots} is trivially bounded by $V_{reachable}$ and would not change the overall sequential results in a significant way. The parallel results are entirely unaffected by the presence of this term.

Phase 2 Phase 2 has a serial complexity of $O(|E_{unhinted}|)$ since we must explore every outbound edge from every unhinted object exactly once. In the limit, the parallel complexity is again $O(1)$.

Note that unlike Phase 1, there may be aliasing when we add parallel processors. As such, it is likely that the scalability of phase 2 would practically be limited by the contention on updates to mark bits by multiple processors. With a straight-forward implementation, this would be linear in the maximum in-degree ($O(\max_{V \in V} indegree(v))$). In principal, this dependence could be reduced from linear to logarithmic in the number of inbound edges using a parallel reduction operation. We doubt this would result in a practical marking algorithm and have not explored the topic.

Phase 3 Phase 3 has a sequential complexity of $O(|V_{hinted}| + (|V_{hinted}| + |E_{hinted}|))$. The first term is influenced by the number of hinted objects. The second is driven by the need to perform a graph traversal of the reachable hinted objects. The parallel complexity is $O(D_{hinted})$ as P goes to infinity.

Naively, the portion of the graph explored is bounded by $V_{reachable}(V_{missing}) \cup V_{inaccurate}$ where the first term is the objects reachable from objects with missing hints and the second is the set of objects inaccurate hinted. While at first glance it seems like a missing hint could force an exploration of the entire unreachable graph, this is not the case. Any portion of the unreachable graph which was not also hinted has already been marked. As a result, the subgraph explored is bounded by $(V_{reachable}(V_{missing}) - V_{unhinted}) \cup V_{inaccurate}$ which we know to be a subset of V_{hinted}.

As a reminder, if the set of hints is exact phase 3 does not execute. If the given hints were only nearly exact, the second term should be small, leaving $O(|V_{hinted}|)$ and $O(1)$ as the dominant terms for the sequential and parallel cases respectively. Our expectation is that the set of hints is often nearly exact.

Key Observations

It is useful to discuss the impact of hint accuracy on the proposed algorithm. On one extreme, a collection for which all objects are hinted reduces to a standard traversal based collector. Phase 1 and 2 become trivial, and only Phase 3 does useful work. The scanning for marked objects at the beginning of Phase 3 is mostly wasted, but since the roots are marked, the traversal will eventually mark all reachable objects. On the other extreme, if no objects are hinted, then the collector reduces to a pair of scans over the set of objects which mark every object and reclaim nothing.

Given a mixture of accurate (unreachable) and inaccurate (reachable) hints, we claim that all reachable nodes will be marked after the algorithm executes. This follows directly from our earlier observation that increasing the root set for a reachability traversal can only grow the set of reachable objects, not shrink it. Phase one ensures that any unhinted object is marked and that any hinted root is marked. Phase 2 and 3 can be be abstracted as a reachability traversal - with a slightly strange ordering - starting from the union of the actual roots and the unhinted objects. Since this union must by definition be a superset of the root set, we know that the set of objects reachable from this initial set must be a superset of the set reachable from only the roots. Another way of reasoning about this is that each hinted object reachable from the roots must be reachable (without passing through a marked object) from at least one object which would be marked as a result of phases 2 and 3 combined. By the same arguments, the presence of missing hints can not cause an object to be incorrectly reclaimed. To summarize this point, it suffices to say that missing and inaccurate hints are safe for correctness, if not necessarily performance.

Chapter 2

A Serial Collector

Taking the algorithm from the previous section, we implemented two versions of a hinted collector for C and C++ programs. The version described in this chapter is a serial implementation. We will extend this to a parallel implementation in the following chapter. The source code for both collectors – including all microbenchmarks, test drivers and most data files – is available at the following url: `https://github.com/preames/hinted-collection`.

By replacing the normal "free" routine with one which simply records the deallocation hint for later processing, we are able to evaluate the effectiveness of a hinted collector on real programs with large numbers of deallocation hints.

2.1 Design & Implementation

Moving from an abstract collector to a concrete one, there are a few questions we need to address:

- How does one efficiently record membership in the sets of hinted and unhinted objects?
- How does one handle overflow during Phase 3 of the algorithm?
- What are the practical bottlenecks and what key optimizations are relevant?

As before, our discussion will focus nearly exclusively on the mark phase of the collector. The implementation uses lazy-sweeping during allocation to reclaim objects unmarked by the collector. This is not a point of difference with a conventional collector.

Platform

We have modified the Boehm-Demers-Weiser [12] conservative garbage collector for C/C++. The Boehm-Demers-Weiser collector provides a well tuned implementation of a sequential mark-sweep stop-the-world collector.

The Boehm-Demers-Weiser collector provides a free-list style malloc/free allocator. When acting as a pure garbage collector, calls to free are ignored by the allocator. There are a number of predefined size classes. Each size class has an associated set of heap blocks (hblks) which store objects of that size, and a free list threaded through the free objects in those pages. Information about the contents of the hblk are stored in a header (hblkhdr) which is allocated in scratch space. Not every hblk has its own header; when larger blocks of memory are needed, multiple hblks are coalesced with only a single header associated with all of them.

Set Membership Metadata

By default, all objects are assumed to be unhinted. To record deallocation hints, we modified the allocator to store a boolean flag in the hblkhdr to indicate some object in the hblks associated with that header has been hinted. The advantages of the chosen approach are:

- Adding the flag did not require modifying the heap layout. Room was already available in the hblkhdr for additional flags.
- Iterating through objects in a given set can be done cheaply by iterating through a preexisting table of hblkhdrs.
- The many-to-one nature of the flag increases the odds that the flag will be in cache if checked for many objects at once. This will be useful in implementing an edge-filtering optimization.

The downside of the metadata storage scheme is that - since headers are shared by many objects - when one object is hinted by the user, many objects are actually hinted. A likely effect is that many of those extra objects were inaccurately hinted - resulting in slightly longer pause times. We will investigate the impact of this, and discuss possible alternative implementations in Section 4.3.

Mark Stack Overflow

Before introducing the concrete implementation, we need to address a slightly more fundamental issue with the algorithm presented previously. In that discussion, we made the assumption that the mark stack - used during Phase 3 depth first traversal of hinted objects - was infinite. In practice, the mark stack is finite and could potentially overflow.

To understand why overflow can occur, picture a heap graph where the entire space is consumed by an inaccurately hinted long linked list. Unless there is room in the mark stack for every object in the program, the mark stack must overflow[1]. We must preserve correctness in this case without reserving an excessively large amount of memory for the mark stack in the normal case.

The Boehm-Demers-Weiser mark implementation includes a mechanism to restart a general heap mark. Every time an object is to be pushed onto the mark stack, it is marked first. If the mark stack overflows, excess items are dropped and the traversal continues. Once the current mark stack empties - to make as much progress as possible - the mark stack is expanded[2], and the heap is scanned for marked objects. Any unmarked objects directly reachable from a marked object is pushed on the mark stack. The mark stack may overflow again, but some forward progress must be made during the emptying of the mark stack. The process will eventually terminate since there are only a finite number of reachable objects. As should be clear, this is a fairly complex process, and in the worst case, could result in the heap being scanned approximately $O(\log(|V|))$ times for a total runtime of $O(|E| * \log(|V|))$.

We must adapt this handling to our own marking algorithm. Neither Phase 1 or 2 use the mark stack in any way. Phase 3, on the other hand, is an adaption of the classic mark algorithm. We extend it with overflow handling in a similar way to the Boehm-Demers-Weiser mark implementation. For the full version of the modified phase 3, see Figure 2.1.

To avoid needing a mark stack large enough to hold every hinted object, we interrupt the scan occasionally[3] to empty the mark stack. If despite this measure, the mark stack overflows, any objects which

---

[1] As described, the mark stack could consume up to 50% of total memory since every object could contain only a single "next" field and be inaccurately hinted. During the depth-first traversal, every object would be on the mark stack at once. This particular case could easily be avoided by optimizing the traversal to remove single reference objects from the mark stack before exploring their children, but similar cases could be easily constructed for unbalanced trees of arbitrary degree.

[2] We note that expanding the mark stack is not required for correctness. In a low memory situation, the mark stack might not be expanded and more iterations would be required.

[3] We currently perform the modified traversal when the mark stack is 50% full. The performance of the collector is largely insensitive to this parameter.

```
func modified_dfs:
  while( mark stack not empty ):
    o = pop
    for e in o.outbound_edges:
      if not e.target.marked:
        mark e.target
        if not mark_stack full:
          push e.target
        else:
          mark_stack_too_small = true;

func mark_hinted_objects:
  for o in hinted objects:
    if o.marked:
      push(o)
      occasionally modified_dfs();
  modified_dfs();

phase 3:
mark_stack_too_small = false
mark_hinted_objects()
while( mark_stack_too_small ):
  resize_mark_stack( 2 * mark_stack_size )
  mark_stack_too_small = false
  mark_hinted_objects()
```

Figure 2.1: Phase 3 of the hinted collector with mark stack overflow protection added. The algorithm is otherwise unchanged.

would have otherwise been added are simply discarded and an overflow flag is set. The traversal continues - potentially discarding many objects - until it is once again empty. We then increase the size of the mark stack, and repeat the entire process. Since every object is marked before being placed on the mark stack, we are guaranteed to make progress; since there are a finite number of hinted objects, we will eventually terminate.

Since only edges from hinted objects must be considered when repopulating the mark stack, our fully functional algorithm can execute in $O(|E_{hinted}| * \log(|V_{hinted}|))$ - i.e. potentially significantly less than the standard algorithm.

Worth highlighting is that the mark stack can only overflow if a large number of hinted objects are reachable from those unhinted. By design, this should be a rare case. Long chains occurring solely within the unhinted (i.e. live) section of the heap graph are never traversed and can not trigger overflow. Even in theory, we would expect such a case to be rare given the imprecise hinting implied by our current approach to metadata storage. Assuming that the probability of any particular object being imprecisely hinted is approximately uniformly random, the likelihood of finding a long chains solely within a imprecisely hinted region falls off exponentially with the length of the chain. We believe this to be a reasonable assumption based on knowledge of typical allocation patterns and allocator strategies, but can not formally justify it. We have no data to justify this assumption or refute it.

```
for hdr in hblkhdrs:
  if not hdr.hinted:
    set_all_mark_bits(hdr)
```

Figure 2.2: Phase 1 of the concrete collector.

Practicalities

In the previous subsections, we have addressed the two key differences between the idealized algorithm and the form we can actually implement. Next, we describe the actual implementation - which has some minor differences from the ideal algorithm - and describe key optimizations which reduce the absolute runtime without changing the asymptotic complexity.

Phase 1 Phase one is the simplest to implement. Conceptually, we simply need to walk the set of hblkhdrs, select those with the deallocation hint flag not set, iterate over each object they contain, and mark all of them. The naive implementation is correct, but the inner-most loop adds about 20% to the overall runtime. Instead, we can take advantage of the layout of mark bits - which are stored in a contiguous bitmask in the hblkhdr - to set all the mark bits for a given hblkhdr at once with a small handful of assignments. Pseudocode for phase 1 is listed in Figure 2.2.

In the implementation, the marking of root objects is integrated into the modified depth first search of phase 3. The only reason for this is that it allows us to use a manually unrolled and pipelined routine to process items on the mark stack for marking all objects in the root set quickly. There is no intrinsic reason this code could not be duplicated and executed earlier in the process.

