Connectivity-Based Garbage Collection[*]

Martin Hirzel
University of Colorado
Boulder, CO 80309
hirzel@cs.colorado.edu

Amer Diwan
University of Colorado
Boulder, CO 80309
diwan@cs.colorado.edu

Matthew Hertz
University of Massachusetts
Amherst, MA 01003
hertz@cs.umass.edu

ABSTRACT

We introduce a new family of connectivity-based garbage collectors (CBGC) that are based on potential object-connectivity properties. The key feature of these collectors is that the placement of objects into partitions is determined by performing one of several forms of connectivity analyses on the program. This enables partial garbage collections, as in generational collectors, but without the need for any write barrier.

The contributions of this paper are 1) a novel family of garbage collection algorithms based on object connectivity; 2) a detailed description of an instance of this family; and 3) an empirical evaluation of CBGC using simulations. Simulations help explore a broad range of possibilities for CBGC, ranging from simplistic ones that determine connectivity based on type information to oracular ones that use run-time information to determine connectivity. Our experiments with the oracular CBGC configurations give an indication of the potential for CBGC and also identify weaknesses in the realistic configurations. We found that even the simplistic implementations beat state-of-the-art generational collectors with respect to some metrics (pause times and memory footprint).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors–memory management (garbage collection)

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Connectivity based garbage collection

1. INTRODUCTION

One of the most useful features of modern object-oriented programming languages is garbage collection.[1] GC reduces debugging time because it eliminates certain kinds of memory related bugs. In addition, GC leads to cleaner code since code does not need to be cluttered with memory management concerns (e.g., who will free up the object pointed to by an argument?). This paper proposes and evaluates *connectivity-based garbage collection* (CBGC), a new family of garbage collection algorithms. Like prior garbage collectors CBGC uses object connectivity to determine which objects may be reclaimed. Unlike prior garbage collectors, CBGC also uses connectivity information to partition objects and to decide which objects to collect.

Most modern garbage collectors improve performance by first partitioning the objects and then only collecting a subset of the partitions at any one time. For example, generational garbage collectors partition the heap according to an object's age [44] and focus their efforts on the partitions containing the youngest objects. Since most objects tend to die young, generational collectors often perform well. If, however, an application creates many long-lived objects, they may perform poorly [21]. Thus, it is worthwhile to explore other partitionings that may be more stable across diverse object lifetime distributions. Since garbage collectors already use connectivity to determine which objects are garbage, partitioning objects by their connectivity seems a natural choice. Indeed, our prior work indicates that connectivity information is effective in predicting when objects die [24].

Exploiting this insight, we propose the CBGC family of garbage collectors. To determine the connectivity of objects, CBGC uses compiler analyses. These analyses are conservative, i.e., they determine the absence of a pointer between two partitions only if they can prove that such a pointer cannot exist in any run of the program. These analyses enable CBGC to collect only certain subsets of the partitions at any given collection (much like generational collectors, but without the need for write barriers).

Broadly speaking there are three aspects of garbage collector performance: (i) cost in time, (ii) cost in space, and (iii) pause time. Cost in time includes time spent on mem-

---

[1]Hereafter, we will often use the abbreviation "GC" for "garbage collection".

ory management tasks (whether in the garbage collector or the application) as well as memory system costs [42]. Cost in space refers to the amount of space required by an application when running with a particular garbage collector. Finally, pause time refers to the delays the application experiences when it must wait for GC. CBGC can improve upon existing GC algorithms in each of the above performance aspects. First, using connectivity information, CBGC can focus its collection efforts on the partitions where the least amount of work will yield the most benefit (i.e., free the most memory). Thus, CBGC can have a low cost in time. Second, and unlike many other collectors, CBGC does not perform full heap collections (except in pathological cases), and thus it can have a smaller memory footprint than other copying collectors. Third, since CBGC opportunistically performs partial collections (i.e., picks those partitions that will free the most memory with the least amount of work), it can have short pause times.

For this work, we evaluated CBGC using a trace-driven simulator with a variety of traces drawn from the SPECjvm98 [36] and Java Olden [10] benchmark suites, as well as SPECjbb2000, the Ipsixql XML document query system, and the Xalan XSLT tree transformation language processor. We obtained traces from Jikes RVM [2] and our implementation of the Merlin trace generator [22].

The key contributions of this paper are as follows:

- A novel family of GC algorithms based on object connectivity;

- A detailed description of an instance of this family;

- An empirical evaluation of a range of configurations for CBGC using a GC simulator. Using a simulator for evaluation allows us to easily experiment with a wide range of configurations, ranging from simple and realistic ones to ones that use oracles. Our experiments with configurations that use oracles determines the potential of CBGC and also pinpoints weaknesses in our realistic configurations. Our results show that even the simplistic non-oracle-based configurations outperform state-of-the-art generational garbage collectors with respect to pause times and footprint.

The remainder of the paper is organized as follows: Section 2 describes a framework for our family of connectivity-based garbage collectors. Section 3 describes an instance of this framework in detail, providing a concrete example of a connectivity-based garbage collector. Section 4 reveals other algorithms in the CBGC family. Section 5 describes the infrastructure we use to evaluate the efficacy of CBGC and Section 6 presents the results of our evaluation. Section 7 distinguishes our contributions from prior work and Section 8 concludes.

2. THE CBGC ALGORITHM FAMILY

This section introduces our CBGC family of garbage collection algorithms. To provide a better understanding of how CBGC exploits the connectivity of heap objects, this section primarily focuses on an abstract CBGC algorithm leaving explicit details to the next section.

2.1 Partitioning

Based upon conservative information about object connectivity, CBGC divides the set of heap objects, O, into a set of disjoint partitions, P. A *partitioning* (m, P, E) of the objects consists of a *partition map* $m : O \rightarrow P$ and *partition dag* (P, E) (a dag is a directed acyclic graph). The partition map m associates each object $o \in O$ with its partition $m(o) \in P$. The edges E of the partition dag represent the *may-point-to* relations.

A partitioning (m, P, E) for CBGC must be *conservative* and *stable*. In a conservative partitioning, if a pointer may exist between two heap objects, then either the objects must be in the same partition or there must exist an edge between their partitions in the partition dag. Equation (1) formalizes this definition of conservatism.

$$(o_1 \rightarrow o_2) \Rightarrow \left(\begin{array}{c} m(o_1) = m(o_2) \quad \vee \\ (m(o_1), m(o_2)) \in E \end{array} \right) \quad (1)$$

In a stable partitioning, two objects that belong to the same partition at one point in time must belong to the same partition ever afterward. Thus CBGC cannot split partitions. More specifically, when (as is common in Java) a class is dynamically loaded, a connectivity analysis of the class may cause CBGC to add new partitions or merge existing partitions, but never to divide existing partitions.

Figure 1 gives an example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges. Thus m, the partition map, is implicitly defined by graphic inclusion, e.g. $m(o_1) = m(o_2) = p_1$. Because there are no cycles of partition edges, the partitions form a dag. This partitioning is conservative because there exists an edge between the corresponding partitions wherever a pointer crosses the partition boundaries. Because of possible weaknesses in the program analysis, the reverse does not have to hold. For example there is an edge (p_2, p_3) even though no object in p_2 points to an object in p_3.
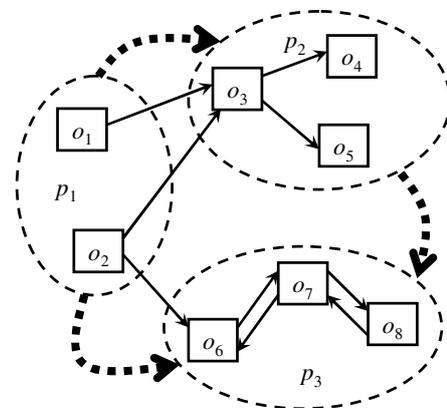
Figure 1: Example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges.

2.2 Partial GC

A *partial garbage collection* is a GC of only a portion of the heap, such as when one or more young generations are collected in a generational collector.[2] Because they do not examine the entire heap, partial GCs usually take less

---

[2] A related, but different, term is incremental GC. Partial

time than full GCs, improving responsiveness. A partial GC typically focuses on the parts of the heap where a lot of garbage can be reclaimed at low cost, thereby improving program throughput. CBGC is normally able to perform only partial GCs and performs full GCs only in pathological cases.

To understand how CBGC performs partial GCs we first review the *tricolor* abstraction [15]. In this abstraction, the GC is presented as performing a *reachability traversal* with the colors encoding where the traversal has visited. Each object has one of three colors: *black*, meaning the traversal has visited the object and its immediate successors; *white*, meaning the traversal has not visited the object; and *gray*, meaning the traversal has visited the object, but may not have visited some of its immediate successors yet. In the end, there are no gray objects anymore. Black objects are reachable and survive, white objects are unreachable and can be reclaimed.

