Towards Hinted Collection *

Annotations for decreasing garbage collector pause times

Philip Reames
University of California, Berkeley
reames@cs.berkeley.edu

George Necula
University of California, Berkeley
necula@cs.berkeley.edu

Abstract

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 collector algorithm uses these hints to identify a subset of unreachable objects with both better parallel asymptotic complexity and practical performance. On some benchmarks, our prototype collector implementation achieves 10-20% pause time reductions. We close with a discussion of the design trade-offs inherent in our approach and lessons to be learned from our collector.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection); D.3.3 [*Language Constructs and Features*]: Dynamic storage management

Keywords hinted collection, deallocation hint, memory management, parallel garbage collection, mark and sweep

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 paper, 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.

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

* 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).

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

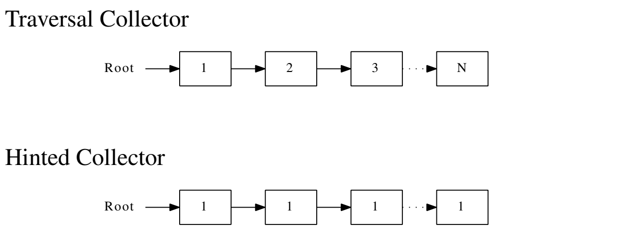

Figure 1. Minimum number of parallel steps required by a traversal collector and a hinted collector to explore a long linked list.

is live during the collection (Figure 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 [4, 5, 25]. Removing this ordering dependency is a potentially profound change.

We present a working prototype of a hinted collector to illustrate its feasibility 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 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. 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 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 accurate hints - is asymptotically faster in parallel settings. When given inaccurate 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 highlight lessons learned with the current collector and propose a modified hinted collector design so inspired.

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. Tracing 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, garbage collectors use some traversal algorithm for solving graph reachability.

2.1 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.

In this paper, 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 [5]. 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.

Throughout this paper we will use $V_{reachable}$ and $V_{unreachable}$ to describe the set of vertices reachable and unreachable by a traversal. These are not known a priori, but are useful for analysis purposes. An edge is reachable if the source vertex is reachable.

2.2 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:

- 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 [7, 10, 17].
- 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 [4, 5, 20].
- Sub-dividing the heap into sections (as in generational and region-based collectors) 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 pauses 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 [1]. 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 accurate deallocation hints.

3. A Hinted Collector

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 [7]) 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.

3.1 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 3.3)

3.2 The Abstract Algorithm

The hinted collector marking algorithm (given in Figure 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 accurate, 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 4.2. 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.

Phase 1 In the first phase, any unhinted objects are marked. Hinted objects in the root set are also marked. A key observation

```
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 2. Hinted collection algorithm discussed in Section 3.2.

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 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.

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.

Phase 3 In the third phase, any objects reachable from the previously marked 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.

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 4.3. 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.

3.3 Asymptotic Scalability

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. We note that these terms are usually incomparable with the terms for reachability. We term an individual hint *accurate* if the object so hinted is unreachable. We describe an unhinted unreachable object as having a *missing* hint.

[1] 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.

	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. 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.

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 term a set of hints which is fully accurate and with none missing to be *exact*. 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. A detailed analysis of each phase of the algorithm is available in Appendix A and a summary of these results in Table 1.

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 inaccurately hinted or reachable from objects with missing hints*.

3.4 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 traversal. Each object reachable from the roots must be reachable (without passing through a marked object) from at least one object which would be placed on the mark stack as a result of Phase 2 and Phase 3 combined. To summarize this point, it suffices to say that inaccurate hints are safe for correctness, if not necessarily performance.

4. A Practical Serial Collector

Taking the algorithm from the previous section, we implemented a hinted collector for C and C++ programs.[2] 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. 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.

4.1 Platform

We have modified the Boehm-Demers-Weiser [8] 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. There is also a parallel mark-sweep implementation available; we do not compare against that in this work.

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.

4.2 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

[2] The full source code, including all microbenchmarks and test drivers, is available under a BSD-style license at `https://github.com/preames/hinted-collection`.

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 6.3.

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 [3]. 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[4], 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 3.

To avoid needing a mark stack large enough to hold every hinted object, we interrupt the scan occasionally[5] to empty the mark stack. If despite this measure, the mark stack overflows, any objects which 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.

[3] 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.

[4] 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.

[5] 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 3. Phase 3 of the hinted collector with mark stack overflow protection added. The algorithm is otherwise unchanged.

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. It is possible such a case would be triggered by imprecision in the stored hint metadata. We have not observed this to be a practical concern.

4.4 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 tree 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 4.

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 Phase two consumes the majority of runtime for the case study and microbenchmarks we have investigated.

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

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

Figure 4. Phase 1 of the concrete collector.

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, but has the benefit of being easy to audit for correctness.

While functionally correct, this version is not sufficiently fast to be competitive with the well tuned tracing collector baseline. 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 5.3, 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 completely forgo the outer loop. Depending on the benchmark, we see as much as a 30% improvement from this change alone.

  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 5. 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 would likely 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 4.3. 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

```
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 5. Phase 2 with Edge-Filtering & Object-Coalescing.

standard reachability collector, we were able to reuse many of the components of the Boehm-Demers-Weiser collector.