Phase 2 The starting point for optimization is a fairly simple loop that iterates over each unhinted object and marks any unmarked object it references. For simplicity, we re-purposed the mark stack and its supporting routines. This was desirable since the task of filtering the word-sized values to identify potential outbound references - required by the fact we are targeting an type-unsafe language - is fairly complex and error prone. The unoptimized version of the code loops through every object in every hinted hblk, pushes all the objects onto the mark stack (without marking them again), and then calls a modified stack processing routine which does not push outbound edges onto the mark stack. This involves a lot of redundant memory traffic on the mark stack for no good reason (in the serial collector), but has the benefit of being easy to audit for correctness and simplifies implementing the parallel collector.

While functionally correct, this version is not sufficiently fast to be competitive with the well tuned tracing collector baseline. Phase two consumes the majority of runtime for the case study and microbenchmarks we have investigated. We therefor incorporated the following two key optimizations:

- **Edge-Filtering** – Before checking to see whether an object pointed to by a reference is marked, we check to see whether the target is an unhinted object. If so, the target must have been marked in phase 1 and we avoid dereferencing the pointer. Since it is expected that most hinted objects are unreachable, if we can filter edges to objects already known to be marked, we can ignore most outbound edges. The set membership check reduces to a read of a flag in the hblkhdr. The header check itself is relatively cheap since, in practice, there are few enough header blocks that most of them fit in cache at any one time. By performing the additional check, we can shave about 10% of runtime from the full hinted collection. In Section 2.2, we discuss alternate designs considered.
- **Object Combining** – Instead of processing individual objects, we combine all objects in a hblk into a single contiguous range and process all potential references together. Since we have to be conservative about what might possibly be an outbound reference anyway, this combining of objects allows us to

```
func mark_targets(begin, end):
  while begin < end:
    if is_reference(begin):
      if is_hinted(begin):
        if not begin.marked:
          mark(begin)
          exact = false
    begin += 1 word

phase 2:
exact = true
for hdr in hblkhdrs:
  if combinable(hdr):
    mark_targets(hdr.hblk_start, hdr.hblk_end)
  else:
    for o in hdr.objects:
      edges = o.outbound_edges
      mark_targets(edges.start, edges.end)

if exact:
  exit with marking done
```

Figure 2.3: Phase 2 with Edge-Filtering & Object-Coalescing.

completely forgo the outer loop. Depending on the benchmark, we see as much as a 30% improvement from this change alone[4].

We believe this to be from a mixture of low level code quality improvements (i.e. increased instruction level parallelism and overlapping memory loads from a hand unrolled and prefetched loop) and decreased memory traffic on the mark stack.

The combined algorithm is shown in Figure 2.3. One point that is important to mention is that we do not implement the check for exactly correct hints shown. The key reason for this check is to avoid executing phase 3 - which affects the theoretical results slightly, but does not have a significant impact on the running time of the practical algorithm. This might be worthwhile to implement at some point, it just has not been done yet.

Phase 3 The implementation of Phase 3 of the algorithm is a mostly direct translation of the version discussed in Section 2.1. When tuning Phase 3, there are two major goals. First, we want the case where very few hinted objects are marked - hopefully the common case - to mostly reduce to a single pass over each hinted object. Second, we want the graph traversal to be as fast as possible when it is forced to execute.

For the scan of hinted objects, we take a similar approach to the unoptimized algorithm described for Phase 2, but with the difference that marked objects are pushed onto the mark stack rather than directly scanned. If we find a marked object, then we must follow any outbound references to ensure that any reachable hinted objects are marked. Given the algorithm's similarity to a standard reachability collector, we were able to reuse many of the components of the Boehm-Demers-Weiser collector.

---

[4] This optimization is applicable to most (but not all) object classes. The unsupported object classes fall back to the correct, but slow version described above. There is no fundamental reason the other object classes could not be supported.

In principle, we could use a variant of the object-combining optimization to reduce traffic on the mark stack and potentially improve scan performance, but we have not implemented this. We have implemented a version of the edge-filtering optimization.

Sweep Once Phase 3 has terminated, any reachable object is known to be marked. This is the same invariant ensured by the mark phase of a mark-sweep collector. As a result, we are able to reuse the sweep phase from the baseline collector without modification. The Boehm-Demers-Weiser collector uses a lazy sweeping strategy. Control immediately returns to the mutator after marking and memory is incrementally reclaimed on demand during allocation.

2.2 Evaluation

In Table 2.1, we present results from a set of microbenchmarks written to highlight the strengths and weaknesses of hinted collection. As a reminder, the implementation we evaluate in this section is a serial collector. A brief discussion of some preliminary results with the parallel collector can be found in Section 3.2.

Methodology & Test Platform

All microbenchmarks were written in C++, but use only malloc/free for memory management. Each benchmark was written in a style to allow manual memory deallocation, but with the free call redirected to the hinted collector library as a deallocation hint. When freeing data structures, we chose not to break internal references; this ensures that the hinted collector results pay the full possible penalty when inaccurate hints are given for the relevant benchmarks. All benchmarks were compiled with GCC 4.6.3 with -O3 specified. We also compiled and ran them with Clang 3.1, but do not report these results since they were essentially identical[5].

The times reported in this section are the pause times of individual collections. We do not report overall runtimes or mutator utilization ratios. Each benchmark is run 20 times, and the arithmetic mean value is reported. Before each benchmark run, we fragment the relevant size classes by allocating a large number of objects with high average turnover, but randomly chosen lifetimes. We disable garbage collection while building the heap structures and hinting any objects necessary. To prevent accumulation of fragmentation, we use a fork/join wrapper around each iteration. As a result, every data point for a particular benchmark shares the same starting heap state. After each collection, we eagerly reclaim all available memory. This enables us to report on memory reclaimed by each collector.

All results except the scalability experiment were run on a Lenovo Thinkpad with a Intel(R) Core(TM) i7-2620M CPU which has 2 x86_64 cores, each 2-way SMT. The memory hierarchy is organized as a 32 KB L1, 256 KB L2, and 4 MB L3 cache, backed by 8GB of DDR3-1333 memory. There are two memory channels with a maximum bandwidth of 21.3 GB/s.

Overall Performance

The first two sets of benchmarks illustrate the fundamental performance trade-off of a hinted collector. The hinted collector is able to outperform a tracing collector when the entire heap is live and unhinted (the common case). In these microbenchmarks, the tracing collector wins in all other cases; this is caused by the fact that we truncate the data structures at the root, leaving no tracing work.

---

[5] To reproduce our results, we strongly suggest starting with our publicly available source code. The benchmarks exploit undefined behavior in C++, and compilers are extremely good at breaking such programs. The benchmarks are carefully engineered to get correct results with the versions of the compilers used. We assume the compiler can not identify dead stores across procedure boundaries and that no-inlining directives are respected.

benchmark	heap size	gc	hintgc	**speedup**
Linked List (Dead, Hinted)	31.8 MB	1.45	2.00	**0.72**
Linked List (Dead, Unhinted)	31.8 MB	1.40	7.75	**0.18**
Linked List (Live, Hinted)	31.8 MB	11.95	12.10	**0.99**
Linked List (Live, Unhinted)	31.8 MB	12.10	7.40	**1.64**
Fan In (Dead, Hinted)	23.7 MB	0.00	0.00	**n/a**
Fan In (Live, Hinted)	23.7 MB	12.85	12.90	**1.00**
Fan In (Live, Unhinted)	23.7 MB	12.50	8.25	**1.52**
2560 x 1k element LL	88.6 MB	31.70	19.80	**1.60**
256 x 10k element LL	88.6 MB	30.95	18.90	**1.64**
1/3 Cleanup	184.6 MB	50.10	31.95	**1.57**
Deep Turnover	56.6 MB	20.50	12.30	**1.67**
Unbalanced (Live)	744.6 MB	252.85	176.50	**1.43**
Unbalanced (Partly Dead)	744.6 MB	246.30	175.25	**1.41**

Table 2.1: Average mark times (in ms) for the serial collectors. “gc” is the baseline tracing collector. “hintgc” is the hinted collector as described in the text with header edge-filtering and object combining. “speedup” shows the improvement of the hinted collector over the tracing collector.

- **Linked List** - If the heap contains a long list of objects which is reachable from the root, then any traversal to establish reachability must traverse every object in turn. The hinted collector is able to avoid this long chain of dependent loads.
- **Fan In** - A heap graph with a single root node with edges to $N >> P$ vertices, all of which have a single reference to a final vertex. While this structure may seem contrived, is actually fairly common. It arises frequently from objects which implement copy-on-write semantics; at least one platform we are aware of uses this to optimize the creation of strings.

The remaining benchmarks highlight cases where a hinted collector has a strong advantage. The first two are useful for understanding properties of the two collectors, while the remaining three highlight behavior relevant in real world programs.

- The **2560 x 1k element LL** and **256 x 10k element LL** benchmarks consist of a set of linked lists reachable directly from the root set. We vary the number of linked lists and the number of elements to produce two different heap configurations with different heap structures. The entire heap is live. It is interesting to note that the tracing collector comparatively performs slightly better with shorter but more numerous linked lists. This is exactly what we would expect.
- **1/3 Cleanup** highlights the performance of the collectors when only a portion of the heap becomes unreachable with non-trivial data structures remaining. To illustrate, we allocated six one-million element linked lists and then deallocated two of them. As expected, the hinted collector outperforms the tracing collector by a significant margin.
- In **Deep Turnover** a relatively small portion of the heap is being deallocated. However, that portion is deep inside a long linked list. (We choose to delete 1000 elements off the end of a 1 million element linked list.) This case was chosen to reflect a common pattern in real programs where most of the heap stays around for a long period with small chunks of it being recycled.
- In the two **Unbalanced** results, we see a benchmark with most live space consumed by a collection of 256 depth-6 octtrees. A similar number of linked lists are allocated, but not retained. The first experiment is with all the lists held live, the second is when they are allowed to die. Interestingly, the percentage difference between the two rows is much higher for the standard traversal than the hinted collector. This highlights the stability of our approach with regards to heap shape.

benchmark	heap size	base	no-oc	no-ef	header	range	both
Linked List (Dead, Hinted)	31.8 MB	10.10	0.15	0.30	2.00	2.05	0.15
Linked List (Dead, Unhinted)	31.8 MB	10.55	10.30	8.35	7.75	7.50	7.60
Linked List (Live, Hinted)	31.8 MB	12.00	11.70	12.30	12.10	12.25	12.30
Linked List (Live, Unhinted)	31.8 MB	11.10	10.70	8.85	7.40	7.25	7.05
Fan In (Dead, Hinted)	23.7 MB	0.00	0.00	0.00	0.00	0.00	0.10
Fan In (Live, Hinted)	23.7 MB	13.25	12.35	13.30	12.90	12.50	12.15
Fan In (Live, Unhinted)	23.7 MB	14.60	9.35	11.10	8.25	6.00	6.60
2560 x 1k element LL	88.6 MB	28.60	27.70	21.75	19.80	18.50	18.05
256 x 10k element LL	88.6 MB	29.00	27.35	21.65	18.90	18.35	18.50
1/3 Cleanup	184.6 MB	55.80	44.60	37.15	31.95	35.45	32.55
Deep Turnover	56.6 MB	16.30	15.70	16.55	12.30	10.00	10.80
Unbalanced (Live)	744.6 MB	240.05	223.95	192.15	176.50	174.85	174.70
Unbalanced (Partly Dead)	744.6 MB	237.30	223.40	187.30	175.25	175.90	174.90

Table 2.2: Average mark times (in ms) for variants of the hinted collector. "base" is a variant with neither edge-filtering or object-combining and illustrates well the importance of the two optimizations. "no-oc" is a variant without object combining, but with header edge-filtering. "no-ef", "header", "range", and "both" are variants of the hinted collector with no edge-filtering, header-filtering, range-filtering, and both header and range filtering respectively; all three use object-combining. The "header" configuration is the same described elsewhere in the thesis.

