Early storage reclamation in a tracing garbage collector

Timothy Harris, Citrix Systems (Cambridge) Ltd,
Poseidon House, Castle Park, Cambridge, UK, CB3 0RD.
Tel: +44 1223 515010. Fax: +44 1223 359779. tim.harris@citrix.com

December 6, 1998

Abstract

This article presents a novel technique for allowing the early recovery of storage space occupied by garbage data. The idea is similar to that of generational garbage collection, except that heap is partitioned based on a static analysis of data type definitions rather than on the approximate age of allocated objects. A prototype implementation is presented, along with initial results and ideas for future work.

1 Introduction

It has been widely acknowledged that the type information present in a program written in a high level language can provide valuable opportunities for improving run-time performance.

Examples of this are pervasive. For instance, information gleaned from type analysis allows natural *unboxed* representations of primitive quantities like integers, even in the presence of polymorphism [Mor95, HU95]. The precision of pointer-aliasing analysis in an imperative language can be improved by considering the types of the pointers involved [App98]. Run-time feedback can help reduce the overhead of virtual method lookup in an object oriented language [Höl94].

This article describes a further exploitation of type information: aiding the safe and early reclamation of storage space in a garbage collected heap. In particular, the approach addresses a well-known problem with classical mark-sweep and copying garbage collectors which is that no space can be reclaimed until the entire graph of reachable objects has been examined [JL96]. By allowing earlier reclamation of storage space it may be possible to reduce the total size of the heap required with practically no additional effort on the part of the collector.

Section 2 summarizes the approach taken. Section 3 describes the implementation of a prototype collector. Section 4 presents some initial results from the prototype. Section 5 discusses the relationship to previous work. Section 6 suggests some future work and Section 7 concludes.

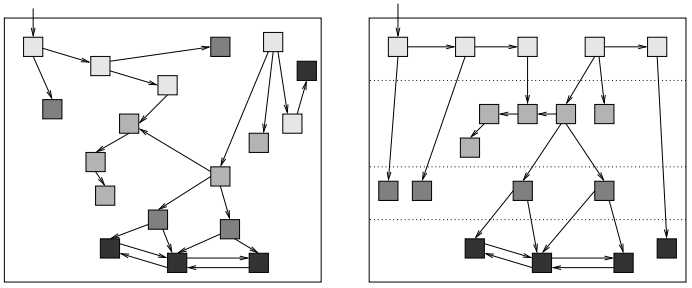

Figure 1: A traditional heap containing various objects (left) and the same objects segregated into four partitions (right). Each object is represented by a shaded box and each reference between objects is represented by an arrow. The node indicated in the top-left is the only root of this object graph. Some of the other nodes (for example the pale gray node at the top right) are unreachable.

2 Design

The idea is to separate the heap into a number of *partitions* between which a partial ordering can be established so that if $p_i < p_j$ then references may exist from p_i to p_j, but not vice versa. This means that when all of the reachable objects in partitions up to p_i have been scanned then any remaining objects in p_i are known to be garbage and the space that they occupy may be reclaimed.

The partitioning algorithm presented here is phrased in the terms of the Java programming language, for which a prototype implementation has been developed. Essentially, it uses the class definitions to derive constraints on how object references may be manipulated at run time.

2.1 Example

By way of illustration, Figure 1 shows a stylized snapshot of a traditional heap. The marked node in the top-left is a root – in this example it is the only root. The nodes represent objects and the directed edges between them represent references. Figure 1 also shows a corresponding heap in which the placement of the objects has been re-arranged and the heap has been segregated (by the horizontal lines) into four partitions. Note how inter-partition references only flow down the figure.

2.2 Object model

A simplified version of the Java object model is used for the sake of brevity. Most notably, the simplification discards the distinction between Java `interface` and `class` definitions. The interfaces implemented by a class are considered to be additional direct superclasses. The `public`, `protected` and `private` modifiers are also discarded. These simplifications remove the aspects of a class definition which are concerned with the run-time implementation of the class.