A collector in the CBGC family performs a partial GC in the following steps:

1. Choose a set C of partitions.
   CBGC chooses a set $C \subseteq P$ of partitions that is closed under the predecessor relation, i.e. for each chosen partition, all its predecessors are chosen as well $(q \in C \land (p, q) \in E \Rightarrow p \in C)$.

2. Color all objects in C white.
   Nothing has been visited yet. The reachability traversal will only be concerned with objects in C, hence, we do not care about the color of objects in the rest of the heap.

3. Scan the program roots.
   For each pointer in a stack or global variable that points to a white object o with $m(o) \in C$, color that object gray.

4. For each chosen partition $q \in C$ in topological order:

   (a) While q contains a gray object,
       i. Pick a gray object o with $m(o) = q$.
       ii. For each pointer in a field of o that points to a white object in C, color that object gray.
       iii. Blacken o.

   (b) Reclaim all white objects in q.

The outer loop of Step 4 visits each partition $q \in C$ in topological order. Since Step 1 ensures that C is closed under the predecessor relation, the topological order guarantees that at the loop iteration for a partition q, all predecessor partitions $p \in P$ have already been visited.

LEMMA 1. *Step 4b reclaims exactly the unreachable objects in partition q.*

Proof.
   i. *After Step 4a, if an object o_n in partition q is reachable from the roots via objects in q or any predecessor partitions of q, then it is black, otherwise white.* To see

GC is the opposite of full-heap GC, whereas incremental GC is the opposite of stop-the-world GC. Some partial collectors stop the world (e.g. [44]), and some incremental collectors process the full heap in every collection cycle (e.g. [6]).

this, suppose root r reaches object o_n via the pointer chain $r \to o_1 \to \ldots \to o_n$. Per induction hypothesis, the prefix $r \to o_1 \to \ldots \to o_i$ residing in predecessor partitions of q is black. Because of the tricolor abstraction, o_{i+1} is gray, and Step 4a makes $o_{i+1} \ldots o_n$ black since they reside in q.

   ii. *If an object o_n in partition $q = m(o_n)$ is reachable, then it is reachable via objects in q or in predecessor partitions of q.* Suppose o_n is reachable via the pointer chain $r \to o_1 \to \ldots \to o_n$. Let $r \to p_1 \to \ldots \to p_n$ be the corresponding partitions, where $p_n = q$. Then conservatism (Equation (1)) guarantees that $p_1 \ldots p_{n-1}$ are predecessors of q or the same as q.

Parts i. and ii. together show that after Step 4a, exactly the unreachable objects in partition q are white. Step 4b reclaims them. □

A corollary of Lemma 1 is that a partial GC reclaims exactly the unreachable objects in the chosen set of partitions $C \subseteq P$.

2.3 Opportunism

In earlier work [24], we found that connected objects die together and that lifetime and connectivity are related. CBGC exploits these properties by making some connectivity information explicit, allowing an opportunistic choice about where to collect. This is similar to how generational GC tries to exploit the hypothesis that young objects die quickly by making some age information explicit.

When CBGC performs a partial GC, it first chooses a set of partitions to collect. CBGC uses two functions in making this choice: the *estimator* estimates how many objects are dead and live in each partition, and the *chooser* chooses a set of partitions where, based on the estimates, it expects to collect a sufficient amount of garbage at low cost.

The task of the estimator is to annotate each partition $p \in P$ with two integers $dead(p)$ and $live(p)$ as shown in Figure 2.
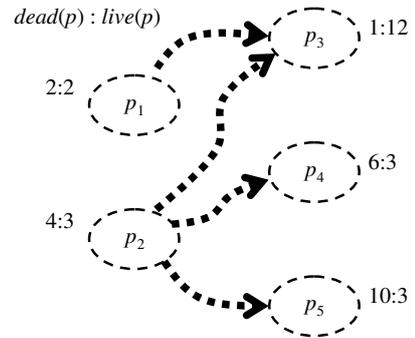
Figure 2: Example partition dag annotated by the estimator.

After this annotation, the chooser opportunistically chooses a set C of partitions that is closed under the predecessor relation. In Figure 2 it might choose $C = \{p_2, p_5\}$ because they have the best ratio of total dead to live objects $\sum dead / \sum live = (4 + 10)/(3 + 3) = 7/3$. The ratio 10/3 for the individual partition p_5 would be better, but the set $\{p_5\}$ is not closed and thus cannot be independently collected.

3

2.4 Discussion

CBGC has some inherent advantages over other collectors that perform partial GC. In CBGC's partial GC, the uncollected partitions, U, do not affect the reachability of objects in the portion of the heap chosen for GC, C. Therefore, CBGC need not track pointers from U to C, eliminating the need for a write barrier. CBGC also does not suffer from *nepotism* (nepotism is when a dead object in U falsely keeps an object in C live). Furthermore, CBGC allows early reclamation in that some objects in C can be reclaimed even before all of C is collected (Harris used a similar approach for early reclamation during full GC [20]). Early reclamation means that when collecting partition q, the memory reclaimed earlier during the collection of q's predecessors can already be reused. Finally, except for the degenerate case where one of the partitions is reachable from all other partitions, CBGC can collect all heap objects without ever performing a full GC. This substantially improves upon most other garbage collectors that require occasional full GCs for completeness (the MOS collector is an exception [25]).

In addition to the above, CBGC may make better opportunistic choices and deliver better locality than existing collectors.

3. A CBGC ALGORITHM

This section describes the CBGCHCG algorithm, which makes specific choices for each of the ingredients of CBGC described in Section 2. We will explain what the name CBGCHCG means as we go.

3.1 Partitioning

CBGCHCG uses the Harris Analysis [20] to find the partitioning (m, P, E) (the 'H' in 'HCG' stands for "Harris"). Harris demonstrated that his analysis is efficient and practical enough for use in the context of dynamic class loading and just-in-time compilation. The Harris Analysis inspects declared field types to find a may-point-to type graph (N_T, E_T) where the nodes N_T are types, and an edge $(t_1, t_2) \in E_T$ signifies that objects of type t_1 may point to objects of type t_2. It then constructs the partition dag (P, E) where the partitions P are strongly connected components (SCCs) of the type graph, and there is an edge $(p_1, p_2) \in E$ if there is an edge $(t_1, t_2) \in E_T$ in the type graph with $SCC(t_1) = p_1$ and $SCC(t_2) = p_2$. The partition map $m : O \to P$ from objects to partitions is $m(o) = SCC(type(o))$.

At runtime, CBGCHCG represents each partition by a list of blocks of a fixed power of 2 size, e.g. $2^{10} = 1024$ bytes. CBGCHCG allocates new objects at the end of the last block in the list and appends more blocks as needed. After allocation, the partition map m is implicit in the object address.

3.2 Partial GC

CBGCHCG instantiates the abstract partial GC algorithm described in Section 2.2 with a generalized version of Cheney-scan copying [12]. We picked this algorithm because it is simple and efficient and works naturally on lists of blocks.

1. Choose a set C of partitions.
   See Section 3.3.

2. Color all objects in C white.
   For each partition $p \in C$, treat the current list of blocks as from-space and create a new empty list as to-space. Objects in the from-space are white.

3. Scan roots.
   For each pointer in a stack or global variable that points to a non-forwarded object in C, copy that object to the to-space of its partition and install a forwarding pointer in the from-space. Update the root to point to the copy.

4. For each partition $q \in C$ in topological order,

   (a) Initialize the scan pointer to the beginning of the to-space. All objects between the start of the to-space and the scan pointer are black, all objects between the scan pointer and the end of the to-space are gray.
   While the scan pointer has not reached the end of the to-space of partition q,

      i. Consider the object o at the scan-pointer.
      ii. For each pointer in a field of o that points to a non-forwarded object in a from-space in C, copy that object to the to-space of its partition and install a forwarding pointer in the from-space. Update the field to point to the copy.
      iii. Advance the scan-pointer to the next object.

   (b) Reclaim the from-space of q.

3.3 Opportunism

3.3.1 Estimator

CBGCHCG uses the *combined* estimator to annotate each partition $p \in P$ with $dead(p)$ and $live(p)$ (the 'C' in 'HCG' stands for "combined"). This estimator is a hybrid estimator combining the roots estimator and the decay estimator, described below.