The key reason the hinted collector outperforms a standard collector on these benchmarks is its ability to explore live objects and edges in any order. There are two key benefits that result:

- As previously discussed, not having to follow edges between live nodes in order of discovery prevents the hinted collector from being sensitive to the depth of the live graph. This would mainly benefit a parallel collector, but we see some benefit in our serial collector due to instruction level parallelism and instruction reordering by the hardware.
- By allowing the collector to explore the live edges in any order, the hinted collector converts a series of dependent loads - which is primarily limited by the latency of a memory access - to a set of parallel loads - which is limited by the bandwidth of the memory system. While not reflected in the asymptotic results, this is probably of more practical importance.

We do not directly report the amount of memory reclaimed for each collector on each benchmark. We have manually inspected each benchmark and confirmed that all hinted data is reclaimed for these examples. When unhinted, a small amount of memory (80k) is still reported as being reclaimed, but this is mostly independent of the benchmark. We believe this amount to be from internal approximation in the collector framework; the amount does not vary between collector types.

Edge-Filtering and Object-Combining

We investigated the impact of the edge-filtering and object-combining optimizations - introduced in Section 2.1 - by running each of the micro benchmarks against versions of the collector with each optimization disabled. We additionally explored an alternate edge-filtering implementation based on tracking a high and low water mark for hinted pointers (henceforth range-based)[6]. If during the scan a pointer outside this range was found, it clearly must be unhinted. Next. we considered the combination of both edge-filtering implementations, with the range check executing first. Finally, we evaluated a version with neither edge-filtering or object-combining.

---

[6] This implementation was inspired by a similar range base filtering optimization used by the Boehm-Demers-Weiser implementation to quickly discard potential references which could not be actual pointers to objects.

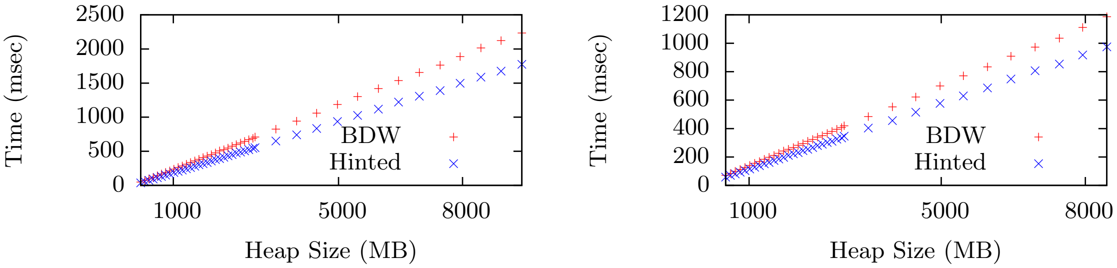

Figure 2.4: Scalability of hinted and tracing collectors given 10% (left) and 50% (right) deallocation rate over range 200 MB to 10GB of allocated data. Despite the difference in y-axis ranges, the trend is similar.

As can be seen from the results in Table 2.2, edge filtering is clearly advantageous, in some cases contributing a substantial improvement over the base algorithm. When comparing the different implementation options, it is clear the range-based check and the default header flag check both have their own advantages. The combination of the two (in the last column), appears to perform well across the board. Object-combining is clearly also profitable. Note that all the results presented elsewhere in this thesis use (only) the header implementation of edge-filtering and object-combining.

Scalability

We ran an additional experiment with a microbenchmark which allocated reasonably complex heap structures of an arbitrary size and then deallocated a specified amount[7]. In Figure 2.4, you can see the results of an experiment which varied the heap size from 200 MB to nearly 10 GB while deallocating a fixed percentage at each heap size. We varied the percentage of objects deallocated and present two representative graphs: one at 10% deallocation, and another at 50%.

The hinted collector outperforms the standard collector across the entire range with a roughly 20% reduction in pause time for a given heap graph. As shown, the absolute runtime of both collectors is heavily influenced by the total amount of memory which must be explored (i.e. is live). We can see this in two ways. First, as the allocated amount increases with the percentage deallocated fixed, so does the runtime. Second, when the fraction of the heap which is live decreases with the amount allocated held constant, so does runtime. We believe that the relative difference between the collectors can be attributed to the hinted collectors ability to better exploit memory bandwidth. We do not see the dependence on heap structure in this benchmark since the heap graph is fairly shallow, which allows both collectors to perform well.

These results are fairly insensitive to the percentage deallocated, but only up to around 90% where the much smaller heap explored by the tracing collector allows it to break even. Unlike the tracing collector, The hinted collector must inspect the mark state of every hinted objects even it doesn't scan them. This penalty is fairly minimal, but with a large fraction of the heap deallocated it can come to dominate. We note that while the exact thresholds are dependent on the structure of the benchmark, but the general pattern was observed across multiple workloads.

SPEC 2006

Full results from running a subset of the C programs in the SPEC 2006 benchmark suite can be found in Table 2.4 and 2.3. Table 2.4 focuses on the pause times observed, while Table 2.3 focuses on the amount of

---

[7] The heap structure allocated was essentially an extremely wide shallow tree (depth two) with a short linked list (length 50) hanging off each leaf. The amount allocated was varied by controlling the number of children at the root level. Nodes in the first level had a fixed degree of 500. All references to hinted objects are explicitly set to NULL ensuring their are no inaccurate hints.

benchmark	hinted	reclaimed	% rec	leaked	% leak
bzip	0 bytes	0 bytes	n/a	0 bytes	n/a
cactusADM	7.03 GB	7.00 GB	99.56	23.87 MB	0.34
calculix	64.08 GB	62.47 GB	97.50	1.14 GB	1.79
gobmk	2.89 GB	2.89 GB	99.97	1.76 MB	0.06
gromacs	14.01 MB	13.83 MB	98.74	621.12 KB	4.30
h264ref	3.79 GB	3.41 GB	90.12	284.11 MB	7.69
hmmr	8.09 GB	7.90 GB	97.61	227.58 MB	2.80
lbm	0 bytes	0 bytes	n/a	0 bytes	n/a
libquantum	2.63 GB	540.99 MB	20.58	67.46 MB	11.09
mcf	0 bytes	0 bytes	n/a	0 bytes	n/a
milc	260.67 GB	252.23 GB	96.76	3.10 GB	1.22
perlbench	104.36 GB	100.27 GB	96.08	15.18 GB	13.15
sjeng	0 bytes	0 bytes	n/a	0 bytes	n/a
sphinx3	48.73 GB	48.71 GB	99.96	101.25 MB	0.21

Table 2.3: Summary of space reclamation across all collections for each SPEC benchmark. “hinted” is the total number of bytes directly hinted by the application. “reclaimed” is the amount of space reclaimed by the hinted collector. “% rec” is the percentage of hinted data reclaimed; due to collateral hinting, this can be greater than 1.0. “leaked” is the total bytes reclaimed by the tracing collector. “% leaked” is the leaked column divided by the total memory available for collection (leaked + reclaimed).

memory reclaimed by the hinted collector.

To summarize the results, out of the 10 benchmarks with any deallocation captured by a collection cycle, 9 reclaim 90% or more of the memory hinted as free across the run. The 10th (`libquantum`) reclaims only 20% of the hinted memory, but the tracing collector reclaims only a small amount more. It appears that `libquantum` is retaining a reference to dead data past the last collection cycle. We will discuss this benchmark more in Section 3.2 since this behavior limits the performance of both parallel collectors.

The benchmark `perlbench` has the largest amount of memory leaked by the hinted collector at 13.15%. Despite this, the reclaim rate for the hinted collector is actually quite high at 96%. The fact that these numbers sum to more than 100% implies that the original benchmark had a memory leak when run with manual deallocation. Most of the other benchmarks are in the 1-5% range. We suspect, this leakage is most likely due to hinted objects not actually becoming unreachable until after the next collection.

Pause time results are mixed with several benchmarks taking slightly longer with the hinted collector than the standard collector. A few benchmarks (`perlbench`, `milc`, `cactusADM`) show significant pause time improvement; `perlbench` improves by nearly 40%.

Methodology Each benchmark was run three times using its reference input set. Statistics were computed across all observed collections for each benchmark. `gcc` and `wrf` were excluded since they failed to complete when run with the collector inserted via LD_PRELOAD. `gcc` is known to use xrealloc which is not currently supported. The cause of failure for `wrf` has not been investigated.

The tracing collector was run immediately after the hinted collector completed. This was done to ensure both collectors encounter the same heap graph; it would be undesirable for one collector to encounter a deep narrow heap graph (for example) which the other collector misses due to a difference in collection timing between runs. Unfortunately this choice has, in practice, the effect of minorly understating the tracing collector’s runtime. There is some noise in the amount of data reclaimed, though we do not believe it to be significant for the overall results. We have seen up to a few hundred KB of space per collection falsely accounted due to marking of collector structures such as free lists. The most likely effect is to overstate the leaked amount slightly.

		Hinted Collector				Tracing Collector			
benchmark	count	min	max	mean	median	min	max	mean	median
bzip	36	6.0	118.0	57.03	57.0	6.0	116.0	55.58	56.0
cactusADM	33	0.0	183.0	91.52	74.0	0.0	194.0	94.76	72.0
calculix	786	0.0	77.0	12.47	0.0	0.0	62.0	11.51	0.0
gobmk	156	1.0	21.0	10.46	11.0	1.0	20.0	10.88	11.0
gromacs	15	0.0	3.0	0.87	0.0	0.0	3.0	1.20	1.0
h264ref	134	0.0	25.0	7.38	5.0	0.0	19.0	7.10	5.0
hmmr	1025	0.0	15.0	0.95	0.0	0.0	6.0	0.75	0.0
lbm	3	42.0	42.0	42.00	42.0	41.0	41.0	41.00	41.0
libquantum	24	6.0	902.0	121.62	18.0	6.0	910.0	123.33	20.0
mcf	3	1.0	1.0	1.00	1.0	1.0	1.0	1.00	1.0
milc	181	64.0	562.0	413.34	422.0	62.0	607.0	450.71	462.0
perlbench	545	0.0	291.0	101.29	106.0	0.0	482.0	122.26	134.0
sjeng	6	12.0	24.0	18.00	18.0	12.0	23.0	17.50	17.5
sphinx3	1538	0.0	24.0	11.02	11.0	0.0	16.0	12.47	13.0

Table 2.4: Statistical summary of pause times observed for each SPEC benchmark. All times are in msec. The first set of columns are from the hinted collector; the second set are a standard collector run immediately after the hinted collector completes. We note that this slightly understates the standard collectors runtime since some garbage has already been reclaimed. Interesting highlights include the sharp drop in maximum pause time for `perlbench`, and improvement in mean & median pause times for `milc`.

Case Study

To highlight the possible impact of hinted collection, we ran a case study with the Clang/LLVM compiler toolchain. We used instrumented versions of Clang 3.1 and the GNU gold linker 1.11 to build Clang itself from source. We instrument the programs to keep track of the amount of data hinted, perform a hinted collection, and then immediately perform a traditional collection.

As can be seen in the table below, the hinted collector reduced the maximum pause time observed by 12% and 99th percentile pause by 18% (across 34861 unique collections). We consider this a strong result. We note that both collectors encountered mark stack overflows during some of the collections; as a result, the reduced overflow cost of the hinted collector was advantageous. Across all the runs, the hinted collector was able to reclaim 95% of all memory hinted, with 5% of memory leaked.

version	min	max	mean	median	95th	99th
Hinted	0.0	490.0	20.42	10.0	70.0	140.0
Tracing	0.0	560.0	24.53	10.0	90.0	170.0

Chapter 3

A Parallel Collector

In addition to the serial implementation presented previously, we have completed a parallel implementation of the hinted collection algorithm. The results presented in this section illustrate that hinted collection is performance competitive with a standard tracing collector on a four-core machine.