Figure 2 illustrates this by way of an example showing a set of classes modelled after the description of a *binomial heap* in [CLR90]. The exact details of this are unimportant since it is only used as a running example – the general idea is that a binomial heap holds a mapping from keys (instances of subclasses of `Comparable`) to values (instances of subclasses of `Storable`).

The gray boxes represent classes. A dashed arrow is drawn from a class to each of its direct superclasses. A solid arrow, annotated by a field name, is drawn from a class to each of the classes referenced by one of its fields. For example, `IntegerValue` is a subclass of `Storable` and the class `BinomialTreeNode` contains a field `value` of type `Storable`.

A field can contain either a reference to an instance of a subclass of the field type, or the special value `null` which does not refer to any instance. In this example the field `value` of an instance of `BinomialTreeNode` could hold a reference to an instance of `Storable` or to an instance of `IntegerValue`.

2.3 Relationships between classes

More precisely, there are two relevant relationships between classes: $<_d$, the direct-superclass relationship and $\rightarrow_r$, the direct-reference relationship.

$c_1 <_d c_2$ holds between classes c_1 and c_2 when c_2 is a direct superclass of c_1. For example `IntegerValue` $<_d$ `Storable`, but `IntegerValue` $\not<_d$ `Object`. $<_d^*$ denotes the reflexive transitive closure of $<_d$, so `IntegerValue` $<_d^*$ `Object`.

$c_1 \rightarrow c_2$ holds when a field of type c_2 is defined in class c_1. For example `BinomialTree` $\rightarrow$ `BinomialTreeNode`, but `BinomialHeapNode` $\not\rightarrow$ `BinomialTreeNode` and `DebugIntegerValue` $\not\rightarrow$ `Integer`.

From these a further relationship is established, $\Rightarrow$, which informally means *can refer to*. That is, $c_1 \Rightarrow c_2$ when an instance of class c_1 could contain a reference to class c_2. For example `BinomialTree` $\Rightarrow$ `BinomialTreeNode`, and `DebugIntegerValue` $\Rightarrow$ `Integer`. This relationship is defined recursively as follows:

$$\frac{c_1 \rightarrow c_2}{c_1 \Rightarrow c_2} \qquad \frac{c_1 <_d^* c_3 \quad c_3 \Rightarrow c_2}{c_1 \Rightarrow c_2} \qquad \frac{c_1 \Rightarrow c_3 \quad c_2 <_d^* c_3}{c_1 \Rightarrow c_2}$$

Figure 3 shows the resulting *class graph* for the example class hierarchy. An arrow is drawn from c_1 to c_2 when $c_1 \Rightarrow c_2$. The effect is to propagate the 'from' and 'to' sides of field definitions down from superclasses to their subclasses. For example, the class `IntegerValue` defines a field `value` of type `Integer`. This means that an instance of `IntegerValue`, or any subclass of `IntegerValue`, can potentially refer to any instance of `Integer`, or any subclass of `Integer`.

This static analysis is, of course, overly pessimistic. $c_1 \Rightarrow c_2$ means that an instance of c_1 *may* refer to an instance of c_2, even though this kind of relationship may never be established during the execution of a program – c_1 and c_2 may even never be instantiated. The extent to which a more precise analysis may be developed is discussed in Section 6.

2.4 Segregating the heap

We now wish to segregate the nodes in this *class graph* into *partitions* and to establish a partial ordering between partitions, as described above in the overview. That is, we wish to generate a *partition graph* in which each node represents a partition and an edge is drawn from p_1 to p_2 when objects held in p_1 may refer to objects held in p_2.

This is achieved by noting that each partition corresponds to a strongly connected component in the class graph: each partition consists of a maximal set of classes within which $c_1 \Rightarrow^* c_2$ and $c_2 \Rightarrow^* c_1$ for each pair of classes c_1 and c_2. The resulting partition graph is acyclic by its construction [CLR90].

This is illustrated in Figure 4, in which a dashed box is drawn around the classes in each partition.

3 Implementation

A prototype implementation for the Java programming language has been constructed based on Baker's treadmill collector [Bak91] integrated with version 1.1.4 of the Sun JDK. This section provides a brief overview of the treadmill collector, followed by details of how the heap-partitioning analysis is implemented and how the treadmill collector is modified to use the information that this generates.