The *roots* estimator is motivated by our observation that objects reachable from global variables tend to be immortal, while objects reachable only from the stack tend to be shortlived [24]. The roots estimator first scans the roots to find out which partitions contain objects directly pointed to by stack variables and which by global variables. It then propagates this information over the edges of the partition dag to find out which partitions may contain objects that are reachable from stack variables, and which may contain objects that are reachable from global variables. The information is conservative: if a partition p contains objects reachable from stack/global variables, this is noted, but the reverse is not necessarily true. The roots estimator then assumes the survivor rate function

$$s_{roots} = \textbf{if}(globalsReach(p)) \textbf{ then } 90\%$$
$$\textbf{else if}(stackReach(p)) \textbf{ then } 20\%$$
$$\textbf{else } 0\%$$

The *decay* estimator assumes that the survivor rate of a partition p is an inverse exponential function of the average age, a, of its objects as shown in Figure 3. This expresses the intuition that the longer you wait, the more objects die. The literature refers to the model behind our decay estimator as the radioactive decay model. Stefanović found that none of the well-known analytical models for object lifetime distributions is completely satisfactory [41], but we were still interested in how well a simple model works for CBGC.
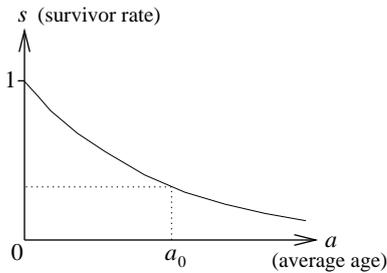
Figure 3: Decaying survivor rate $s_{decay}(a) = e^{-da}$.

For each partition p, the decay estimator maintains the observed decay factor, d, at the previous GC of p, the average age, a, of objects in p, and the total number, n, of objects in p. At each allocation, the decay estimator updates the average age: $a \leftarrow (a + timeSinceLastAlloc) \cdot (n/(n+1))$. After each garbage collection of partition p, the decay estimator updates the decay factor $d \leftarrow -\ln(S)/a$ based on the exact observed survivor rate S in p. Before the first garbage collection, the decay factor d defaults to the same constant for all partitions, e.g. 10^{-8}.

The *combined* estimator combines the roots and decay estimator. If a partition has not been collected before, it uses the roots estimator, otherwise it uses the decay estimator. This means that it does not have to use an arbitrary default decay factor, since it has an opportunity to learn a better one before using the decay estimator.

3.3.2 Chooser

CBGCHCG uses the *greedy* chooser to pick a closed set of partitions to collect based on the results of the combined estimator (the 'G' in 'HCG' stands for "greedy"). Equation (2) defines the *quality* of a set $C \subseteq P$ of partitions.

$$quality(C) = \frac{\sum_{p \in C} dead(p)}{\sum_{p \in C} live(p)} \qquad (2)$$

The numerator $\sum dead$ represents the payoff of reclaiming garbage from C, and the denominator $\sum live$ represents the cost of traversing the reachable objects in C.

The greedy chooser works as follows:

1. Initialize $C \leftarrow \emptyset$.

2. For each partition q, let $A(q) = \{p \in P \setminus C \mid p \rightarrow^* q\}$ be the set that contains all ancestors of q that have not yet been chosen. Since the ancestor relation is reflexive, q is an ancestor of itself.

3. Find the partition p with the highest $quality(A(p))$.

4. If $dead(C)$ is not yet enough to satisfy the current allocation request, or if $quality(C) < quality(C \cup A(q))$,

   (a) then $C \leftarrow C \cup A(p)$, and go back to Step 2,
   (b) else return the choice C.

4. ALTERNATIVE CBGC ALGORITHMS

Section 2 presented the family of CBGC collectors, and Section 3 described one specific member of the family. This section discusses how different choices for the ingredients of a CBGC algorithm yield different members of the family.

4.1 Partitioning

There are many ways to come up with a partitioning (m, P, E) for CBGC. The simplest possible partitioning is to have only one partition, in which case CBGC degenerates to full GC. A simple, realistic partitioning is based on the Harris Analysis [20] as described in Section 3.1. We are investigating how to postprocess the results of various other static analyses from the literature (e.g. [3, 16, 19, 27, 28, 32, 37, 45]) to obtain partitionings; some of them will be based on types, some on allocation sites.

To get a sense for whether or not more precise partitioning would improve the performance of CBGC, we also experimented with two limit partitioners: *allocsite-dynamic* and *type-dynamic*.

The initial partitions in *allocsite-dynamic* are the allocation sites. If our benchmark run contains an assignment that creates a pointer from an object created at one allocation site to an object created at another allocation site, *allocsite-dynamic* adds a corresponding edge between the allocation sites. Finally, *allocsite-dynamic* collapses SCCs of the allocation site graph to form partitions.

The initial partitions in *type-dynamic* are program object types (classes in Java terminology). If our benchmark run contains an assignment that creates a pointer from an object of one type to an object of another type, *type-dynamic* adds a corresponding edge between the types. Finally, *type-dynamic* collapses SCCs in the type graph to form partitions.

We find the *allocsite-dynamic* and *type-dynamic* partitionings by doing offline passes over the trace before simulation.

All partitionings we consider in this paper are based on strongly connected components of some form of may-point-to graph. Using finer partitions than SCCs would require some changes to the framework, e.g. introducing write barriers. Using coarser partitions would not require any changes to the framework, but would reduce the flexibility for partial garbage collections.

4.2 Partial GC

A member of the CBGC family has two choices for partial GC: how to implement the tricolor abstraction and in which order to do the reachability traversal.

There are various ways to implement the tricolor abstraction for CBGC, for example copying GC or mark-sweep GC [26].

Partial GC in a CBGC performs a reachability traversal, but Section 2.2 leaves the traversal order for this undefined. It is up to the specific algorithm to decide in which order Step 3 scans the roots and Step 4a processes gray objects. In Section 3.2 we use generalized Cheney scan, leading to breadth-first order in Step 4a. If the tricolor abstraction is based on marking, it would be natural to use a mark-stack, leading to depth-first order in Step 4a. Wilson, Lam, and Moher [47] describe alternatives for Steps 3 and 4a, in particular a hierarchical decomposition order for Step 4a. Furthermore, one can even perform Step 4a on multiple processors in parallel without need for synchronization if the involved partitions are independent.

The choices for implementing the tricolor abstraction and for the traversal order do not have to be fixed. As pointed out for example by Moss [29] and by Brooksby and Barnes [9], it is possible to manage different partitions with different tricolor abstractions. In CBGC, we could also treat

objects in the same partition differently, e.g. by copying small objects, but re-linking the blocks of old objects into their partition's to-space to avoid copying them. Furthermore, we could even change the tricolor abstraction and traversal order based on dynamic feedback, as done, for example, by Chilimbi and Larus [14].

4.3 Opportunism

4.3.1 Estimator

So far, we have investigated four estimators: roots, decay, combined, and oracle. Section 3.3.1 describes the roots, decay, and combined estimators.

In our simulator, we have also implemented an *oracle* estimator that uses the precise deathtimes obtained from Merlin [22]. The oracle estimator assumes information unavailable without a reachability traversal of the whole heap. Thus, unlike the roots, decay, and combined estimators, it is not useful in practice. However, it allows a limit study of how CBGC behaves with the most precise estimator possible.

4.3.2 Chooser

Section 3.3.2 defines the quality of a set of partitions, and describes the *greedy* chooser, a simple greedy algorithm that tries to choose a closed set with a high quality. More formally, it approximates a solution to the following problem:

> Given a partition dag (P, E) and a pair of functions $dead, live : P \rightarrow \mathbb{N}$, find a closed subset $C \subseteq P$ of partitions that maximizes $quality(C)$.

We have also invented a *flow-based* chooser that solves this problem exactly. The algorithm uses network flow [1], for details see our technical report [23]. We found the greedy chooser to be much faster while maintaining good choice quality.

One alternative for the chooser is to add more constraints, for example a lower bound on the total size of dead objects that must be reclaimed, or an upper bound on the total size of live objects that should be traversed. We have implemented this for the greedy chooser. It has a similar effect for CBGC as Barret and Zorn's dynamic threatening boundary for generational GC [7].

So far we have assumed that CBGC runs the estimator, the chooser, and the actual partial GC in that order. We followed a divide-and-conquer strategy by investigating these CBGC components separately. One could also imagine interleaving them, which may enable the estimator and chooser to make more informed decisions and may thus lead to synergy. For example, the estimator may revise its estimates based on exact survivor rates of some partitions in a GC in progress, and the chooser may revise its choice based on the updated estimates.

5. METHODOLOGY

Section 5.1 describes our traces and how we generated them. Section 5.2 describes our garbage collection simulator that consumes the traces. Section 5.3 describes how we validated our traces and simulation runs. Finally, Section 5.4 discusses the strengths and weaknesses of our simulation-based approach to evaluating CBGC.