3.1 Design & Implementation

As highlighted in the discussion of our serial implementation, phases 1 & 2 consume the majority of the collector time in the common case. Given this, we choose to prioritize the parallelization of these phases. We have not invested in parallelizing phase 3 explicitly, but since some of the traversal code is reused from the baseline collector, we see some parallel speedup in that phase as well. After discussing the implementation of the core algorithm, we discuss an optimization which helps to reduce parallel overhead for some collections and identify a few code changes outside for core mark algorithm that turned out to be critical for performance.

Mark Implementation

We choose to use a batch-synchronous style of parallelism where each phase of the algorithm is fully parallelized, but each phase completes fully before the next phase begins. To parallelize each phase, we adapted the load balancing approach used by the baseline parallel collector. It is worth noting that the barrier between phases 1 and 2 is only necessary if the header edge-filtering optimization is not being used. Otherwise, the execution of these two phase can be overlapped - potentially reducing synchronization overhead. The barrier between phases 2 and 3 is never necessary for correctness, but is likely best retained for performance. We have evaluated neither of these options and defer doing so to future work.

Background In the baseline collector, the global mark stack becomes a mark queue, and each marker thread has its own local mark stack. Ideally, most marking activity uses only the thread-local mark stack. The only cases where the marker threads interact with the global mark queue are to push work to the global queue when either the local mark stack is nearly full or the global queue is empty, and to pull work from the global queue when the local mark stack is empty. Additions to the global mark queue are performed using a single global lock. This ensures correctness, but at the potential cost of some performance.

Pulling work from the global mark queue is assumed to be a much more frequent operation, so a non-blocking read scheme is used. The consequence of this scheme is that the mark queue never shrinks – though the actively considered part of it does – implying that space on the global mark queue can not be reused. Additionally, since items in the global queue can be copied to multiple local stacks, every work item in the global queue must be idempotent - i.e. executing it one or more times must give an equivalent result. We experimented with a locking based variant and didn't find the lock free access to the global queue to be significant for the performance of the hinted collector in the cases we examined; it is significant for the traversal collector in some cases. Our investigation of this trade-off was by no means exhaustive.

To support efficient lookup of hblk headers from arbitrary pointers, the baseline collector includes a data structure (either a tree or hash table depending on configuration) which includes at its lowest level a doubly linked list of arrays of hblk descriptors. From this linked list, one can access every active in-use hblk managed by the collector. The entries in this list are called bottom_index structures.

The core marking loop used by both the baseline collector and the hinted collector supports four distinct marking tasks. The simplest and most common is simply a range of addresses to be scanned - represented by a start pointer and a length. The other three are largely irrelevant for our purposes except that one - GC_DS_PROC (which can be used by a developer to provide custom mark procedures for specific object types) is entirely unused in any C program which has not been specifically modified for use with the collector library. As a result, none of our benchmarks use this marking task. Nor would any standard C program.

Implementation Details To adapt this framework to the hinted collector, we replace the GC_DS_PROC marking task with a custom one of our own. In principal, our custom marking task could be implemented as one of the user specified marking functions (thus not breaking the functionality exposed to clients of the library), but our implementation does not do so. The new marking task encodes a pointer to one of the bottom_index structures and an index into the array of hblks contained by that bottom_index.

On entry to each phase of the algorithm, we walk (in serial) the linked list of bottom_index structures and place a marking task for each bottom_index found into the global mark queue. The index field, which will be used for the load balancing described shortly, is set to the maximum index. Since the array of hblk pointers is statically sized, this is a known small constant.

Once this is accomplished, each marker thread is started with its own local mark stack. Each will steal some work from the global queue and process it. When executed, the new marking task examines some set of hblk regions and pushes any work required to process them fully onto the local mark stack. The highest index remaining to be processed (in the array of hblk pointers) is then stored back into the task. When a task with nothing remaining to be processed is encountered, it is removed from the mark stack. There are two key design parameters here:

- Should the scanning of the hblk regions be immediate (i.e. done within the new marking task) or deferred (i.e. added to the mark stack)? Scanning immediately reduces traffic on the mark stack and improves best case performance. Deferring the scanning of hblk regions enables load balancing since the bottom_index marking task can be passed to another thread while the local thread is busy scanning hblks.
- How many hblk regions should be examined in a single step? A smaller granularity improves load balance in some cases (by allowing other threads to steal work), but does impose a performance penalty.

Some form of load balancing is clearly needed. Consider a program which allocates a small number of very large objects (for example, in a custom allocator). As a result, there are only a small handful of hblk regions in use - potentially fewer than there are mark threads. Without the deferred processing described above, some marker threads would be utterly unutilized. It is unclear how common such cases might be in practice.

Currently, we choose to enable load balancing by not scanning hblk regions immediately and process at least 10 hblks in each examination step. These choices appear to be a good balance between best case performance and worst case load balancing[1]. However, we can not claim to have fully explored the parameter space; there may exist a better balance that we have not identified.

There are two non-obvious concerns to this approach worth explicitly discussing:

---

[1] Worth noting is that the runtime performance of the collector is *heavily* influenced by the code quality of the inner marking loop. As a result, several optimizations that might seem like a good idea (ex: separate marking routines for phases 1 and 2), turn out to have negative performance impacts. It took significant careful effort to ensure that adding load balancing did not disrupt the code quality of this critical inner loop.

- First, each bottom_index task must be able to locally guarantee that the result of examining a given number of hblks does not overflow the mark stack. It must be the case that all entries placed on the mark stack by a given bottom_index execution can be processed in their entirety without overflowing. (This requirement comes from the fact that we do *not* use the mark-then-push scheme leveraged elsewhere in the collector to ensure progress in the event of overflow.)

  In practice, this has not been a concern since the per hblk functions in each phase store at most a fixed amount of data on the local mark stack (the number of objects in a single hblk). In addition to this, we rely on the facts that a) there can be at most a small constant number of bottom_index tasks on a local mark stack at any time, b) we process the mark stack in DFS order, and c) that we do not recursively follow edges during the first two phases. Together, these facts ensure that the local mark stack does not overflow. Various more complicated schemes could be devised to handle the relaxation of some of these assumptions, but this has not been a practical performance limit.

- Second, we assume that the number of bottom_index structures is small enough to all fit in the global mark queue at once. In practice, this has been the case for all benchmarks we've considered.

  If we did encounter a case where the number of bottom_index structures exceeded the mark stack size we could either a) simply grow the mark stack since the parallel threads haven't started yet, or b) run multiple iterations of the parallel step handling some of the bottom_index structures in each iteration. Since the per-hblk actions taken by phases 1 and 2 are dependence-free, this would be correct if potentially slow.

Interesting, the next performance bottleneck encountered in the parallel implementation was not phase 3 of the mark algorithm. Currently in phase 3, we scan all of the hinted blocks for marked objects in serial using only a single thread. We do leverage the baseline collector's mark implementation once the mark stack has been populated. We have not found this phase of the algorithm to be a performance limit except in one special case: that of `libquantum` from the SPEC 2006 benchmarks. This case will be discussed in Section 3.2.

We note that the implementation strategy described here was chosen primarily for speed of implementation, and that a – largely unexplored – design space of parameter choices exist within our current implementation. We would like to explore this further. Alternatively, there may be (and likely are) alternate parallel implementation strategies work pursuing.

Parallel Overhead & Wasted Effort

As would be expected, there is a higher initialization cost for running the parallel implementation than for executing the serial algorithm. This is true of both the hinted collector and the baseline traversal collector. In large heaps, these overheads are largely amortized away, but for smaller heaps the differences in absolute mark times can be significant.

When inspecting data on these small collections, we noticed an interesting pattern. *Many of the small collections were not collecting any garbage at all.* This was true of both the hinted and traversal collectors. Further investigation revealed that many of these collections were occurring during the warmup phase of an application; objects were being allocated, exhausting the currently available heap space, and then forcing an expansion of the heap. A key observation was that no deallocation had occurred since program initialization. Several benchmarks show multiple small collections which fit this pattern.

Another interesting pattern revealed in the SPEC 2006 results from the serial collector was that all collections for some benchmarks (in particular `bzip`, `ibm`, `mcf`, and `sjeng`) fail to reclaim any garbage. This isn't simply a quirk of the collectors; these benchmarks fail to deallocate any data until after the last collection. This implies that *every collection performed during the execution of the benchmark is wasted effort* for both collectors.

To capitalize on these observations, we introduced a special case to the hinted collection algorithm which skips the collection if there are no hinted objects in the heap. This avoids wasted work since the amount of memory reclaimed by a hinted collection is bounded from above by the amount hinted. This is an optimization that is only possible for a program with deallocation hints. We refer to this optimization as the **profitability-check**. The benefit of this optimization can be clearly observed by comparing the serial (run without) and parallel results (run with) for the previously discussed benchmarks.

Currently, we implement this optimization only for the parallel hinted collector. There is no reason it couldn't be extended to the serial version. We deliberately disable this optimization for most of the microbenchmarks results reported since it complicates the writing of effective benchmarks.

In principle, this exact check could be extended to a heuristic based on an estimate of profitability of the upcoming collection. The amount of hinted data provides one easy upper bound, but an estimation heuristic could also use information about reclamation rates from previous collections. We have not investigated this further, but believe such a check might be worth implementing. Similarly, we have not explored extended this check to the traversal collector in a combined system.

Metadata Tracking Overhead

After parallelizing phases one and two of the mark algorithm, we observed that while the core mark algorithm for the hinted collector was faster than that of the traversal baseline collector, the overall mark time was not. In fact, the difference was a large fraction - roughly 20% - of the absolute mark time . The portion outside the core mark algorithm includes the logic to stop-the-world, clear mark bits, initialize the collector, start lazy reclaim, and clear hint metadata after the collection. (All times reported elsewhere include the time for these actions in addition to the core mark algorithm.)

Investigation revealed that two serial actions were limiting performance: 1) the check for hinted objects just described, and 2) the need to clear the per-hblkhdr hint flag after the collection had completed. Thankfully, each is easily addressed.

- The scan for hinted objects in the profitability-check optimization can be replaced with a single global flag which is set the first time a hint is given during a collection cycle.

  It is worth noting that the original implementation of the check was not an early-exit loop. Even if a hinted hblkhdr was found, every hblkhdr would be visited. It is possible that simply using an early-exit loop would give the same benefit for the case when a collection must occur. The single flag implementation is clearly superior when the collection is not needed.

- The clearing of the per-hblkhdr metadata can be combined with the initialization of lazy sweeping. The lazy sweep implementation needs to visit every hblkhdr to reclaim the block if the entire region is empty or queue it for later processing if not. By combining the two visits, we can remove the extra scan required by the hinted collector. Note that combining the hint clear is not strictly free; it introduces a write into a code path which previously only performed reads in the common case. Despite this, overall, it is clearly a net win.

With these two changes, the time outside the core mark algorithm for the hinted collector and the traversal collector became roughly equivalent. Interestingly, on large heaps both collectors still spend significant fractions of their runtime clearing mark bits in serial on the main thread before even beginning the mark algorithm. Since this was not a difference between the two, we did not investigate parallelizing this step; we believe it would likely be profitable to do so if the goal was to improve absolute rather than relative performance.

3.2 Evaluation

This section focuses on evaluating the performance of our parallel implementation run with four marker threads on a machine with two cores (each 2-way SMT, giving 4 hardware thread contexts). This is the same machine used for the majority of the evaluation of the serial collector.