3.1 Treadmill collector

The traditional treadmill collector shown in Figure 5 is a non-copying implementation of a two-space garbage collector. It was originally proposed to avoid the repeated copying of all live objects during each collection cycle.

Objects are organized on a circular doubly linked list which is divided into four sections:

1. *Free* section, containing blocks which are available for allocation.

2. *New* section, containing objects which have been allocated since the start of the current collection cycle. Using the traditional terminology of tri-colour marking, objects are allocated black.

3. *To* section, containing objects which have been scanned during the current collection cycle and which need to be preserved.

4. *From* section, containing objects which were allocated before the start of the current collection cycle and which have not yet been scanned.

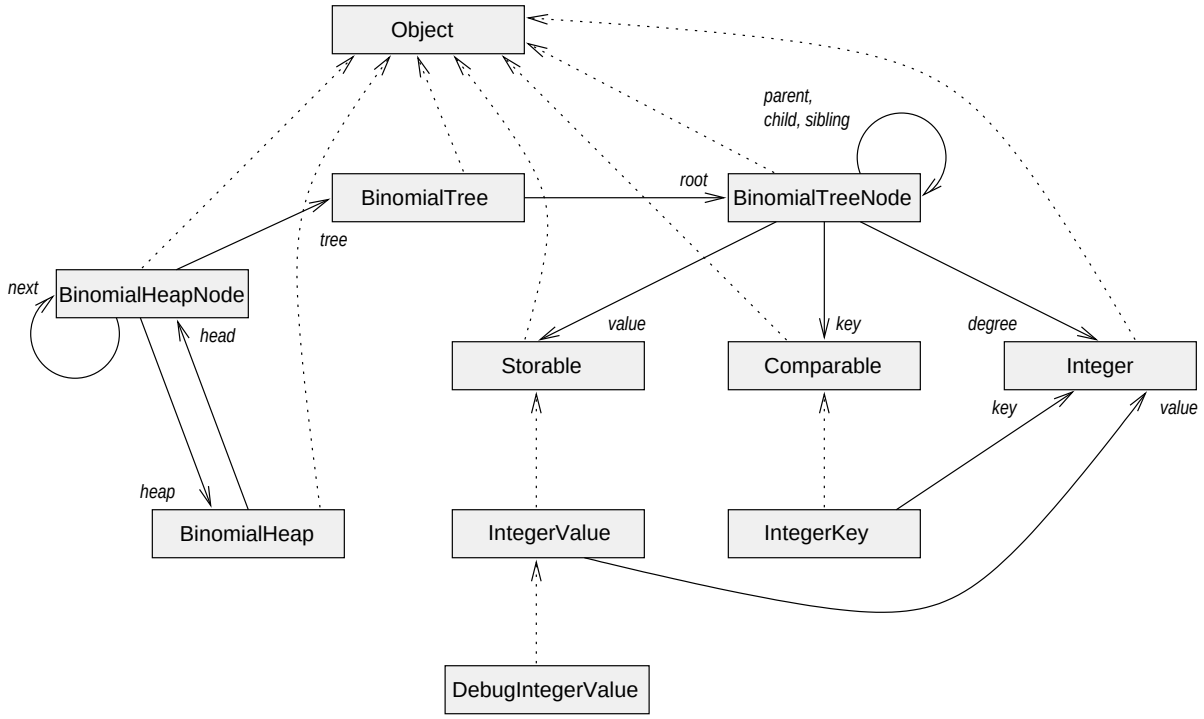

Figure 2: Example class hierarchy.

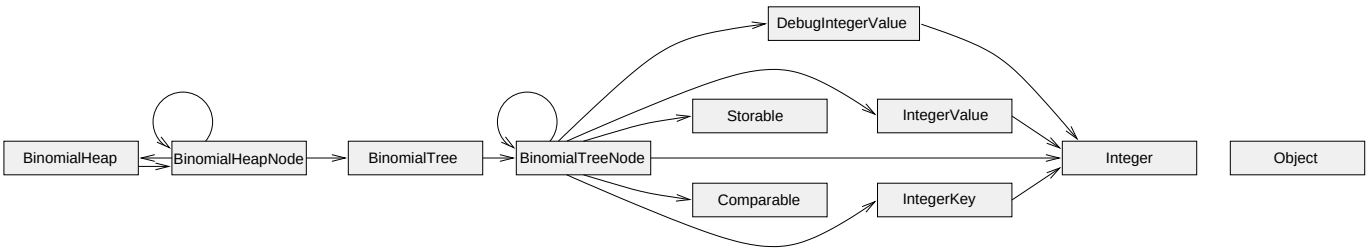

Figure 3: Class graph.