5.1 Garbage Collection Traces

We use traces from a number of Java benchmarks to drive our simulations. These traces are chronological recordings of every object allocation, heap pointer update, and object death (when an object becomes unreachable) over the execution of a program. The traces also include an enumeration of all changed roots at each point where garbage collection can occur. Each of these events includes the information required to simulate it: object allocations include a unique identifier for the new object, its size (including a two-word header), type, and allocation site; pointer updates identify the pointer being updated and the updated value; and object deaths identify the object that becomes unreachable. The object death events are perfect in the sense that they denote the exact time when the object first becomes unreachable [22].

Table 1 lists our benchmarks. The table shows the total allocation and high watermark (maximum number of simultaneously reachable bytes) to give a feeling for the size of the runs. The benchmarks *bh*, *health*, and *power* are the only programs from the Java Olden suite of pointer-intensive kernels that exercise GC. The *deltablue* benchmark is a small constraint solver that has been translated into many languages. The benchmarks *compress*, *db*, *jack*, *javac*, *jess*, and *mtrt* are all the programs from the SPECjvm98 suite that exercise GC. The benchmark *ipsixql* is an XML database, and *xalan* is an XSLT processor; both are real-world programs, their class files take up 1,986KB and 4,200KB respectively. The *pseudojbb* benchmark is a version of SPECjbb2000 modified to perform a fixed number of transactions [8]. Finally, *null* is an empty Java program, and thus it gives an indication of how much memory the virtual machine uses just for starting up. We included it to put the data in Table 1 into perspective, but of course do not present numbers for it in Section 6.

We implemented the trace generator in version 2.2.0 of Jikes RVM [2]. Jikes RVM is a highly-optimizing compiler and run-time system for Java. It is written mostly in Java and runs on a variety of hardware platforms and operating systems. Our platform was an Intel Pentium workstation running Linux. Jikes RVM allocates its own objects on the same heap as application objects, and we treated objects from all owners uniformly in our simulations. We ran Jikes RVM with adaptive optimization system [5] enabled, and started tracing after one benchmark run had completed. Since most (if not all) of the compilation happens in the first run, tracing only the second run ensures that our traces contain events mostly from application code and not the compiler.

5.2 Garbage Collection Simulator

We developed a simulator, gcSim, to perform the experiments in this study.[3] It consists of implementations of the collectors described in this paper, supported by a number of abstractions. These abstractions include models of the root set, the heap, and individual objects and a block manager (the heap is organized as a number of fixed-sized blocks).

5.3 Validation

Garbage collectors are notoriously difficult to write and debug. While undertaking the experiments for this paper,

---

[3] http://www.cs.colorado.edu/~hirzel/gcSim

Table 1: Traces used in this evaluation.

Program	Input	Total allocation objects	Total allocation bytes	High water-mark bytes	URL
null	–	373,315	48,791,248	48,695,104	`class Null{public static void main(String[] args){}}`
power	–	1,230,662	73,228,028	49,700,876	`http://www-ali.cs.umass.edu/~cahoon/olden`
deltablue	–	1,303,984	77,502,904	49,305,572	`http://research.sun.com/people/mario/java_benchmarking`
bh	`-b 500 -s 10`	1,453,904	83,503,920	49,342,316	`http://www-ali.cs.umass.edu/~cahoon/olden`
health	`-l 5 -t 500 -s 1`	2,102,507	86,718,316	51,499,180	`http://www-ali.cs.umass.edu/~cahoon/olden`
db	`-s100`	3,807,582	133,782,036	58,714,536	`http://www.specbench.org/osg/jvm98`
compress	`-s100`	388,832	159,951,240	56,684,200	`http://www.specbench.org/osg/jvm98`
mtrt	`-s100`	7,973,471	237,305,784	59,777,468	`http://www.specbench.org/osg/jvm98`
ipsixql	`3 2`	10,089,370	351,117,828	53,827,992	`http://systems.cs.colorado.edu/colorado_bench`
jess	`-s100`	12,345,040	437,641,308	54,330,928	`http://www.specbench.org/osg/jvm98`
jack	`-s100`	14,274,816	473,120,964	55,925,844	`http://www.specbench.org/osg/jvm98`
xalan	`3 2`	7,388,779	488,960,484	85,682,372	`http://systems.cs.colorado.edu/colorado_bench`
pseudojbb	1 warehouse, 70,000 trans.	18,063,813	566,361,852	78,049,072	`http://www.specbench.org/osg/jbb2000`
javac	`-s100`	17,943,604	579,746,244	61,123,296	`http://www.specbench.org/osg/jvm98`

we found ourselves in the unenviable position of having to implement a large number of garbage collectors. When we first implemented these collectors we noticed much anomalous behavior and had a difficult time convincing ourselves of the correctness of our results. To address this, we adopted a new methodology where we incorporated significant redundancy into our implementation. The redundancy enabled us to find bugs and gain confidence in our results.

Broadly speaking a garbage collector has two parts. The first part is in the compiler and is responsible for identifying the roots. Since we were evaluating garbage collectors using trace-driven simulations, the compiler also needed to generate maps that enable listing the roots and other relevant events (such as pointer assignments) to a trace file for simulator consumption. The second part is the collector itself (which in our case is in a simulator).

To validate the trace generation, we performed periodic sanity checks: we would trigger a full heap collection in Jikes RVM and also perform a full heap traversal using the roots that were output to our trace. Any disagreement between the two indicated a bug. Since Jikes RVM handles most of Java, including threads, it was surprisingly difficult to get the root identification right. Our methodology was instrumental in identifying possible problems as soon as they arose.

To validate the simulations, we compared the dead objects found by a GC simulator to the precise deathtime information obtained from our implementation of Merlin [22]. Any disagreement between the two indicated a possible bug. We found a number of bugs using this methodology.

5.4 Strengths and Weaknesses of our Methodology

Using a simulator to evaluate Cbgc had some advantages over a full implementation: it allowed us to abstract from implementation details, it allowed us to compare Cbgc to other collectors in a controlled environment, and it allowed us to experiment with various Cbgc algorithms, some of which are not possible to implement in practice, but are interesting for limit evaluation.

There are also drawbacks to not using a full implementation in a Java virtual machine. The most important is that simulation can not give us concrete timing numbers. Another drawback is that we have no cache-level locality numbers. However, previous work has shown that simulators can be useful for GC research: the older-first collector

was first evaluated using a simulator similar to ours [40], and the results carried over to the real implementation [39].

6. RESULTS

Broadly speaking, there are three aspects of performance for evaluating a garbage collector: (i) cost in time, (ii) cost in space, and (iii) pause times. We now compare a range of Cbgc collectors to each other and to the Semispace and Appel garbage collectors, with respect to these three performance aspect. Semispace is a copying garbage collector that does not use any partitioning, but instead performs only full GCs. Appel [4] is a generational collector with two partitions: a nursery partition containing all objects allocated since the last GC, and a mature partition containing the remaining objects. Prior work has found Appel to be one of the best performing generational collectors [8].

For Cbgc, we examined several different partitionings, estimators, and choosers. We abbreviate the configurations with Cbgc*pec*, where

- $p \in \{H, T, A\}$ is the Harris, Type-dynamic, or Allocsite-dynamic partitioning (Section 4.1);

- $e \in \{D, R, C, O\}$ is the Decay, Roots, Combined, or Oracle estimator (Section 4.3.1); and

- $c \in \{G, F\}$ is the Greedy or Flow-based chooser (Section 4.3.2).

Figure 4 shows the relative strength of our various configurations. Two configurations are connected by an edge if the upper configuration is theoretically "better" than the lower one. For example, CbgcAOF is theoretically better than CbgcAOG, since CbgcAOF uses the flow-based chooser, which is optimal given a particular partitioning and estimator, whereas CbgcAOG uses the greedy chooser, which is only approximate.

Components in Figure 4 that are not realistic (e.g., they require information from a program run) are underlined. For example, *T*DF uses an unrealistic type-dynamic partitioning and so we underline the 'T'. We include the unrealistic configurations to allow us to better explore the design space of the Cbgc family.