		Hinted Collector			Tracing Collector		
benchmark	heap size	1P	4P	**speedup**	1P	4P	**speedup**
Linked List (Dead, Hinted)	31.8 MB	2.00	1.05	**1.90**	1.45	1.00	**1.45**
Linked List (Dead, Unhinted)	31.8 MB	7.75	5.70	**1.36**	1.40	1.00	**1.40**
Linked List (Live, Hinted)	31.8 MB	12.10	21.35	**0.57**	11.95	13.00	**0.92**
Linked List (Live, Unhinted)	31.8 MB	7.40	5.90	**1.25**	12.10	13.10	**0.92**
Fan In (Dead, Hinted)	23.7 MB	0.00	0.25	**n/a**	0.00	0.05	**n/a**
Fan In (Live, Hinted)	23.7 MB	12.90	6.45	**2.00**	12.85	5.25	**2.45**
Fan In (Live, Unhinted)	23.7 MB	8.25	7.05	**1.17**	12.50	5.10	**2.45**
2560 x 1k element LL	88.6 MB	19.80	13.90	**1.42**	31.70	13.00	**2.44**
256 x 10k element LL	88.6 MB	18.90	12.80	**1.48**	30.95	13.20	**2.34**
1/3 Cleanup	184.6 MB	31.95	30.60	**1.04**	50.10	33.65	**1.49**
Deep Turnover	56.6 MB	12.30	7.55	**1.63**	20.50	16.35	**1.25**
Unbalanced (Live)	744.6 MB	176.50	97.95	**1.80**	252.85	110.25	**2.29**
Unbalanced (Partly Dead)	744.6 MB	175.25	96.60	**1.81**	246.30	109.00	**2.26**

Table 3.1: Average collection times (in ms) for microbenchmark results. The machine used in these experiments had two cores with 4 hardware contexts total. One key methodology note is that we have *disabled* the profitability-check optimization for these results. This was done to give more insight into the general case without having to modify the benchmarks to deallocate.

Methodology The methodology for comparison is identical to that used for evaluating the serial collector. The only difference is the baseline collector used. In this section, we compare against the parallel stop-the-world mark algorithm provided by the baseline collector.

Worth noting is that the baseline collector is run from the same code base as the hinted collector. We have strived to avoid modifying codepaths used by the baseline collector, but there is the possibility we may have biased the results. We don't believe this has occurred. Where possibly we have sanity checked our results against an unmodified baseline collector, but we can not always do so.

Microbenchmark Results

In Table 3.1, we contrast the performance of our hinted collector and a baseline tracing collector. One key methodology note is that we have *disabled* the profitability-check optimization for these results. This was done to give more insight into the general case without having to modify the benchmarks to deallocate.

From a parallelization perspective, the first seven benchmarks are relatively uninteresting. As we would expect, the hinted collector outperforms the tracing collector when given exact hints, but the inverse holds when entirely inaccurate hints are given.

The **2560 x 1k element LL** and **256 x 10k element LL** benchmarks consist of a set of linked lists reachable directly from the root set. As we would expect, both collectors are able to fully utilize the available memory bandwidth with four threads and achieve essentially the same performance. Worth noting is that the hinted collector performs substantially better in the single threaded case.

Two cases worth highlighting occur in **1/3 Cleanup** and **Deep Turnover**. These benchmarks highlight the common case where only a section of the heap graph is collectible. With four threads, the traversal collector has not matched the hinted collector's single thread performance. In each case, the parallel hinted collector reduces pause times further. We see the same general trends in the **Unbalanced** benchmarks, but the results are less pronounced.

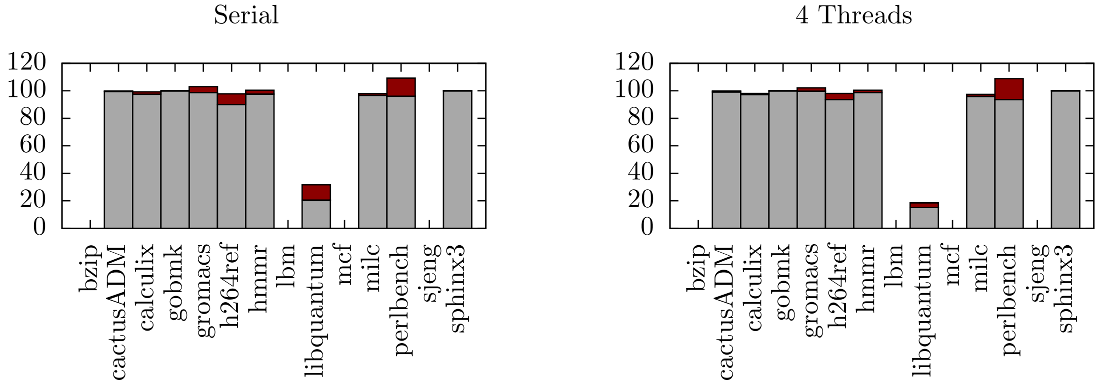

Figure 3.1: Percentage of memory reclaimed (gray) and leaked (red) by hinted collector. 100% indicates the amount of memory manually deallocated. The sum of the amounts collected by the hinted and tracing collectors can exceed this amount.

SPEC 2006 Results

Full results from running a subset of the C programs in the SPEC 2006 benchmark suite can be found in Table 3.2 and 3.3. The format of these tables is identical to those from the serial collector, as is the set of benchmarks run.

The results for the amounts of memory reclaimed by the parallel collector are largely uninteresting. As one would expect, the amounts of memory reclaimed by the parallel version of the hinted collector are quite similar to those amounts reported by the serial implementation. Some benchmarks leaked slightly more, some leaked slightly less, but in all cases the changes are within normal variation. There is slightly more variation in parallel runs than serial runs; we believe the variation to be complete explained by differences in collection timings across runs. A graphical summary of the results can be found in Figure 3.1.

The pause time results show the general trends we would expect. Both collectors show substantial speedups for collections which were previously long running, but have non-trivial slow downs for small collections due to coordination overhead. For benchmarks eligible for the profitability-check optimization, hinted collection times drop to zero. For the benchmarks with pauses over 100ms in the serial run, the reduction in worst-case pause time is on the order of 2-2.5x across the board for the hinted collector. The traversal collector sees the same general trend, but with one glaring exception (`perlbench`) which we'll discuss shortly. The exceptions to these trends reveal some interesting facts:

- `libquantum` – As noted when discussing the memory leakage results for the serial collector, the `libquantum` benchmark is retaining pointers to large amounts of memory after freeing it. This means that neither collector can reclaim most of the freed memory. It also implies that most of the hints given to the hinted collector are inaccurate. By breaking down the results (not shown), we see that the majority of the runtime for the hinted collector is in the phase 3 traversal. As a result, the performance of the hinted collector for this benchmark is limited by the performance of the tracing collector.

  As an experiment, we modified each of the calls to free in this benchmark to assign NULL to the pointer passed to free[2] and then reran only this benchmark. As expected, this removed some of the dangling references to free data.

---

[2] This was done by defining a macro which overrode the definition of free provided by the C standard library. Our macro replaced each call to free with a direct call to the GC_free routine and an assignment to NULL. This is sufficient for a test, but in general, such macro magic is extremely dangerous. We note that this technique only works for a subset of the benchmarks in SPEC since it can not distinguish between rvalue and lvalue arguments.

benchmark	hinted	reclaimed	% rec	leaked	% leak
bzip	0 bytes	0 bytes	n/a	0 bytes	n/a
cactusADM	5.74 GB	5.69 GB	99.15	42.87 MB	0.75
calculix	64.05 GB	62.26 GB	97.20	595.16 MB	0.95
gobmk	2.89 GB	2.89 GB	99.97	1.85 MB	0.06
gromacs	36.84 MB	36.75 MB	99.75	899.22 KB	2.39
h264ref	3.81 GB	3.57 GB	93.68	165.41 MB	4.42
hmmr	8.08 GB	7.98 GB	98.83	133.69 MB	1.65
lbm	0 bytes	0 bytes	n/a	0 bytes	n/a
libquantum	2.23 GB	336.77 MB	15.12	11.66 MB	3.35
mcf	0 bytes	0 bytes	n/a	0 bytes	n/a
milc	259.71 GB	249.19 GB	95.95	3.72 GB	1.47
perlbench	103.36 GB	96.67 GB	93.52	17.54 GB	15.36
sjeng	0 bytes	0 bytes	n/a	0 bytes	n/a
sphinx3	48.71 GB	48.67 GB	99.91	141.29 MB	0.29

Table 3.2: Summary of space reclamation across all collections for each SPEC benchmark when run with four marker threads for each collector. “hinted” is the total number of bytes directly hinted by the application. “reclaimed” is the amount of space reclaimed by the hinted collector. “% rec” is the percentage of hinted data reclaimed; due to collateral hinting, this can be greater than 1.0. “leaked” is the total bytes reclaimed by the tracing collector. “% leaked” is the leaked column divided by the total memory available for collection (leaked + reclaimed).

		Hinted Collector				Tracing Collector			
benchmark	count	min	max	mean	median	min	max	mean	median
bzip	36	0.0	0.0	0.00	0.0	2.0	40.0	19.89	20.0
cactusADM	33	0.0	66.0	30.39	18.0	0.0	66.0	30.06	18.0
calculix	789	0.0	49.0	7.81	0.0	0.0	34.0	6.59	0.0
gobmk	158	0.0	16.0	5.85	6.0	0.0	9.0	5.26	6.0
gromacs	16	0.0	1.0	0.56	1.0	0.0	1.0	0.56	1.0
h264ref	138	0.0	8.0	3.07	3.0	0.0	7.0	2.79	2.0
hmmr	762	0.0	11.0	0.84	0.0	0.0	4.0	0.38	0.0
lbm	3	0.0	0.0	0.00	0.0	15.0	17.0	15.67	15.0
libquantum	21	2.0	511.0	79.19	10.0	2.0	514.0	79.71	10.0
mcf	3	0.0	0.0	0.00	0.0	0.0	1.0	0.33	0.0
milc	147	0.0	240.0	182.50	190.0	22.0	246.0	184.19	193.0
perlbench	546	0.0	153.0	58.02	60.5	0.0	614.0	68.01	69.5
sjeng	6	0.0	0.0	0.00	0.0	4.0	10.0	7.17	8.0
sphinx3	1537	0.0	18.0	6.44	6.0	0.0	8.0	5.06	5.0

Table 3.3: Statistical summary of pause times (in msec) observed for each SPEC benchmark when run with four marker threads for each collector. The first set of columns are from the hinted collector; the second set are a standard collector run immediately after the hinted collector completes.

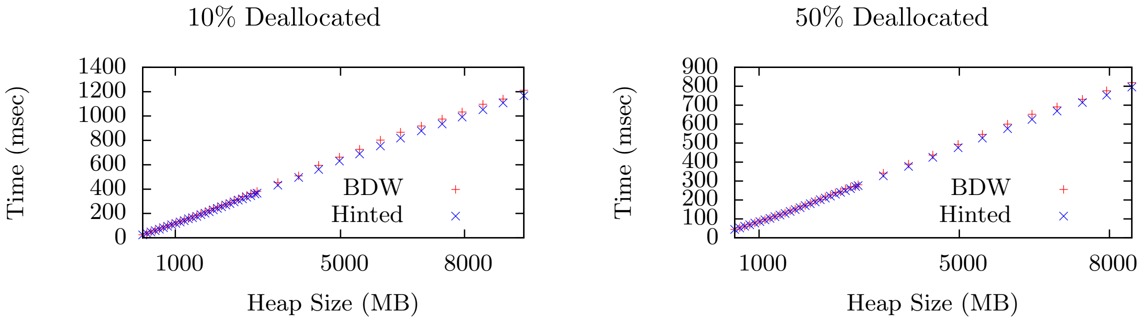

Figure 3.2: The parallel collectors scale with heap size and with the amount of live data. (The hinted collector actually scales with the total amount, but in this case the effect is the same.) Note that the y-axis ranges in the two plots are not the same.

benchmark	hinted	reclaimed	% rec	leaked	% leak
libquantum	2.45 GB	682.91 MB	27.88	5.39 MB	0.78

Examining the results, we see that there is substantially more memory reclaimed in absolute terms (688.3 MB vs 348.4 MB) and that the fraction reclaimed by the hinted collector also increases (27.88% vs 15.12%). Oddly, the total amount reclaimed is still low. This implies that there must be other dangling references not involved in calls to free. Despite this, the runtime of both collectors improved substantially.