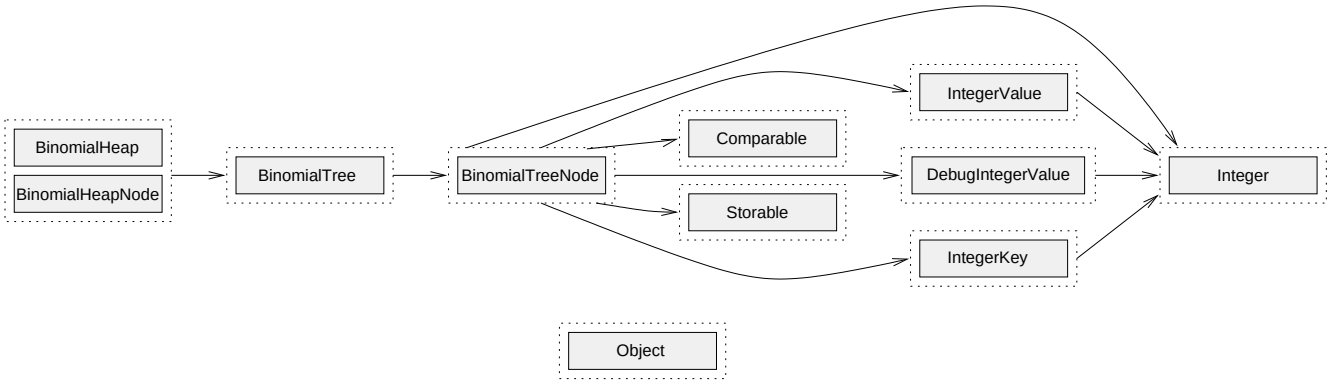

Figure 4: Partition graph.

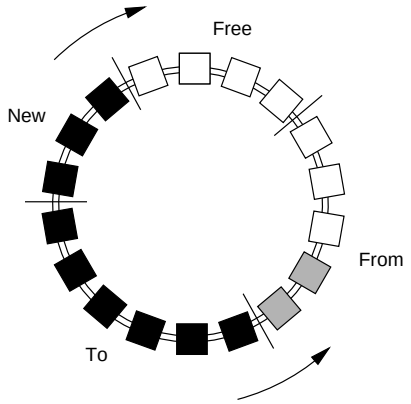

Figure 5: Baker's treadmill collector. As objects are allocated the boundary between the new and $free$ regions circulates clockwise. As objects are scanned the boundary between the to and $from$ regions circulates counter-clockwise. If memory is not to be exhausted then the two boundaries must be prevented from colliding. In practise there would be many more objects in each of the sections.

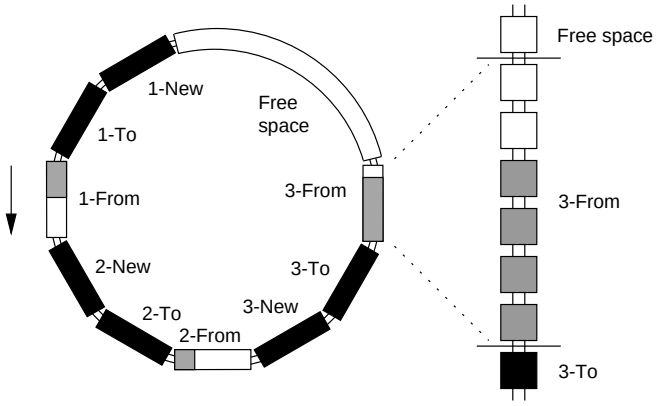

Figure 6: Modified treadmill in which a separate new, to and $from$ section is defined for each partition. The partitions are organized so that they are topologically sorted when considered counter-clockwise from the common $free$ section. Within each partition instances are organized as before on a doubly linked list. This is illustrated for the $3\text{-}from$ section, in which there are four gray objects and two white objects.