Sections 6.1, 6.2, and 6.3 evaluate the cost in time, cost in space, and pause times for both CbgcAOG and CbgcHCG. We chose to evaluate these configurations in Sections 6.1, 6.2, and 6.3 because CbgcHCG is a fully realistic configuration, showing what can be achieved in practice, whereas
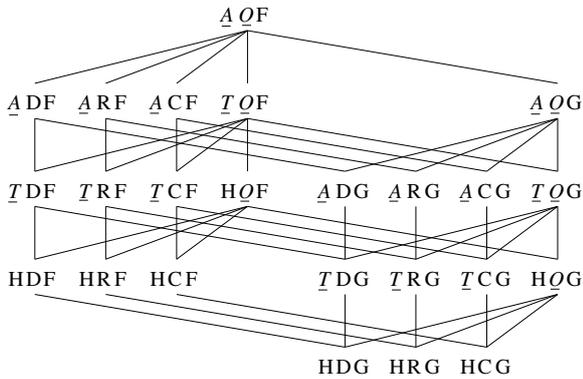
Figure 4: CBGC Configurations. Higher positions represent a "stronger" or more optimal configuration. The configurations with an '$\underline{A}$', '$\underline{T}$', or '$\underline{O}$' use information from a benchmark run and are therefore not realistic.

Figure 5: gcWorkPerTime (bytes copied / bytes allocated).

CBGCAOG is one of our strongest (albeit unrealistic) configurations, presenting the full potential of CBGC.

Section 6.4 explores many of the other points in the design space in order to better understand the tradeoffs in CBGC algorithms.

In our experiments, and unless otherwise indicated, the collectors use a heap size of three times the high watermark of the benchmark, and a block size of 1KB. Section 6.5 shows that our results generalize to different heap sizes and block sizes.

6.1 Cost in Time

We now compare SEMISPACE, APPEL, CBGCHCG, and CBGCAOG with respect to the time cost of garbage collection. Section 6.1.1 compares the amount of work each collector does during garbage collection, while Section 6.1.2 considers the other time costs of garbage collection.

6.1.1 GC Work Per Time

To compare the work done by different garbage collection algorithms during GC, we use the gcWorkPerTime metric. We define gcWorkPerTime as the total number of bytes copied in all garbage collections (work), divided by the total allocation, in bytes, of the program (time). Measuring time in bytes allocated is common in memory management research; Table 1 shows the total allocation of our benchmark programs.

As an example, if the gcWorkPerTime is 0.5, then for every 2 bytes that the application allocates, GC must perform 1 byte of copying work. Since in classical mark-sweep collectors the main work consists of marking objects and the classical LISP function for allocation is `cons`, the gcWorkPerTime metric is often called mark-cons-ratio in the literature.

Figure 5 shows gcWorkPerTime for SEMISPACE, APPEL, the simplistic CBGCHCG, and the oracle-based CBGCAOG. In this and subsequent figures taller bars indicate worse performance.

When comparing CBGCAOG to APPEL, we see that except for one benchmark (*jess*), CBGCAOG outperforms APPEL. For the benchmarks that do relatively little allocation[4] we
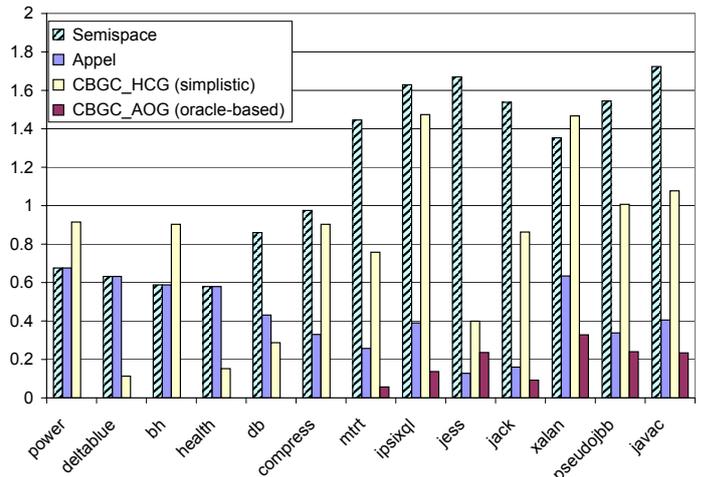
cannot see a bar for CBGCAOG! In other words, at each garbage collection CBGCAOG is able to choose only partitions where almost every object is a dead object. For the benchmarks that allocate more, CBGCAOG copies more, but typically still copies far fewer bytes than APPEL does.

On the other hand, the realistic CBGCHCG collector is usually worse than APPEL, but usually outperforms SEMISPACE. CBGCHCG performs worse than SEMISPACE for *xalan* because of weakness in the estimator (Figure 11); the combined estimator tells CBGCHCG that certain partitions have many dead objects when they do not. CBGCHCG performs worse than SEMISPACE for *bh* and *power* because these benchmarks perform only one collection each with SEMISPACE (Table 2) and thus the exact timing of the one collection ends up being significant.

6.1.2 Other Time Cost Factors

Besides the time spent in garbage collection, there are many other costs of memory management [42]. Other time costs of CBGC include time to perform the partitioning analysis upon class loading, and running the estimator and chooser before GC. Time costs present in APPEL but not in CBGC include the time to compile and execute write barriers. In addition, the two collectors may have different memory system costs (we present a preliminary exploration of these last costs in Section 6.2.1). In order to quantify these costs we need to implement CBGC, measure its actual time cost (i.e., execution time, memory stalls, etc.) and compare this to the costs incurred by SEMISPACE and APPEL for each of our benchmark programs. As the goal of the present paper is to better understand CBGC and explore the tradeoffs between different members of the CBGC family, these experiments are beyond our current scope.

6.1.3 Cost in Time Conclusions

- The oracle-based CBGCAOG usually performs much less GC work per time than APPEL. A good CBGC algorithm can potentially have lower cost in time than state-of-the-art collectors.

---

[4]The benchmarks in this and subsequent figures are ordered from left to right by increasing numbers of bytes allocated.

8

- The simplistic CBGCHCG usually performs more GC work per time than APPEL, but less than SEMISPACE. We need better realistic CBGC components to reduce CBGC's cost in time.

- There are other time cost factors besides GC work per time. The authors are working on a real-world CBGC implementation to allow more definitive comparisons.

6.2 Cost in Space

There is an obvious space-time tradeoff with garbage collectors. Increasing the memory available to run an application reduces the time spent in garbage collection. At one extreme, if an application has an infinite amount of memory, it will never need to garbage collect. Conversely, if an application has little memory, it needs to perform more collections which causes an increase in the time costs of GC.

To allow for controlled and fair comparison across garbage collectors, our experiments used a fixed heap size equal to three times the high watermark of the benchmark program. The high watermark is the maximum number of bytes that are reachable at the same time. Table 1 shows the high watermark for each benchmark. We computed these values using the Merlin perfect death time traces. We used a fixed heap size of three times this high watermark, because previous research shows it to be a heap size at which many algorithms perform well [4, 8]. In Section 6.5, we show that our results hold across a range of heap sizes.
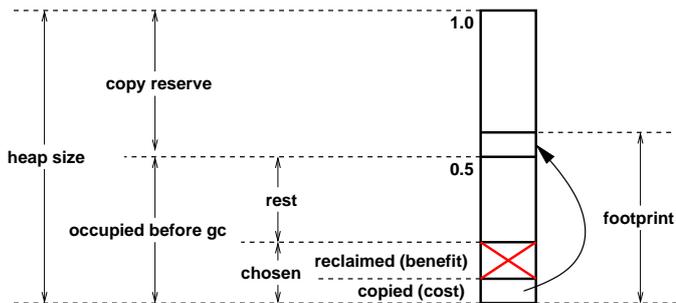
Figure 6: Heap Anatomy.

Even though the *heap size* is fixed, we can still evaluate the cost in space by measuring the *footprint* of memory in use. Figure 6 shows the space usage of garbage collectors in our simulator (for simplicity, Figure 6 shows the memory regions as contiguous; in reality, they are represented as sets of fixed-sized blocks.) During program execution, the collectors maintain half of the heap as "copy reserve" since in the worst case, all objects may survive GC. While SEMISPACE collects the entire heap at each GC, CBGC and APPEL may perform a partial GC and examine only a few partitions ("chosen"). A partial GC will use only a subset of the copy reserve. The *maximum footprint* of a collector is the maximum fraction of the heap that is simultaneously in use. A smaller footprint generally translates into better memory system performance (e.g., less paging).

6.2.1 Maximum Footprint

Figure 7 shows the maximum footprint for SEMISPACE, APPEL, the realistic, but simple CBGCHCG, and the oracle-based CBGCAOG.

Figure 7 shows that the realistic CBGCHCG consistently has a smaller footprint than APPEL. For some benchmarks

(e.g., *jess*) the footprint of CBGCHCG is much smaller than that of APPEL.