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.

5. Evaluation

In Table 2, 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. Preliminary results with a parallel implementation can be found in the lead author's forthcoming Master's report.

5.1 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.[6]

[6] 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**	no-oc	no-ef	range	both	base
Linked List (Dead, Hinted)	31.8 MB	1.45	2.00	**0.72**	0.15	0.30	2.05	0.15	10.10
Linked List (Dead, Unhinted)	31.8 MB	1.40	7.75	**0.18**	10.30	8.35	7.50	7.60	10.55
Linked List (Live, Hinted)	31.8 MB	11.95	12.10	**0.99**	11.70	12.30	12.25	12.30	12.00
Linked List (Live, Unhinted)	31.8 MB	12.10	7.40	**1.64**	10.70	8.85	7.25	7.05	11.10
Fan In (Dead, Hinted)	23.7 MB	0.00	0.00	**n/a**	0.00	0.00	0.00	0.10	0.00
Fan In (Live, Hinted)	23.7 MB	12.85	12.90	**1.00**	12.35	13.30	12.50	12.15	13.25
Fan In (Live, Unhinted)	23.7 MB	12.50	8.25	**1.52**	9.35	11.10	6.00	6.60	14.60
2560 x 1k element LL	88.6 MB	31.70	19.80	**1.60**	27.70	21.75	18.50	18.05	28.60
256 x 10k element LL	88.6 MB	30.95	18.90	**1.64**	27.35	21.65	18.35	18.50	29.00
1/3 Cleanup	184.6 MB	50.10	31.95	**1.57**	44.60	37.15	35.45	32.55	55.80
Deep Turnover	56.6 MB	20.50	12.30	**1.67**	15.70	16.55	10.00	10.80	16.30
Unbalanced (Live)	744.6 MB	252.85	176.50	**1.43**	223.95	192.15	174.85	174.70	240.05
Unbalanced (Partly Dead)	744.6 MB	246.30	175.25	**1.41**	223.40	187.30	175.90	174.90	237.30

Table 2. Average mark times (in ms). "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. "no-oc" is a variant without object combining, but with header edge-filtering. "no-ef", "range", and "both" are variants of the hinted collector with no edge-filtering, range-filtering, and both header and range filtering respectively; all three use object-combining. "base" is a variant with neither edge-filtering or object-combining and illustrates well the importance of the two optimizations.

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.

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.

5.2 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.

- **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 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.

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

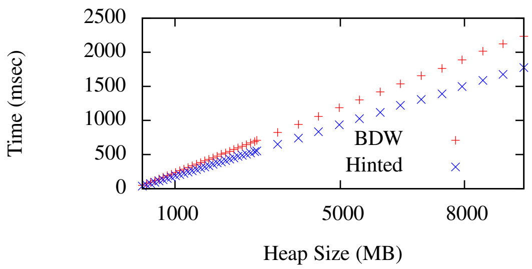

Figure 6. Scalability of hinted and tracing collectors given 10% deallocation rate over range 200 MB to 10GB of allocated data.

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.

5.3 Edge-Filtering and Object-Combining

We investigated the impact of the edge-filtering and object-combining optimizations - introduced in Section 4.4 - 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).[7] 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.

As can be seen from the results table, 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 paper use the header implementation of edge-filtering and object-combining.

5.4 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. In Figure 6, you can see the results of an experiment which varied the heap size from 200 MB to nearly 10 GB while deallocating 10% of the heap at each size. The hinted collector outperforms the standard collector across the entire range. This result is 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. The exact thresholds are dependent on the structure of the benchmark, but the general pattern was observed across multiple workloads.

[7] 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.

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

Figure 7. Statistics (across 34861 unique collections) on pause times observed for hinted and tracing collectors for the case study.

5.5 SPEC 2006

Full results from running a subset of the C programs in the SPEC 2006 benchmark suite can be found in Table 3 and 4. Table 3 focuses on the pause times observed, while Table 4 focuses on the amount of 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 backup collector reclaims only a small amount more. It appears that libquantum is retaining a reference to dead data past the last collection cycle. Across all benchmarks, the largest amount of memory leaked by the hinted collector is 13.15%. Most of the other benchmarks are in the 1-5% range. 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 remove potential timing differences between runs, but 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.

5.6 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 Table 7, the hinted collector reduced the maximum pause time observed by 12% and 99th percentile pause by 18%. 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.