The *from* section therefore comprises a mixture of objects which are non-garbage and awaiting scanning (gray) and objects that are candidate garbage (white). The objects within from *from* section are organized so that the gray objects are held contiguously towards the front of the *from* section, that is towards the boundary between the *from* and *to* sections. Note that this approach differs from the original presentation of the treadmill collector, in which gray objects were held in the *to* section and a separate pointer was used to identify the boundary between black and gray objects.

Initially the *to* section is empty and the objects referenced from the roots are grayed and immediately at the front of the *from* section. Scanning proceeds as follows:

```
while (object at front of 'from' is gray) {
  mark object black
  place object in 'to'
  foreach (child of object) {
    if (child is white) {
      mark child gray
      move child to front of 'from'
    }
  }
}
```

This corresponds to a depth-first traversal of the object graph. An alternative breadth-first traversal could be implemented by moving objects to the white/gray boundary within the *from* section rather than moving them directly to the front of the section.

3.2 Assigning classes to partitions

The algorithm described in Section 2 analyses the complete class hierarchy in order to assign classes to partitions. However, this approach is not sufficient for use with Java since the JVM may load and initialize classes at various points during the execution of a program – it is often possible for an application to define new classes dynamically through the `loadClass` method of a class-loader [LB98].

Rather than performing a single analysis phase, the prototype implementation updates the mapping from classes to partitions on each occasion that a class is *instantiated* for the first time. A secondary benefit of only considering instantiated classes is that it ensures that, at any point in the analysis, only a single class needs to be considered. It may also allow partitions to contain fewer classes if there are classes which contribute edges to the class graph but which are never instantiated.

3.3 Representing partitions

Each partition is represented at run time by a singly linked list of classes. This makes it straightforward to merge partitions when new classes are loaded and also allows the size of partitions to vary dynamically. Each class definition contains a pointer back to the partition of which it is a member.

The partition graph is represented by holding the partitions topologically sorted on a linked list. The organization of partitions on this list corresponds to the order in which they will be scanned (see Section 3.4, below).

Note that there is no per-instance overhead since the JVM already maintains sufficient detail to recover the class of any instance. This information is usually used for virtual method dispatch, for unavoidable run-time type checks, for distinguishing fields containing references during garbage collection and for the implementation of the `Object.getClass()` method.

Partition number	Classes	Size	Partition number	Classes	Size
1	139	1919032	20	1	560
2	1	160	21	1	28172
3	1	344	22	1	213376
4	1	16	23	1	8
5	1	40	24	1	3480
6	1	1360	25	1	22344
7	1	1304	26	4	84
8	1	6024	27	1	48
9	1	7168	28	1	352
10	1	336	29	1	444
11	1	576	30	1	516
12	1	1312	31	1	328
13	1	2504	32	1	291768
14	1	11144	33	1	1085220
15	1	1864	34	1	4856170
16	1	640	35	1	912303
17	1	4860	36	1	3656
18	1	27340	37	1	216
19	1	96			

Figure 7: Heap partitions that can be established when using the `javac` application to compile some Java source code to bytecode.

Partition number	Classes	Size	Partition number	Classes	Size
1	21	198788	20	1	4
2	1	64	21	1	91484
3	1	204	22	4	84
4	1	4	23	1	20280
5	1	128	24	1	1152
6	1	6960	25	1	36
7	1	20880	26	1	15132
8	1	1612	27	1	5056
9	1	27840	28	1	211968
10	1	15156	29	1	288900
11	1	16840	30	1	7395
12	1	10056	31	1	16493054
13	1	13472	32	1	8785176
14	1	6736	33	1	232700
15	1	28628			
16	1	44			
17	1	18128			
18	1	5064			
19	1	8			

Figure 8: Heap partitions that can be established when using the `jar` application to create an archive of `class` files.