The oracle-based CBGCAOG consistently has a much smaller footprint than APPEL. Oftentimes, the footprint of CBGCAOG is close to 0.5 indicating that CBGCAOG hardly uses the copy reserve at all.
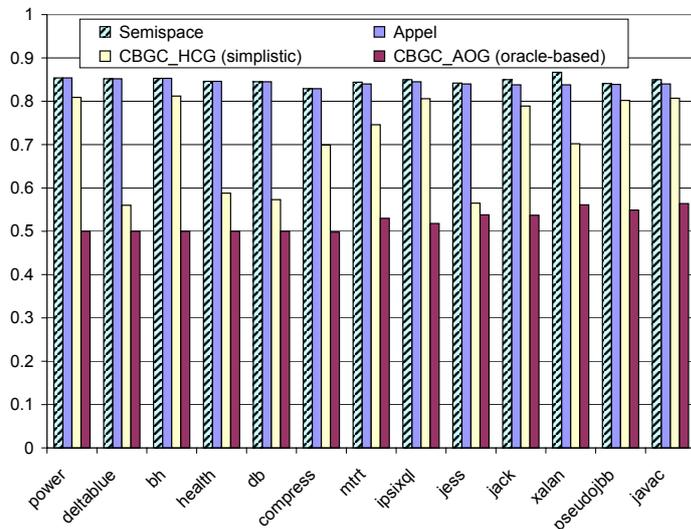

Figure 7: maxFootprint (maximum footprint / heap size in bytes).

There are two reasons why CBGC has such a good maximum footprint. First, we never observed the pathological case where it would need to do the equivalent of a full GC in APPEL. Second, it allows early reclamation and reuse of memory while a GC is still in progress as discussed in Section 2.4. APPEL, on the other hand, usually has a maximum footprint close to 85% of the heap size. This is because most of the immortal data of a benchmark is allocated upfront and survives the first garbage collection, where APPEL's flexible-sized nursery occupies 50% of the heap. The immortal data includes the runtime system of the virtual machine, the application stacks, and compiled methods.

Both CBGC and APPEL may suffer from some amount of internal fragmentation. One might have expected this to be worse for CBGC than for APPEL, because CBGC has more partitions. The simplistic CBGCHCG usually uses around 85 partitions, and on average 3.4 partitions contain 95% of the heap objects. The oracle-based CBGCAOG usually uses around 900 partitions, and on average 13.7 partitions contain 95% of the heap objects. But as Figure 7 shows, the differences in fragmentation due to many partitions has little impact on the footprint.

6.2.2 Other Space Cost Factors

Besides the space occupied by objects, all garbage collectors maintain a number of data structures that also contribute to the space cost. For example, CBGC needs space for the partition graph and APPEL needs space for the remembered sets and the write barrier instructions (in the code). While we do not have detailed experimental results for these costs, it is our experience that the above mentioned space costs for CBGC are insignificant.

9

6.2.3   Cost in Space Conclusions

- Even the simplistic CBGCHCG has a lower cost in space than APPEL. Therefore, the paging and TLB activity of CBGC are likely to be lower than that of APPEL. That can lead to better memory system performance.

- Since CBGC has a lower cost in space than APPEL, it may enable programs to run in less memory.

6.3   Pause Times

The amount of work that a garbage collector performs during a collection determines the amount of time for which the application is paused.[5] We consider two measures for this: the amount of work the garbage collector performs on average during a collection (Section 6.3.1) and the *maximum* amount of work the garbage collector performs on any collection (Section 6.3.2).

6.3.1   Average Work Per GC

Figure 8 gives the average amount of copying performed by a collector as a fraction of heap size in bytes (avgWorkPerGc). For example, if the avgWorkPerGc is 0.01, then the average collection copies 1% of the heap size (see Figure 6).
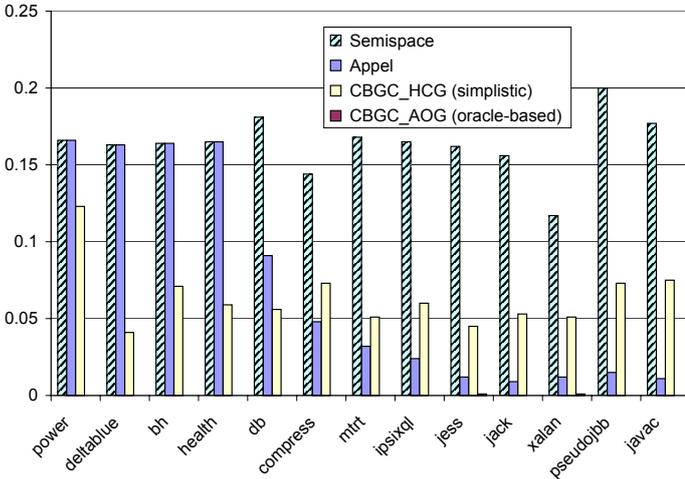

Figure 8: avgWorkPerGc (average copied / heap size in bytes).

Figure 8 shows that while APPEL usually performs well (especially for the larger benchmarks), CBGCAOG performs much better. As a matter of fact, CBGCAOG performs so well that its bars are not visible for most of the benchmarks.

The realistic CBGCHCG performs well, even outperforming APPEL for some benchmarks. As expected, SEMISPACE performs poorly, since at each collection it needs to copy all the reachable objects.

CBGCAOG has such a low avgWorkPerGc because it usually collects only partitions that contain mostly garbage *and* it performs many garbage collections (Table 2). Since there is an overhead to triggering a garbage collection (e.g., root scanning), it may be worthwhile to consider other choosers that pick more partitions to collect at each collection (Section 4.3.2).

---

[5]This is not true for incremental and concurrent collectors. In this paper we consider only stop-the-world collectors.

Table 2: Number of Garbage Collections.

Program	SEMISPACE Total	APPEL Total	APPEL Major	CBGCHCG Total	CBGCAOG Total
power	1	1	0	2	1
deltablue	1	1	0	1	55
bh	1	1	0	4	11
health	1	1	0	1	10
db	2	2	0	3	9
compress	3	3	0	5	5
mtrt	6	6	0	13	1,925
ipsixql	11	22	1	34	1,097
jess	14	15	0	17	692
jack	14	34	0	30	2,155
xalan	9	33	2	31	703
pseudojbb	12	39	1	27	2,619
javac	17	79	2	35	3,349

6.3.2   Maximum Work Per GC

Even if a garbage collector has a low average pause time, it may still be disruptive if some pauses are much longer. Thus, in this section, we consider the maximum pause time.

Figure 9 shows maxWorkPerGc for SEMISPACE, APPEL, the realistic CBGCHCG, and the oracle-based CBGCAOG. The maxWorkPerGc is the maximum number of bytes copied at any garbage collection divided by the heap size (see Section 6.2). For example, if the maxWorkPerGc is 0.18, then the largest collection copies 18% of the heap size (see Figure 6).
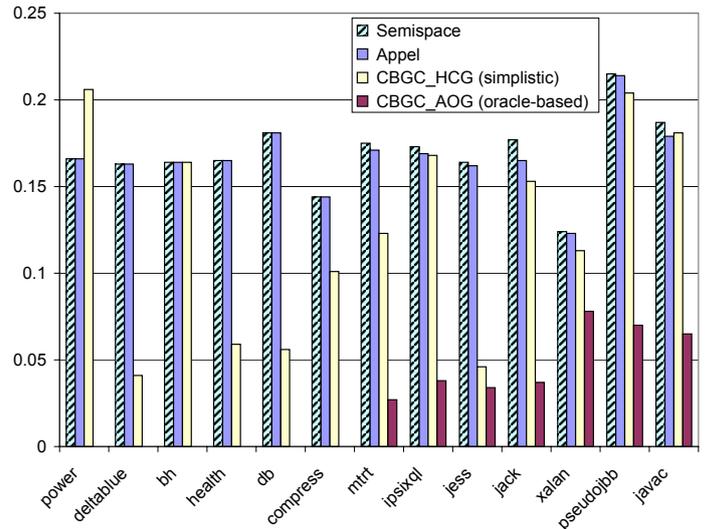

Figure 9: maxWorkPerGc (maximum copied / heap size in bytes).

From Figure 9 we see that even the realistic CBGCHCG has a better maxWorkPerGc than APPEL for all benchmarks except power and javac, where it is slightly worse. CBGCAOG is consistently the best configuration.

With respect to this metric, APPEL does not perform any better than SEMISPACE. The reason for this is that APPEL occasionally performs major collections that collect the entire heap. Also, the first collection with APPEL is effectively a full heap collection.

6.3.3   Pause Times Conclusions

- Even the simplistic CBGCHCG tends to incur less work per GC than APPEL.

- CBGC never required full-heap collections in our experiments. It always chose few enough and small enough partitions to collect. While one can construct a pathological situation that forces CBGC to do a full-heap GC, we never observed that in practice. APPEL, on the other hand, usually does full-heap GCs in long runs.