6. Discussion

6.1 Memory Leaks

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

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

Table 3. 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.

benchmark	hinted	reclaimed	% rec	leaked	% leaked
bzip	0	0	n/a	0	n/a
cactusADM	7030985408	7000243504	99.56	23867552	0.34
calculix	64077201264	62473916656	97.50	1137503776	1.79
gobmk	2893063104	2892301136	99.97	1761104	0.06
gromacs	14008320	13832240	98.74	621120	4.30
h264ref	3786058800	3411968688	90.12	284106672	7.69
hmmr	8091643648	7898631712	97.61	227583184	2.80
lbm	0	0	n/a	0	n/a
libquantum	2628571280	540985888	20.58	67455024	11.09
mcf	0	0	n/a	0	n/a
milc	260674870560	252234544432	96.76	3103414976	1.22
perlbench	104360698480	100269287632	96.08	15176672352	13.15
sjeng	0	0	n/a	0	n/a
sphinx3	48729799008	48708420560	99.96	101248256	0.21

Table 4. 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).

- 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. As we saw with our case study in Section 5.6, very little memory is leaked in practice. We have not attempted to break down the contributing causes.

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 over time 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.

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. Alternatively, one could use a hinted collector as the stop-the-world fallback for a concurrent collector; this would likely reduce stop-the-world pause time when the collector could not keep up. We have not explored the design space of possible pairings, and suggest this would be a profitable area for future work.

6.2 Manual Reasoning

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. 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.

One possibility we would like to explore is how deallocation hints might affect efforts on compile-time object deallocation [11, 12, 18]. 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*. 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.

6.3 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 4.2, 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.

The issue is that phase 3 of the collector algorithm is no longer something which runs only when the user gives inexact hints. 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. Another downside, is that the current behavior potentially endangers the tune-ability that is so attractive about a hinted collector.

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. Alternatively, one could take advantage of our tolerance of approximation by using a data structure such as a bloom filter to store an approximate set of deallocation hints.

When 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 phase 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. During hinted collection, this flag could be restored if all objects in the page become marked, potentially improving the efficiency of the edge-filtering optimization within a single collection. The performance of this would need to be explored. An alternate approach would be to use multiple sets of mark bits similar to how one might support concurrent marking and sweeping as in [14].

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.

6.4 Further Discussion

In addition to the serial implementation presented in this paper, we have completed a parallel implementation. Since our parallel implementation is not yet mature, we have chosen not to present those results here. We note that an extended discussion of the implications of hinted collection and full copies of the preliminary performance results for the parallel implementation can be found in the lead author's forthcoming Master's report.

7. Related Work

Since we covered much of the related work in garbage collection in Section 2.2, we do not restate it here. Many of the general topics are well covered in Wilson's survey [26].

The only work we know of that directly addresses the fundamental scalability of parallel collection is that of Barabash & Petrank [4, 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 [8]. The BDW 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 [9, 13, 21]. The most relevant for our own work are attempts to create memory allocators which can transparently tolerate deallocation errors in production environments through over-provisioning with randomized object placement [6, 19, 22], type-specific pool allocation [2, 15, 16], checkpointing and environmental perturbation [24], or runtime patching with probability based identification [23]. 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.

8. 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 a collector implementation which is 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%. Remarkably, the current implementation takes advantage of only a small amount of the parallelism enabled by the new algorithm; future collectors could easily outperform this by a large margin.

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. Finally, we closed with a discussion of the lessons learned from the current collector regarding metadata storage and possible directions for followup work.

Acknowledgments

We would like to thank Martin Maas, Joel Galenson, and Krste Asanović for early constructive criticism of the ideas that appeared in this paper. We would also like to thank the anonymous reviewers for their comments.

References

[1] TIOBE Programming Community Index for January 2013. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` 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, 2010.

[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, 2009.

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

[5] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, 2010.

[6] 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, 2006.

[7] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, 2000.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, Sept. 1988.

[9] 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, 2012.

[10] 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, 2004.

[11] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, 2006.

[12] S. Cherem and R. Rugina. Uniqueness inference for compile-time object deallocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, 2007.

[13] 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, 2007.

[14] 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, 2005.

[15] 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, 2003.

[16] 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, 2006.

[17] 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.

[18] 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, 2006.

[19] 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, 2008.

[20] 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, June 2012.

[21] 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, 2010.

[22] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[23] 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, 2007.

[24] 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, 2005.

[25] F. Siebert. Limits of parallel marking garbage collection. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, 2008.

[26] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, 1992.

A. Per Phase Asymptotic Analysis

Phase 1 has a serial complexity of O($|V_{unhinted}|$) since it must consider every unhinted vertex. 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).

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($\max_{V \in V} indegree(v)$)).

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. 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. The parallel complexity is O(D_{hinted}) as P goes to infinity.

As a reminder, if the set of hints is exact phase 3 does not execute. If the given hints were only close to 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.