3.4 Changes

The basic treadmill of Figure 5 is extended to incorporate separate *new*, *to* and *from* sections for each of the partitions. These sections are organized so that, when considered counter-clockwise from the common *free* section, the partitions appear topologically sorted.

Figure 6 illustrates this with an example. There are three partitions, labelled *1*, *2* and *3* and so perhaps $1 \Rightarrow 2$ and $1 \Rightarrow 3$. Scanning is in progress in the *from* region of the first partition. Note that there are already gray objects in each of the other partitions – these are objects which are reachable directly from the roots, or which are reachable from one of the scanned objects in the first partition. When the *from* section of the first partition has been scanned completely any white objects which remain in it can be reclaimed immediately. The collector implementation proceeds with remarkably few changes:

```
start with first partition
while (not scanned all partitions) {
  while (object at front of current 'from'
         section is gray) {
    mark object black
    move object to current 'to' section
    foreach (child of next object) {
      if (child is white) {
        mark child gray
        move child to front of 'from' section
           of the partition it is in
      }
    }
  }
  reclaim white objects in current partition
  select next partition
}
```

Note in particular that the inner loop, in which the collector is scanning gray objects and examining their children, is practically unchanged.

Although it is not discussed here, the handling of objects within the *free* section is also substantially modified in order to allow variable sized allocations to be handled with reasonable performance. The approach taken roughly follows the use of segregated free lists in [LPB98].

4 Results

The results presented here show the performance of the modified treadmill collector when it is used with two example applications, `javac` (a Java-to-bytecode compiler) and `jar` (a tool which generates archives of Java `class` files). In each case the collector was configured to scan two objects every time that an allocation was requested.

Figures 7 and 8 show the extent to which the heap may be partitioned in these two cases. The tables in these figures show the partitions in the order in which they will be scanned, the number of classes assigned to each partition and the total size (in bytes) of all instantiations of those classes. In each case there is an initial partition containing many classes, followed by many smaller partitions each of which typically contains only a single class. The classes in this initial partition tend to be those which have fields of type `Object` and which may therefore potentially refer to any instance. The extent to which it may be possible to reduce the size of this initial partition is discussed in Section 6.

In the case of the `javac` application, the level to which the heap may be partitioned is disappointing – almost all of the classes which represent nodes in the parse tree are drawn into the initial partition. However although the initial partition contains many of the classes it does not individually

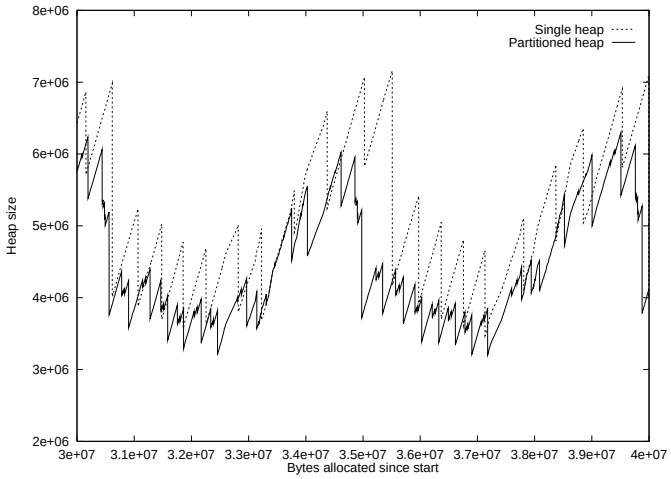

Figure 9: Heap size while running the `javac` Java-to-bytecode compiler using an un-modified treadmill collector and a partitioned heap.