6.4 Exploring the CBGC Design Space

So far, we looked at whether or not CBGC *can* outperform other garbage collectors. In this section, we explore the CBGC design space and try to identify weaknesses in the realistic CBGC implementations. Since cost in time appears to be the biggest challenge for the simplistic CBGCHCG (compared to APPEL, it already has low cost in space and low work per GC), this section uses cost in time to compare CBGC configurations.

Our methodology is to vary one component (e.g., partitioning) while fixing the other components at their strongest level (which may be unrealistic). This methodology allows us to evaluate how different implementations of a component perform without worrying about interactions with poor implementations of the other components. For example, a poor implementation of a partitioner may obfuscate the differences between estimators.

6.4.1 Partitionings

We now explore the partitionings described in Section 4.1. The *Harris* partitioning is realistic, but simple. The *allocsite-dynamic* partitioning is a limit study for the finest-grained partitioning one can get if all objects allocated at the same allocation site must reside in the same partition. *Type-dynamic* falls in between *Harris* and *allocsite-dynamic*.
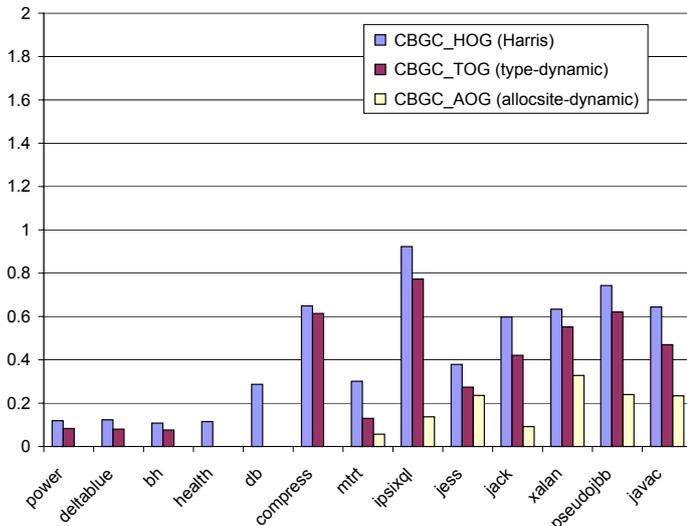

Figure 10: gcWorkPerTime (bytes copied / bytes allocated) for different partitionings. The CBGC_AOG bars are the same as in Figure 5, the other bars are new.

Figure 10 shows the gcWorkPerTime metric from Section 6.1 for CBGCHOG, CBGCTOG, and CBGCAOG. In other words, it keeps the estimator (oracle) and chooser (greedy) constant and varies the partitioner. Figure 4 shows the theoretical ordering between these alternatives.

From Figure 10 we see that as the partitioning improves, so does the gcWorkPerTime metric. Using the *type-dynamic*

partitioning is usually not much better than using *Harris*. Using the *allocsite-dynamic* partitioning often reduces the amount of GC work by a factor of 2 or more over the other partitioners.

Comparing to Figure 5, we see that the differences due to different partitionings are less than the total difference between the realistic CBGCHCG and the unrealistic CBGCAOG. Thus, the estimator also contributes to the difference.

To summarize, the quality of the partitioning makes a big difference in the performance of CBGC. Our results suggest that the next realistic partitioning that we try should be based on allocation sites. Fortunately, there are many practical pointer analyses in the literature (e.g., Steensgaard [37]) that separate allocation sites and thus may be worth trying for CBGC.

6.4.2 Estimators

We now explore the estimators described in Section 4.3.1. The *roots* and *decay* estimators are simple and realistic. The *combined* estimator is a hybrid of these two, so it is still realistic and a little more sophisticated. The *oracle* estimator is a limit study that always estimates correctly. A realistic estimator can perform at most as well as the oracle.
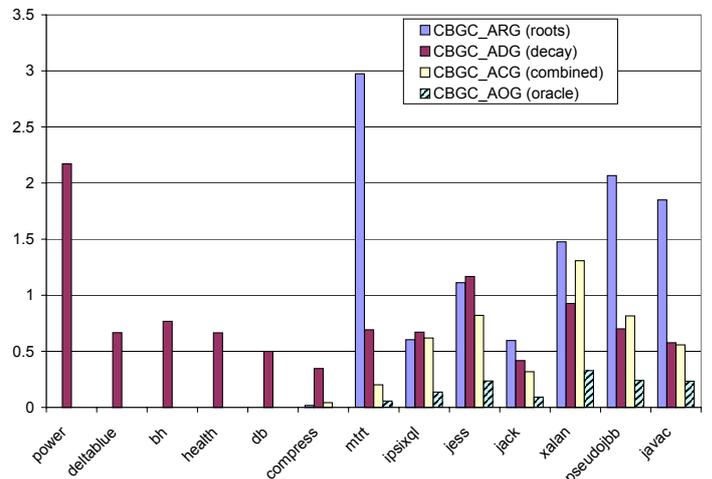

Figure 11: gcWorkPerTime (bytes copied / bytes allocated) for different estimators. The CBGC_AOG bars are the same as in Figure 5, the other bars are new.

Figure 11 shows the gcWorkPerTime metric from Section 6.1 for CBGCARG, CBGCADG, CBGCACG, and CBGCAOG. Since *decay* needs some time to learn the survival rates, it performs poorly for the smaller benchmarks that cause few collections (for those benchmarks, *decay* also has high cost in space and high pause times). The *roots* estimator performs poorly for larger benchmarks. Fortunately, the *combined* estimator combines the strengths of *roots* and *decay* to yield a much better estimator. The *oracle* estimator performs much better than the other estimators especially for the larger benchmarks.

To summarize, the *combined* estimator is the best of our realistic estimators. However, comparison with the *oracle* estimator shows that there is still much room for improvement. It may be worthwhile to incorporate profile information or information from a GC in progress into the estimator.

6.4.3 Choosers

So far, we have reported results for the *greedy* chooser instead of the optimal *flow-based* chooser. We compared the gcWorkPerTime, maxFootprint, avgWorkPerGc, and maxWorkPerGc metrics from Sections 6.1 to 6.3 for CBGCAOG and CBGCAOF, the results are almost identical. For details see the technical report describing the flow-based chooser [23].

6.4.4 CBGC Design Space Conclusions

- A partitioning needs to be more fine-grained than types for CBGC to perform well. The coarsest partitioning (putting all objects in one partition) would correspond to SEMISPACE.

- Our realistic *combined* estimator is a good start, but there is still much room for improvement with better estimators. A bad estimator can lead CBGC totally astray.

- The *greedy* chooser is simple and close to optimal.

6.5 Sensitivity to Heap Size and Block Size

So far we have reported results for a heap size of 3.0 times the high watermark and a block size of 1KB. We now consider how our results change if we use different heap and block sizes.

6.5.1 Heap Sizes

Assuming zero fragmentation, a copying collector needs at least a heap size of 2.0 times the high watermark to work, since it keeps a copy reserve. Previous research indicates that a heap size of 2.5 times the high watermark is tight and a heap size of 4.0 times the high watermark is loose [8].

Table 3 shows the gcWorkPerTime metric from Section 6.1 using three different heap sizes. We see that for all heap sizes and all benchmarks except for *jess* the relative performance of CBGCAOG and APPEL is the same. For *jess*, CBGCAOG performs worse than APPEL at heap sizes 2.5 and 3.0.

While we have not repeated all our experiments with a range of heap sizes, these results give us some confidence that our results hold for other heap sizes as well.

6.5.2 Block Sizes

All data presented in this paper uses a block size of 1KB. We chose this small block size to reduce potential internal fragmentation, especially when CBGC uses a large number of partitions. Since APPEL uses only two partitions, we do not expect it to be affected by a different block size.

We experimented with block size 4K. The numbers for each individual collector are virtually the same with block size 1K as they are with block size 4K. The small variations appear unrelated to the collector.

7. RELATED WORK

Jones and Lins [26] and Wilson [46] provide good introductions to garbage collection and also describe many techniques and algorithms that inspired CBGC.

7.1 Eliminating write barriers

CBGC does partial GC without any write barriers. Shuf *et al.* describe a simple type-based partitioning that allows eliminating some write barriers [34]. Zee and Rinard have presented an analysis for eliminating some of the write barriers for generational GC [48]. We are not aware of any other work on eliminating write barriers for partial GC.

7.2 Partitioning

CBGC partitions heap objects by connectivity to improve locality (allocate connected objects together, since the mutator is likely to access them together), to enable opportunism (do GC where you get high payoff with little effort), and to improve responsiveness (reduce pause times by avoiding full GC). Over the years, other partitioning techniques have been proposed to achieve these goals.