		Hinted Collector				Tracing Collector			
benchmark	count	min	max	mean	median	min	max	mean	median
libquantum	28	2.0	378.0	33.68	6.0	2.0	383.0	33.82	6.0

- `perlbench` – For this benchmark, the traversal collector *slowed down* when adding more threads. This is in sharp contrast to the hinted collector whose performance improved by nearly 2x. As a result, the hinted collector is nearly 4x faster for this benchmark. This result hints at a larger issue with the scalability of the baseline tracing collector.

Overall, we consider these to be highly encouraging results. When moving from one thread to four, we see a consistent scaling for the hinted collector of around 2-2.5x. This is not strong scaling with the number of cores, but closely matches the read bandwidth available. When running a simple read bandwidth benchmark on the same machine, we see approximating the same practical scaling behavior as we add threads. (1 thread: 8GB/s, 4 threads: 18GB/s, ideal: 21.3GB/s) The fact that we are seeing the same scaling trend is a strong result.

Heap Size Scalability

As with the serial collector, we ran a simple experiment where we varied the amount allocated over a wide range and varied the percentage deallocated before a collection. As we can see from Figure 3.2, the running times of the two collectors are comparable across the entire range. The hinted collector does have a very slight advantage, but nowhere near as much as seen for the serial collector[3].

---

[3] This is the same benchmark as used to evaluate the scalability of the serial collector, but is run on a different machine ("Machine 2" vs "Machine 3") due to the variance issues described in Section 3.2.

Digging into these results for larger heap sizes, we found that both collectors are spending a large fraction of their time in serial code clearing the mark bits before the collection begins. At small heap sizes, the effect is small, but with larger heap sizes an increasing fraction of total runtime is spent in this serial code (ex: roughly $\frac{1}{8}$th of total execution time for the 3.5 GB data point). This code is shared by both collectors. It would be fairly simple to parallelize this step, but the need to do so hints at a lack of tuning in the baseline implementation for large heap sizes.

Parallel Scalability

All of the results presented previously are for four marker threads. We have also run a number of experiments on machines with larger core counts, but have had somewhat disappointing results. As you can see from the results in Figure 3.3, both collectors quickly approach a performance plateau and stop scaling with the number of threads. It appears that on most of the benchmark we have examined that both collectors are limited by the available memory bandwidth on the system. When we plot measured memory bandwidth against number of threads in use, we see very similar curves.

More troubling, when we increase the number of marker threads past low double digits, we began observing extremely high variance in the execution times. The baseline tracing collector is particularly impacted (with 4-5x slowdowns observed), but the hinted collectors variance increases noticeably as well. Interestingly, the hinted collector and the traversal collector exhibits variation on largely non-overlapping sets of benchmarks.

After examining the data, we could not identify a hypothesis that seemed to clearly explain this variation. Through a series of experiments, we have been able to eliminate the following factors with reasonable confidence: NUMA latency, thread preemption, cache contention, contention in the memory system, use of hyper-threading, and our own code changes. Given that we can not explain these results, we have chosen not to include them in full. All of the raw data files can be found in the publicly released source code repository.

Previous work by Endo et al [22] has identified several contributing causes that limit the scalability of the Boehm-Demers-Weiser collector. Since the time of that work, some (but not all) of the improvements suggested have been integrated into the baseline collector. One avenue for future work would be to compare our parallel hinted collector implementation against that of Endo et al [22].

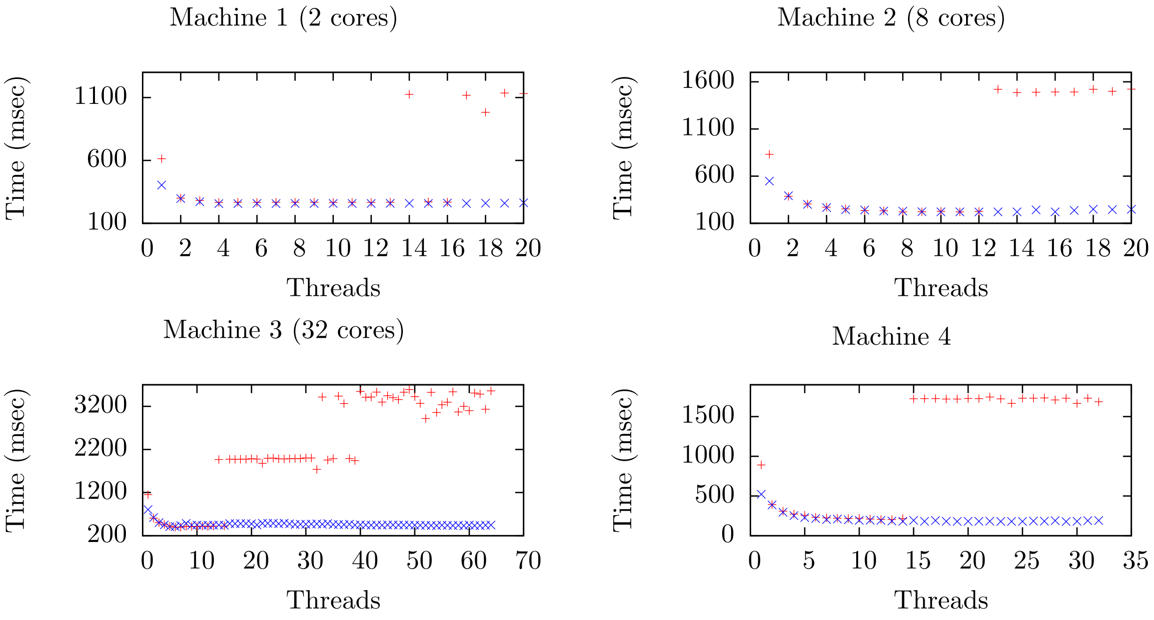

Figure 3.3: Scalability of the collectors with various numbers of marker threads on four different machines. The benchmark used is the same as the heap size scalability experiment.

Chapter 4

Discussion

This chapter focused on the idealized hinted collection algorithm and some of its implications. The points raised here should apply equally to the serial and parallel collector implementations of the last two chapters.

4.1 Memory Leaks

A hinted collector can leak memory unless paired with a backup collector. Such leaks can come from several sources:

- Missing hints (direct) - If the user fails to provide a hint for an object, it will not be reclaimed.
- Missing hints (indirect) - If the user provides a hint for a given object, but does not provide a hint for an object which references it, neither object will be reclaimed. The collector can not distinguish between an object being retained due to references from live objects and references from dead objects which are merely unhinted. As one special case of this, any cycle which contains an unhinted object will be retained in its entirety.
- Hint races - If an object is hinted just before a collection, the last reference might survive into the collection where the hint would be cleared without being the object being collected.
- Conservative References - Since our collector is a conservative collector, we may falsely identify a word value as a valid reference. This could cause an object subgraph to be falsely retained.
- Old References - Our collector scans all objects in hblks that contain unhinted objects; it does not distinguish between objects which might be live and those known to be dead - such as objects on a free list. As a result, references in previously reclaimed objects can force the retention of hinted object.

Out of these sources, only the first two are fundamental to our approach; the latter are artifacts of our particular implementation and could be avoided with an alternate design. Fortunately, as we saw with our case study in Section 2.2, very little memory is leaked in practice. We have not attempted to break down the contributing causes.

One slightly unintuitive fact is that a hinted collector can leak memory which was correctly hinted. If a pointer into the hinted subgraph is found in any unhinted object, that subgraph will be traversed by the collector and retained. In the worst case, a single missing hint can force the retention of an arbitrary subgraph. Consider a long linked list which is unreachable. If all but the head node of the list is hinted, the collector will find an edge crossing from unhinted to hinted and retain the entire list. We have not observed this to be a practical concern. It is possible – though not, we believe, likely – that the use a per-hlbkhdr flag for hinted metadata is obscuring the impact of such missing hints. Another possibility is that our use of mature programs written for manual deallocation – and thus not expected have many missing hints – might be causing our results to understate the performance impact of missing hints.

During development we did encounter a case where old references triggered retention of large amounts of unreachable objects. In the linked list benchmark, one node from the previous iteration was not reused and by happenstance both lived on an unhinted hblk and pointed into the new list at an early position. We believe this to be a very unusual reuse pattern that is unlikely to arise in real programs. As a safeguard, we plan to introduce a concurrent cleaner to break references in dead objects after they have been reclaimed. This would prevent old references from accumulating over time.

In principle, the hint metadata could be accumulated across multiple collection cycles. This would result in some additional objects being reclaimed over time, but at the cost of inaccurate hints accumulating and slowing down the collector. From our experience with patterns of common mistakes in manual deallocation, it is not clear this would be a profitable approach. In practice, we chose to reset the hint metadata on every collection. This is currently a somewhat arbitrary choice.

4.2 Manual Reasoning & Software Engineering

As noted earlier, our entire approach is premised on the assumption that it is reasonable to ask programmers to understand object lifetimes in their programs. We believe the prevalence of programs written in languages with manual deallocation to be conclusive evidence that it is. By that same evidence, we accept the fact that such reasoning is not always simple and that programmers can not in general be *always* correct about object lifetimes. In the context of our current work, we believe that our results clearly demonstrate that real programs provide enough accurate hints to justify the use of a hinted collector.

Taking a step back, we acknowledge that there are times - such as when implementing lock-free data structures - where *not* having to explicitly manage memory greatly simplifies design, reasoning about correctness, and can increase performance. These are among the key reasons that garbage collection has been such a boon for software engineering productivity.

In a production collector, we would expect to pair a hinted collector with some form of backup collector to ensure that small leaks do not accumulate over time. Conceptually, this is very similar to a manual memory deallocation scheme paired with a background collector to increase the reliability and uptime of a long running process, but without the potential unsoundness of trusted deallocation. We have not explored the design space of possible pairings, and suggest this would be a profitable area for future work.

Long term, we would like to explore what programs written from scratch in a language with deallocation hints would look like. We expect that most programmers will rely on common programming idioms or patterns, but not invest in providing deallocation hints in most cases. This is perfectly acceptable; if the concurrent collector can keep up with the application's needs, this is entirely desirable.

In such programs, we foresee deallocation hints coming from two sources. First, library authors are likely to provide hints where possible; widely used libraries already go out of their way to simplify memory management - even in languages with garbage collection. Second, when an application does encounter the limits of the backup collector, we foresee programmers selectively adding deallocation hints to reduce burden on the backup collector. We expect profitable sites would be identified via a profile-guided optimization methodology using some form of an instrumented collector to record reachability. This is similar to how programmers tune the garbage collection performance of Java programs today.

4.3 Metadata Design Alternatives

One of the key design decisions in a hinted collector is how to store the set of hinted objects. As described in Section 2.1, we chose to store a single bit in the hblkhdr - implicitly giving hints for many objects when any one is given. In retrospect, we do not believe this to be the ideal design.