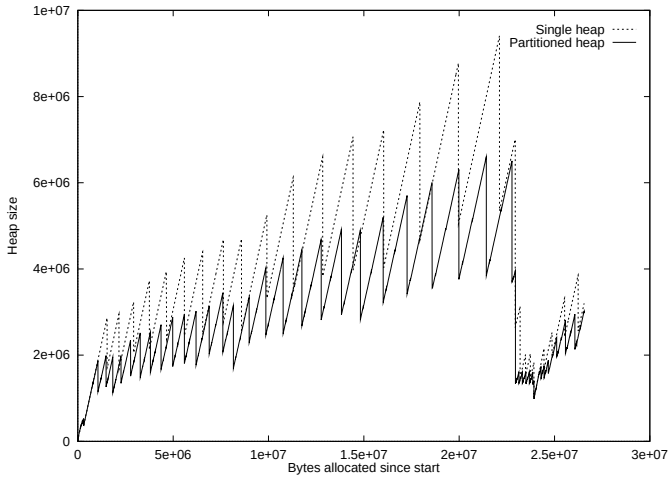

Figure 10: Heap size while building a multi-file `jar` archive using an un-modified treadmill collector and a partitioned heap.

contribute most to the size of the heap – the partition containing arrays of the primitive `char` type has approximately twice the total size.

Figures 9 and 10 show the total heap size during the execution of the `javac` and `jar` applications. The horizontal axis shows elapsed time, measured in the total number of bytes allocated since execution began. The vertical axis shows the total size of the non-free regions of the heap. Note how the traces from an un-partitioned heap have a 'spiky' appearance, in which a long garbage collection cycle is followed by a sudden reclamation of free memory. In comparison, the traces from a partitioned heap exhibit smaller spikes, showing how space is reclaimed at several points during the course of a single collection cycle.

5 Related work

The idea of separating different kinds of objects is not new. Indeed, the usual distinction that is drawn between the *root set* and the *heap* is essentially the same as the distinction between partitions – references may exist from the root set into heap-allocated objects, but heap-allocated objects do not contain references back to the root set.

The technique of segregating objects which contain references from those which do not has been seen to have several potential benefits. For example, separating large objects containing bitmapped graphics or text strings may avoid the cost of scanning these objects in vain for pointers [GR83, JL96]. The technique also allows these sections of the heap to be managed under different policies – for example by using a copying and compacting collector for the majority of the heap, but avoiding copying the large and potentially long-lived bitmaps whenever possible.

5.1 Atomic objects

A similar distinction is drawn in the Boehm-Demers-Weiser collector between *normal* and *atomic* data [Boe93]. This is a conservative garbage collector that may be used in 'hostile' environments such as those provided by C and C++ applications. By segregating *atomic* objects the frequency with which the conservative collector mistakes non-pointer values for pointers is reduced.

5.2 Generational collection

A *generational* garbage collector separates the heap into a number of *generations* according to an estimate of the ages of objects [Ung84]. In the simplest case there may only be two generations, a *new* generation which contains recently allocated objects and an *old* generation which contains objects which have existed for some time. This approach is motivated by the intuition that *most objects die young* and that it is therefore more productive to scan new objects frequently rather than to scan the entire heap less often.

In most languages, however, it is necessary to be particularly careful about inter-generational pointers – it is possible for references to exist between the *new* and *old* generations in either direction. In contrast, the partitioning algorithm presented here leverages the safety of the programming language to *guarantee* that references only flow between partitions in one direction.

Previous work in the context of Standard ML [Rep93] has further segregated generations into *arenas* which contain different kinds of objects. Aside from *records* (the most common ML objects), there are separate arenas for *pairs*, *strings*, *arrays* and *code*. Pairs (records which contain two pointers) are segregated in order to avoid storing a descriptor with each individual pair. Strings are atomic objects which cannot contain pointers. Arrays include all non-atomic mutable objects which are segregated in order to help track inter-generational references more efficiently. Code objects tend

to be both large and long-lived and are segregated so that they may be managed with a mark-sweep collector in order to avoid unnecessary copying.

5.3 Regions

Recent work, also using ML, has proposed a quite different organization in which the heap is formed from a stack of *regions* [TT94, TT97, Tof98]. Each region is itself a stack of potentially unbounded size and allocations may be made at any time into any of the regions. Storage space is reclaimed by popping entire regions from the stack.

A translation is defined from well-typed source language expressions into a target language in which region management is explicit. Although superficially similar, there are several fundamental differences between the approach taken when using regions for memory management and when using the static analysis described above. Essentially, region-based storage management offers a potential replacement for traditional reference-tracing garbage collection. Also, the partitioning algorithm described above considers is based solely on the class hierarchy and field definitions, rather than on an analysis of method definitions.