7.2.1 Age-based partitioning

Age-based garbage collectors partition objects by age. They assume that the survivor rate is related to object age, and exploit this by making opportunistic choices about where to do partial GC.

The weak generational hypothesis states that most objects die young, and thus generational collectors collect young objects most frequently [44]. While Appel flexibly adapts the boundary between young and old objects to achieve optimal memory usage [4], Barrett and Zorn adapt the boundary to achieve a variety of objectives [7].

Some researchers have found that the weak generational hypothesis does not allow the best opportunistic choices about where to collect. Pretenuring allocates objects that are expected to be long-lived directly into the old generation to avoid copying long-lived objects out of the nursery [13]. Older-first collection assumes that the very youngest objects are unlikely to be dead since they have not yet had time to die [40]. The Beltway collector generalizes existing copying age-based collectors by using a configurable partitioning [8].

Age-based collectors are well-studied and often successful, but we believe one should not rely solely on age, as it is not always a reliable predictor for when objects die. While we propose using heap object connectivity as the main principle for guiding garbage collection, it may also be beneficial to use CBGC just for the old generation of a generational collector.

7.2.2 Stack-based partitioning

Stack-based techniques partition objects by the stack frames that allocated them. They assume that the lifetime of objects is related to the time at which the stack frame gets popped, and exploit this by opportunistically deallocating objects together with stack frames if possible.

Some stack-based techniques rely on static information. Stack allocation is based on escape analysis and allocates the objects directly on the stack [30]. In region allocation, regions are allocated and deallocated at statically predefined program points following a stack discipline, and individual objects are allocated into their region, but deallocated only when the entire region is deallocated [18, 43].

Other stack-based techniques rely on dynamic checks. Contaminated GC tracks the lowest stack frame from which an object is reachable, and only deallocates the object when that stack frame gets popped [11]. Qian and Hendren associate a region with each stack frame and track whether objects escape from it; if so, the region is merged into the global region and handled by conventional GC, otherwise it is reclaimed *en masse* when the stack frame is popped [31].

Table 3: gcWorkPerTime (bytes copied / bytes allocated) for different heap sizes.

Program	HeapSizeFrac 2.5		HeapSizeFrac 3.0		HeapSizeFrac 4.0	
	APPEL	CBGCAOG	APPEL	CBGCAOG	APPEL	CBGCAOG
power	0.679	0.000	0.676	0.000	0.000	0.000
deltablue	0.634	0.160	0.632	0.000	0.000	0.000
bh	0.593	0.075	0.588	0.000	0.000	0.000
health	0.596	0.071	0.579	0.000	0.000	0.000
db	0.433	0.000	0.431	0.000	0.430	0.000
compress	0.384	0.020	0.330	0.000	0.312	0.000
mtrt	0.277	0.150	0.257	0.057	0.248	0.029
ipsixql	1.078	0.342	0.390	0.137	0.173	0.049
jess	0.132	0.541	0.128	0.236	0.121	0.101
jack	0.397	0.202	0.160	0.092	0.134	0.050
xalan	0.570	0.373	0.634	0.328	0.467	0.126
pseudojbb	0.641	0.539	0.338	0.240	0.187	0.084
javac	0.734	0.565	0.405	0.234	0.183	0.113

Stack-based techniques may work well for functional languages, but they often require hand-tuning by the programmer. We feel that the job of automatic memory management should be to relieve the programmer from having to pay much attention to memory, and that large object-oriented programs are unlikely to obey a strict stack discipline.

7.2.3 Other partitioning techniques

Age-based and stack-based partitioning are the most popular techniques for improving locality and allowing partial, opportunistic GC. But we are not the first to seek other partitionings to achieve this goal.

Some techniques use static analyses. Dolby and Chien analyze ownership relations between objects. If they find a relation between two objects such that the owned object has a fixed size and dies before the owner, they inline it into the owner [17]. Steensgaard describes thread-specific heaps for objects that are not shared among multiple threads [38]. Other techniques rely on profile information. Seidl and Zorn partition heap objects by the number of dynamic references to them and by their expected lifetime [33], and Shuf *et al.* partition heap objects by whether their type has many instances or not [34, 35].

As early as 1991, Barry Hayes envisioned key object opportunism, which observes when a key object dies and opportunistically reclaims the objects connected to it [21]. Key object opportunism relies on the hypothesis that connected objects die together, and we have evidence that supports this [24]. To implement key object opportunism, a GC needs to be aware of connectivity, hence we hope CBGC will bring us closer to making Hayes's vision a reality.

8. CONCLUSIONS

We introduce a new family of garbage collection algorithms (CBGC) that are based on object connectivity properties. CBGC segregates objects into partitions based on their connectivity, and also uses the connectivity information to decide which partitions to collect at each collection. CBGC is motivated by our prior work that demonstrated that there is a strong correlation between connectivity and the lifetime and deathtime characteristics of programs. We also describe a number of collectors from the CBGC family and evaluate them using a simulator.

Our results demonstrate that even a simplistic member of the CBGC family outperforms APPEL with respect to pause times and memory footprint. Our experiments with oracle-based CBGC reveal that CBGC has potential to dramatically improve upon all performance aspects of existing garbage collectors.

We are currently implementing a prototype CBGC in Jikes RVM. For the partitioning, we are developing an Andersen-style pointer analysis [3] that works with Java and dynamic class loading.

9. REFERENCES

[1] Ravindra Ahuja, Thomad Magnanti, and James Orlin. *Network Flows Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[4] Andrew Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 1989.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter Sweeney. Adaptive optimization in the Jalapeño JVM. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[6] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM (CACM)*, 1978.

[7] Dave A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Programming Languages Design and Implementation (PLDI)*, 1995.

[8] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In *Programming Languages Design and Implementation (PLDI)*, 2002.

[9] Richard Brooksby and Nicholas Barnes. The memory pool system. Unpublished paper, 2002.

[10] Brendon Cahoon. Java-Olden benchmarks. http://www-ali.cs.umass.edu/~cahoon/olden.

[11] Dante Cannarozzi, Michael Plezbert, and Ron Cytron. Contaminated garbage collection. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[12] C. J. Cheney. A non-recursive list compaction algorithm. *Communications of the ACM (CACM)*, 1970.

[13] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[14] Trishul Chilimbi and James Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

[15] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM (CACM)*, 1978.

[16] Amer Diwan, Kathryn McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 2001.

[17] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[18] David Gay and Alex Aiken. Memory management with explicit regions. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[19] Rakesh Ghiya and Laurie Hendren. Connection analysis: a practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 1996.

[20] Timothy Harris. Early storage reclamation in a tracing garbage collector. *ACM SIGPLAN Notices*, April 1999.

[21] Barry Hayes. Using key object opportunism to collect old objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1991.

[22] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *ACM SIGMETRICS*, 2002.

[23] Martin Hirzel, Harold N. Gabow, and Amer Diwan. Choosing a set of partitions to collect in a connectivity-based garbage collector. Technical Report CU-CS-958-03, University of Colorado at Boulder, 2003.

[24] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, 2002.

[25] Richard Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management (IWMM)*, 1992.

[26] Richard Jones and Rafael Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[27] Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. In *Workshop on Memory System Performance (MSP)*, 2002.

[28] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Compiler Construction (CC)*, 2003.

[29] J. Eliot B. Moss. Regions determined by kind and generation. Unpublished note, 1999.

[30] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Programming Languages Design and Implementation (PLDI)*, 1992.

[31] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In *International Symposium on Memory Management (ISMM)*, 2002.

[32] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[33] Matthew Seidl and Benjamin Zorn. Segregating heap objects by reference behavior and lifetime. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[34] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Principles of Programming Languages (POPL)*, 2002.

[35] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[36] Standard Performance Evaluation Corporation (SPEC). SPECjvm98 benchmarks. http://www.specbench.org/osg/jvm98.

[37] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, 1996.

[38] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *International Symposium on Memory Management (ISMM)*, 2000.

[39] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance (MSP)*, 2002.

[40] Darko Stefanović, Kathryn McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[41] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. In *International Symposium on Memory Management (ISMM)*, 2000.

[42] David Tarditi and Amer Diwan. Measuring the cost of storage management. *Lisp and symbolic computation*, 1996.

[43] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[44] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Practical Software Development Environments*, 1984.

[45] John Whaley and Monica Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium (SAS)*, 2002.

[46] Paul R. Wilson. Uniprocessor garbage collection techniques. Accepted for publication in *ACM Computing Surveys*.

[47] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage collected systems. In *Programming Languages Design and Implementation (PLDI)*, 1991.

[48] Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.