In many ways, the choice of how to store whether an object has been given a deallocation hint parallels the ways to record mark bits in a standard collector; much of the previous work from that field should carry over. For instance, one could place a status flag in an object header, a bitmask stored on the side, a flag per group of objects, or even an extra bit per object.

Interestingly, there are also options which are unique to hinted collection. As an example, one could take advantage of hinted collection's tolerance of approximation by using a data structure such as a bloom filter to store an approximate set of deallocation hints. This would reduce the amount of storage required at the cost of some additional objects being hinted (i.e. false positives). Bloom filters have well understood trade-offs between size and false positive rates; it would be interesting to explore this design space.

Critique As a reminder, in the current implementation we choose to store hint metadata using a single-bit flag for each hblkhdr (a large group of objects of common size). This has the effect of implicitly hinting many objects - all those described by the same hblkhdr - every time a true hint is recorded.

The issue with our current scheme is that phase 3 of the collector algorithm is no longer something which runs only when the user gives inexact hints. Instead, a user may give exactly the right hints (all accurate, none missing) and the collector might still be forced to explore a large section of the (live) heap graph since large portions of it may have been implicitly hinted.

While this has not been a problem so far, we are dissatisfied having this case invoked when exact hints have been given. Since its parallel scalability is limited by heap depth, this portion of the algorithm is likely to be a bottleneck in a parallel collector – as we see with `libquantum` in the four mark thread SPEC 2006 results. Another downside is that the current behavior potentially endangers the tune-ability that is so attractive about a hinted collector. It is unclear with the current implementation whether adding a new (accurate) deallocation hint is actually going to improve performance. If one got unlucky and implicitly hinted part of a large linked list, adding an (accurate) hint could hurt performance substantially.

One potential cost of moving away from the current implementation is that implicit hinting of objects gives some protection against missing hints forcing the retention of unreachable objects. As mentioned previously, we don't believe this to be a major effect, but it is something to be considered.

marked-by-default If we eventually re-implement our collector, we plan to reuse the existing mark bits to store deallocation hints between collections. The basic scheme would be to mark all objects initially on allocation and only unmark an object when a deallocation hint is given. The lack of marking would indicate a deallocation hint had been given for that object and that object only; there would be no collateral damage, as there is now. During collections, the mark state invariant would be restored by marking any hinted objects as in phases 2 & 3 of the current algorithm. Notably, only missing or inaccurate hints would trigger phase 3.

There are two tricks to this representation. First, we would need a way to perform the edge filtering optimization. Assuming that per object mark bits continue to be stored in the page header as a bitmask, this check could be implemented via a series of bit operations. Second, having objects which are live, but unmarked between collections complicates lazy-sweeping to reclaim objects. The easiest solution would be to store a "safe-to-sweep" bit in the hblkhdr that is cleared on the first deallocation hint to a hblk. This same flag could be used for the edge-filtering optimization as well. An alternate approach would be to use multiple sets of mark bits – similar to how one might support concurrent marking and sweeping as in [18].

One optimization this scheme makes obvious is to restore the page level flag if all the objects in the hblkhdr become marked during the collection. This could potentially improve the efficiency of the edge-filtering optimization within a single collection. We note that this optimization isn't actually specific to this representation of hints, but follows more naturally from this then our current implementation. The performance of this would need to be explored.

An additional possibility enabled by this design is the optional integration of a read barrier that could silently fix mistaken deallocation hints if the object is again accessed. This read barrier is not necessary for correctness. It is not clear that it would be a net win, but further investigation is certainly merited. Again, this optimization could be used for any storage scheme that has one hint flag per object. It is not specific to the marked-by-default scheme described here.

One concern that might arise when considering a marked-by-default design is the possibility that out-of-bound writes from the mutator could cause the collector to reclaim an object which isn't actually dead. With an adversarial mutator, we can't write out this possibility entirely (neither can the standard tracing collector), but by the properties of the hinted collector algorithm we can be sure that arbitrary writes to only the mark-bits will not cause collector malfunction. Assuming the roots are gathered correctly and every outbound edge from a marked object is explored, the actual reachable subgraph must be marked after the collection. This follows from the properties of reachability given in Section 1.2.

4.4 Explicit Hinted Object Sets & Hazard Pointers

In this section, we describe an alternate implementation strategy for a hinted collector that tracks an explicit set of hinted objects and periodically checks the membership of that set against the set of live pointers. The key difference here is that the set of hinted objects is tracked as a separate data structure rather than by using hblkhdr flags or reusing mark bits (see 4.3). This implementation is partly of historical interest - it is actually the first one we implemented - but is also interesting as it highlights parallels with hazard pointers [27] - a methodology used for safe deallocation in lock-free algorithms in languages with manual deallocation - which are not as clear with the current algorithm.

Before considering the current algorithm for hinted collection, we had constructed a prototype implementation that was primarily intended for instrumentation rather than collection. It was far too slow to be a viable collector. The basic strategy was to create an explicit linked list of deallocated but not yet reclaimed objects. Periodically, we would walk through the entire heap checking to see if any pointer we encountered was also in the list of deallocated objects. If it was, we'd remove that object from the link list and scan it as well. There are obvious ways to accelerate this, but we never implemented them since we moved to the new algorithm instead.

What's interesting about this implementation is that it is *nearly identical* to the "scan" operation of a hazard pointer implementation. As a brief reminder, the hazard pointer methodology [27] is widely used to ensure correct high-performance object recycling in lock-free algorithms and data structures in languages with manual deallocation. Objects that are to be retired are placed on a special "retired list". Periodically, this list is scanned for objects which are known to be dead. The key invariant that must be upheld by an implementation is that there must be a pointer to any retired object which might still be in use in a special set of roots called "hazard pointers". Pointers to retired objects can also exist elsewhere, but if they do, there *must* also be a pointer in the set of hazard pointers which refers to that object.

In the terminology of hazard pointers, our list of hinted objects is simple the "retired list". The key difference between hazard pointers and the more general hinted collection is in what set of potential source pointers gets checked against the objects in this list. The invariant upheld by hazard pointer implementations provides two key guarantees our hinted collector does not have:

- First, that scanning only the set of hazard pointers (i.e. not other stack variables, or heap objects) is sufficient to soundly identify any reachable retired objects.
- Second, that recursive traversal of retired objects found to be potentially live is not required.

To put this another way, hazard pointers are simply a specialized implementation of hinted collection which leverages a user maintained invariant to avoid exploring most edges in the heap.

In making this reduction, we do not mean to minimize the importance of the hazard pointers methodology. In practice, hazard pointers is a far more efficient implementation for this particular use case than hinted collection will ever be. What we find deeply exciting is that there may be other special cases of hinted collection - leveraging some other, as yet unknown invariant - which could be used to reason about the correctness of other interesting patterns of manual deallocation.

Chapter 5

Future Work

This chapter focus on work that we have not completed. The first section discusses ideas for direct extensions of the current work. The second describes a set of design sketches for how our current stop-the-world implementation of hinted collection might be extended to a generational, incremental, or concurrent collector. The final section takes a step back and describes two alternate approaches to the problems hinted collection aims to address. This entire chapter is somewhat speculative by nature.

5.1 Direct Extensions

Integration

Our current effort has focused on evaluating the performance of a hinted collector vs a standard tracing collector in isolation. As a result, we've tried to keep as much of the environment fixed as we could. This has meant that we have not modified such factors as the frequency of collections or allocation strategy. For most of our experiments, we have substituted eager sweeping to ensure a fair comparison between the two collectors. It would be interesting to explore how the performance of a hinted collector changes when the timing of collection is influenced by its own collection rate (rather than the traversal collector's) and what information about scheduling could be extracted from deallocation hints being provided by the program. We have not explored any of these design choices and recommend them for future work.

It might be profitable to choose between hinted and traditional collection at runtime based on properties of the program observed. One interesting approach would be to use techniques from machine learning classification to learn a dividing hyperplane through the parameter space. This might provide interesting insights that could be leveraged in making runtime decisions. This is an aspect that we have not pursued.

Depth Bounded Hinted Collection

One way of accelerating the hinted collector further would be to give up (i.e. reclaim nothing) if a deep structure was detected in the hinted subgraph. By avoiding the degenerate case where a hinted collection is forced to explore a large portion of the heap to ensure any objects reclaimed are unreachable, you could improve average and worst case hinted collector pause times at the cost of some increased memory leakage. This would have the effect of strongly encouraging correct hint usage (i.e. to get good performance), but would not sacrifice correctness or even memory leakage provided a backup collector was present in the system.

Design Space Exploration

As should be clear from the previous chapters, the space of viable designs for a hinted collector is as large if not larger than the design space of traditional tracing collectors. Thankfully, many of the design *decisions* - if not the appropriate *choices* - are fairly clear from the preceding discussion and the literature on traditional tracing collection. We would strongly suggest that, rather than building a particular point in this huge design space, that future work should instead focus on building a collector generator - i.e. a program which is parametrized by the available decisions and outputs appropriate code which can be compiled and tested - and use auto-tuning and design space exploration techniques to discover the most profitable designs. In retrospect, this would have been a much better approach for the entire project.

A few design choices that we believe might be particularly profitable to explore are:

- The traversal order used in phase 3 of the mark algorithm. The current implementation inherits the traversal code from the baseline collector (i.e. DFS in serial collections, DFS w/BFS-like work stealing in parallel collections), but it is unclear whether this is the ideal strategy for a hinted collector. In principal, a "typical" traversal in phase 3 might look very different than a standard collection.
- The collection scheduling heuristic. Exploring the space of possible decision heuristics (which are mainly linear combinations of statistics with a few branches), could be quite profitable and should be easily tractable with a fairly simple auto-tuner.
- The load balancing of the parallel implementation. There is a delicate balance between single thread performance and load balancing. We lightly explored this space, but a more extensive evaluation would be worthwhile. There might also be room for runtime decisions which we have not explored.

Minimum Number of Hints

The current work has focused on establishing the core idea behind hinted collection and establishing that such a collector is feasible at all. With this goal, we have focused on using mature programs from a language with manual deallocation. This has ensured that the set of hints are likely to be reasonably exact. It would be interesting to explore whether hinted collection is still feasible when hints are much less exact. We see two worthwhile approaches:

- The first would be to implement hinted collection for a garbage collected language - i.e. one without existing free calls which can be leveraged as hints - and determine what the smallest code change required to achieve good performance would be.
- An alternate approach to the same question would be to start with a language with manual deallocation, combine a hinted collector with a fallback traversal collector, and then reduce the number of free calls in the benchmark programs until performance starts to drop off.

We suspect that the number and placement of deallocation hints required in either case would be relatively small, but we also suspect the two approaches might arrive at subtly different answers. This difference would be extremely interesting since it would hint at the fundamental design difference of programs written with manual deallocation vs garbage collection in mind. We note that a similar experiment could be run with a system which combined traditional garbage collection and manual deallocation (like the Boehm-Demers-Weiser collector), and might provide similar insight.

Compile Time Deallocation & JIT deallocation

One possibility we would like to explore is how deallocation hints might affect efforts on compile-time object deallocation [16, 17, 24]. If a compiler could predict with high accuracy where an object was likely to become dead, a deallocation hint could be automatically inserted, *even if the compiler could not prove the correctness of deallocation at that point.* This enables the use of unsound analyzes and techniques such as purely local pattern matching. We are particular excited by the possibilities of what a just-in-time compiler could do with a combination of runtime profiling and compiler insertion of deallocation hints.

5.2 Advanced Collector Design Sketches

In this section, we give rough design sketches for incremental, concurrent, and generational versions of a hinted collector.

Generational Collection

Extending a hinted collector to a generational scheme is fairly straight-forward. As with any generational collector, a means to track reference between generations is required. Once this has been provided, the inter-generational references can simply be treated as additional roots in phase 1 of the hinted collector algorithm. The region of the heap outside the current generation can be excluded from the rest of the algorithm with the addition of simple boundary checks on the ranges of hblks explored and a range-based filter during reference detection in phase 3[1].