The success of region based memory management is noted to depend on programs being written in a *region friendly* style based on profiling tools or intuition. In a similar manner, it is possible to note that the success or failure of the partitioning algorithm described here is dependent on the style in which a program is expressed.

6 Future work

The results presented in Figure 7 and Figure 8 illustrate the extent to which the straightforward analysis presented here is able to segregate the heap into partitions. In each case however there is a large initial partition which accounts for a significant proportion of the heap – both in terms of the number of classes that it contains and in the total size occupied by instances allocated within the partition.

The analysis on which the prototype implementation is based takes a very pessimistic view of how references may be manipulated. The worst case of this is that instances of any class which defines a field of type `Object` must be considered to be able to refer to any other object – even when this freedom is not exploited in a particular program.

For example, the class `java.util.HashtableEntry` (used in the definition of 'generic' hash-tables) contains `key` and `value` fields of type `Object`. However, a programmer instantiating a hashtable is likely to do so with the intention that it holds a more restricted set of objects – for example objects which all subclass a particular class or which all implement a particular interface (as was the case with the `BinomialTreeNode` shown in Figure 2).

The problem presented here is related to the one that the current proposals for adding *parameterized types* to the Java language are trying to solve [BOSW98, OW97, AFM97, MBL97]. These proposals generally provide a mechanism for writing a class definition in a 'generic' style and then parameterizing it at different places where it may be instantiated – for example a suitably-parameterized hashtable definition may be able to hold a mapping from `String`s to `String`s when instantiated from `Hashtable<String,String>` but a mapping from `Integer`s to `InputStream`s when instantiated from `Hashtable<Integer,InputStream>`. The key aims for providing parameterized types are to allow more thorough compile-time checking of programs and to reduce the overhead that is imposed by run-time type-checks which the programmer knows are *obviously* going to succeed.

It is possible, to varying degrees, to build on this existing work. The stumbling block is that many of the existing approaches discard the additional type information quite early during compilation as a consequence of being defined in terms of a translation into standard Java source code or into standard Java `.class` files. The approach recently taken by Solorzano and Alagie [SA98] is perhaps more suitable since it preserves information about generic classes and their instantiations until run-time. The original motivation for this approach was to address shortcomings in both the *homogeneous* and *heterogenous* translations of earlier implementations [OW97] and in doing so to extend Java Core Reflection [JCR] to manipulate generic classes cleanly.

7 Conclusions

This article has described a novel technique for allowing early reclamation of storage space based on a static analysis of class definitions. The prototype implementation, based on a modified version of Baker's treadmill collector, has demonstrated the utility of this approach and the ability for it to be implemented with virtually no changes in the inner loop of the collector and no per-instance storage overhead. Future work is proposed to investigate how a more finely grained partitioning may be developed.

References

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, October 1997.

[App98] Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Position paper. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume

28(6) of *SIGPLAN Notices*, pages 197–206, Albuquerque, New Mexico, June 1993. ACM Press.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN Notices*, 33(10):183–200, October 1998.

[CLR90] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Höl94] Urs Hölzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Thesi CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.

[HU95] Urs Hölzle and David Ungar. Do object-oriented languages need special hardware support? In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 283–302, Åarhus, Denmark, 7–11 August 1995. Springer.

[JCR] Java core reflection, jdk 1.1. Sun Microsystems, Inc., `http://java.sun.com`.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, October 1998.

[LPB98] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, pages 118–129, Vancouver, October 1998. ACM Press.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 15–17 January 1997.

[Mor95] J. Gregory Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie-Mellon University, December 1995. Published as CMU Technical Report CMU-CS-95-226.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[POP94] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices. ACM Press, January 1994.

[Rep93] John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993.

[SA98] Jose H Solorzano and Suad Alagie. Parametric polymorphism for java: A reflective solution. *ACM SIGPLAN Notices*, 33(10):201–215, October 1998.

[Tof98] Mads Tofte. A brief introduction to Regions. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, pages 186–195, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[TT94] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report Computer Science 93/15, University of Copenhagen, July 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. An earlier version of this was presented at [POP94].

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, April 1984.