Worth noting is that the profitability of using a hinted collector for one generation in a generational collector is far from clear. We suspect that a hybrid implementation – one with both a hinted and traversal collector available for a given generation – would actually be more profitable.

One scheme would be to have the hinted collector run at a slightly higher frequency than the standard collector for the young generation. If the hinted collector – possibly a depth bounded variant (see 5.1) to minimize pause times – could achieve a high enough reclamation rate, promotion of objects to the older generation could be delayed. Analogous to the previous possibility, a hinted collector could be used in concert with a tracing collector to collect the older generation. This could potentially reduce the average cost to collect a large old generation, but would not benefit worst case collection times.

An alternate idea would be to use the information provided by deallocation hints to dynamically resize the youngest generation as in [7]. By leveraging information about when objects are *expected* to be dead, a generational collector could delay collection (by expanding the young generation) until a threshold value was breached or a specific rate of deallocation was observed. Assuming that hints were mostly accurate and relatively few missed - a fact that could be observed and potentially predicted by a runtime system - this could allow a generational collector to avoid wasted collections of the young generation (i.e. promotion of objects into the old generation which will die early.) This isn't strictly speaking a use of the hinted collection algorithm described in this thesis, but it would certainly be a use of *deallocation hints*.

Incremental or Concurrent Collection

Interweaving the execution of a hinted collector with mutator activity should require relatively little innovation beyond what is done for tracing collectors today. During the execution of phase 2, a barrier is needed to ensure that references to hinted objects are not written into unhinted objects which have already been scanned. Without such a barrier, it is possible that a hinted object would escape being marked by the collector and be incorrectly reclaimed. Assuming we are using the marked-by-default scheme described in Section 4.3, phase 1 no longer exists, and phase 3 is simply a standard traversal to which all the standard approaches should apply.

One interesting observation is that unless the mutator was actively manipulating references in or pointing into the hinted subgraph - which would be an odd program to say the least - few barriers would be triggered by the mutator. As a result, any form of barrier trap storm - where mutator performance is negatively impacted by correcting garbage collector invariants for a large number of objects in a small time period - would be highly unlikely.

As with the options described for integrating a hinted collector into a generation collector, there are several obvious possibilities for combining concurrent hinted and tracing collectors into a combined system. When you introduce the possibilities of depth-bounded hinted collection (see Section 5.1) and truly concurrent

---

[1] Note that we are ignoring any additional issues introduced by the fact our prototype is a conservative collector. The unique challenges of building a non-relocating conservative collector have been well explored [19, 7], and should map cleanly to a hinted collector if required.

mark and reclaim phases [18], the design space is too large to make any prediction of profitability with reasonable confidence. We can only suggest this as a potential avenue for future work.

5.3 Alternate Approaches

This section describes two alternate approaches to leveraging developer knowledge to accelerate collection without sacrificing soundness. The first is an alternate take on hinted collection, while the second inverts the idea of deallocation hints to consider what could be done with hints that an object should be assumed live instead of dead.

Hinted Edges

An alternate framing to the idea of a hinted collector is to associate deallocation hints with the edges in the heap graph rather than the objects. The intuition behind such edge hints is that the hinted edge should not contribute to the reachability of the target object unless the source of the hinted edge is itself reachable. When an object is expected to be reclaimed, all of the object's outbound edges would be marked as hinted and the local reference to that object would be made null.

The collector algorithm would explore all edges in parallel (analogous to phase 2 of the current algorithm), and mark any object with an inbound unhinted edge. This would have the effect of leaving unmarked any object which is only directly accessible from hinted edges. As a second step (analogous to phase 3 of the current algorithm), any edges (regardless of hint) reachable from a marked object would be recursively followed and the target marked. Finally, any remaining unmarked objects would be reclaimed.

We believe that this approach is isomorphic with the algorithm for our current collector, but we have not proven this. There may be practical engineering advantages to using one scheme or the other. As an example, the edge hinting scheme separates the storage of hinting metadata and mark bits which could be advantageous for concurrent collector designs since it would remove the need to coordinate hints with the marking phase. (A tagged pointer could be used to combine the hint flag and the reference into one pointer sized storage location.) On the other hand, the edge hinted algorithm has to traverse the edges twice even in the best case.

Relation to weak pointers It is interesting to consider how an edge hinted collector compares with the traditional weak pointer construct which is supported by many reference counted and garbage collected systems. A *weak pointer* is a pointer which does not itself establish reachability of the target object. If there are no normal (a.k.a. strong) pointers to the object, the object will be deallocated and the weak pointer will (atomically) be made null.

The key differences between a weak pointer and a hinted pointer is that the weak pointer is *strictly* a local decision. On the other hand, a hinted edge contributes to a global decision about the reachability of a *subgraph (not only an object)*. It is unclear whether one approach is strictly better than the other. Since this is nothing intrinsically conflicting about the two, it may be desirable to support both in a given language.

Retain Hints - Inverting the Problem

Rather than having a programmer give hints about when objects should be deallocated, we could instead ask a programmer to specify when an object should be specifically retained. In the case of exact hints, these two framings are obviously equivelent, but the effect of incomplete hints is interestingly different.

The expected use of retain hints would be to provide hints only for large long lived data structures. The hope would be that starting with these objects marked would enable the collector to avoid repeated exploration of large sections of the heap graph on every collection. Consider a program which uses a long link list to manage a queue between a producer and a consumer. By pre-marking every node in the queue - which should be simple for the implementer of the queue - the collector could completely skip the traversal of the list and instead skip directly to exploring objects reachable from each of the items in the queue.

The collector for a retain hint based system would function in a manner largely similar to phase 3 of the current algorithm except that all heap blocks would need to be explored, not simply those with hints. The retain hints could be viewed as essentially jump starting a standard traversal collector with a much larger root set. As with a deallocation hint collector, the collector would be sound (i.e. any object reclaimed was actually unreachable), but not complete (i.e. not all unreachable objects might be reclaimed). The correctness argument follows from the same property of reachability as our current hinted collectors.

A retain hint collector has one key advantage over the class of hinted collectors described in this thesis. Like a hinted collector, a retain hint collector could still end up exploring part of the unreachable graph. Unlike a hinted collector, this *could only happen if an inaccurate hint was given* (i.e. an object marked for retention became dead without having the hint removed), *not* from missing hints. The downside is that the portion of the unreachable graph explored is not limited to the nodes hinted like in a hinted collector.

It is unclear whether a retain hint collector would be more profitable than our current hinted collector design; there are far too many trade-offs to predict with any certainty. We suggest this would be an interesting and potentially fruitful topic for future work.

Conclusion

We have proposed a new approach to the classic problem of automatic memory management. By allowing users to provide hints about object deallocations, we are able to simplify the task that a collector must solve. As we have shown, this leads to better parallel asymptotic bounds and practical performance improvements.

We presented two collector implementations – one serial, one parallel – which are both practical and efficient. On a collection of benchmarks and one case study, we are able to show reductions in pause time of up to 10-20% for the serial collector. For the parallel collector run with four marker threads, we demonstrated pause times which are competitive with the baseline implementation for all benchmarks considered. For one benchmark (`perlbench`), we demonstrate a 75% (60%) reduction in worst case pause time over the parallel (serial) traversal collector. In this case, the parallel traversal collector is slower than the serial version by a large margin; our hinted collector does not exhibit this behavior.

We used a collection of standard benchmarks and a case study to assess the practical leak rate implied by requiring hints to reclaim an object. As expected, the actual rate of leaked memory was low at less than 5% for most benchmarks. Such a low rate could be easily addressed by pairing a hinted collector with a more standard backup collector.

In an expanded discussion section, we highlight the importance of a well chosen representation for hint metadata, the software engineering implications of hinted collections, and how our technique relates to others for safe memory management. Finally, we close with an extensive discussion of possible research directions for exploring the topic of hinted collection in more depth.

Acknowledgments

I would like to thank Martin Maas, Joel Galenson, and Krste Asanović for early constructive criticism of the ideas that appeared in this thesis. Many members of the Parallel Computing Laboratory (ParLab) have been subjected to half-baked versions of these ideas; thank you for your patience and questions. I would like to thank my advisor George Necula for his support.

As always, much of the credit for my successes (and none of the blame for my failures) goes to my parents Cecil and Pamela Reames. Mom, nearly 13 years after we lost you, you are still missed; I come to appreciate your wisdom more every year. Dad, I wouldn't be who I am today without you. Thanks.

To all the friends who I haven't spoken to as much as I'd like these last two years, well, now you know why. Thanks for staying in touch and keeping me sane. Chris & William, that goes particularly for you two.

Research Support

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Also supported by the National Science Foundation (Award #CCF-1017810).

Copyright Notice

This work is based in part on an earlier work: Towards Hinted Collection: Annotations for Decreasing Garbage Collector Pause Times, in Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management, ISMM 13, New York, NY, USA, June 2013. ©ACM, 2013 [32]. The major new additions are in the discussion of the parallel collector implementation and evaluation, the comparison with hazard pointers, and the chapter on future work. There have also been a variety of changes and expansions to the text of the material which appeared at ISMM 2013.

Bibliography

[1] TIOBE Programming Community Index for January 2013. `http://www.tiobe.com/index.php/content/paperinfo/tp`
Accessed: 2/1/2013.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[4] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, Feb. 1989.

[5] K. Barabash. Scalable garbage collection on highly parallel platforms. Master's thesis, Technion - Israel Institute of Technology, 2011.

[6] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 1–10, New York, NY, USA, 2010. ACM.

[7] D. A. Barrett. *Improving the Performance of Conservative Generational Garbage Collection.* PhD thesis, University of Colorado at Boulder, 1995.

[8] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[9] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 153–164, New York, NY, USA, 2002. ACM.

[10] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[11] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 59–64, New York, NY, USA, 2000. ACM.

[12] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, Sept. 1988.

[13] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM.

[14] C.-Y. Cher, A. L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 199–210, New York, NY, USA, 2004. ACM.

[15] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 480–491, New York, NY, USA, 2007. ACM.

[16] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 138–149, New York, NY, USA, 2006. ACM.

[17] S. Cherem and R. Rugina. Uniqueness inference for compile-time object deallocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 117–128, New York, NY, USA, 2007. ACM.

[18] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM.

[19] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[20] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. ACM.

[21] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems*, LCTES '03, pages 69–80, New York, NY, USA, 2003. ACM.

[22] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 48–48, 1997.

[23] R. Garner, S. M. Blackburn, and D. Frampton. A comprehensive evaluation of object scanning techniques. In *Proceedings of the Tenth ACM SIGPLAN International Symposium on Memory Management, ISMM '11, San Jose, CA, USA, June 4 - 5*, jun 2011.

[24] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: a static analysis for automatic individual object reclamation. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 364–375, New York, NY, USA, 2006. ACM.

[25] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 115–124, New York, NY, USA, 2008. ACM.

[26] M. Maas, P. Reames, J. Morlan, K. Asanović, A. D. Joseph, and J. Kubiatowicz. GPUs as an opportunity for offloading garbage collection. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 25–36, New York, NY, USA, June 2012. ACM.

[27] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[28] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[29] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[30] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 1–11, New York, NY, USA, 2007. ACM.

[31] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM.

[32] P. Reames and G. Necula. Towards Hinted Collection: Annotations for decreasing garbage collector pause times. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, New York, NY, USA, June 2013. ACM.

[33] F. Siebert. Limits of parallel marking garbage collection. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 21–29, New York, NY, USA, 2008. ACM.

[34] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.