Cyclic reference counting by typed reference fields

J. Morris Chang, Wei-Mei Chen, Paul A. Griffin, Ho-Yuan Cheng

Abstract

Reference counting strategy is a natural choice for real-time garbage collection, but the cycle collection phase which is required to ensure the correctness for reference counting algorithms can introduce heavy scanning overheads. This degrades the efficiency and inflates the pause time required for garbage collection. In this paper, we present two schemes to improve the efficiency of reference counting algorithms. First, in order to make better use of the semantics of a given program, we introduce a novel classification model to predict the behavior of objects precisely. Second, in order to reduce the scanning overheads, we propose an enhancement for cyclic reference counting algorithms by utilizing strongly-typed reference features of the Java language. We implement our proposed algorithm in Jikes RVM and measure the performance over various Java benchmarks. Our results show that the number of scanned objects can be reduced by an average of 37.9% during cycle collection phase.

1. Introduction

Automatic dynamic memory management (a.k.a. garbage collection) has received a great deal of attention in recent years, especially in the object-oriented language systems. It provides benefits in software development, testing, and security. The productivity is improved when programmers are free from managing objects during the software development stage and the software execution is free from memory leaks. Today garbage collection is a core component in most object-oriented languages and also supported by the modern run-time environments (e.g. Java Virtual Machine and .NET Framework).

There are two general approaches for garbage collection: tracing and reference counting. Tracing collectors, which include mark-sweep, tricolor and generational/copying collectors, are often the choices for high performance applications [9]. Reference counting was not embraced widely in the past, but has received a renewed interest recently [5,7,11,19,22,23,27,28], due to its tight data locality, naturally incremental behavior, and rapid reclamation of memory. Besides, reference counting provides small pauses for collection cycles, so that it can be used for real-time applications and embedded systems.

Although reference counting [14,18] has many innate advantages, it also has particular weaknesses which often limit its usage. Due to the nature of the algorithm, the two features of natural incremental workload and rapid memory reuse are in conflict with each other. But most crucially, reference counting in its basic form is not complete, in that it cannot reclaim cyclic data structures [26], leading to memory leaks. These cyclic data structures can be addressed in three general ways: static elimination of any possible cyclic structures, a backup tracing collector which periodically collects accumulated cyclic garbage, or special functionality built into the reference counting collector to handle garbage cycles. Static cycle elimination, when it is not done by a program analysis in a compiler, could place additional burden in the developer, undermining the stated benefits of garbage collection. A backup tracing collector can introduce significant...
additional complexity to the run-time system and inflate the pause times in execution associated with any kind of garbage collection. These points suggest that cyclic reference counting, a functional extension to reference counting to collect cyclic data without compromising incremental behavior, would be an ideal solution.

Among cyclic reference counting algorithms, some of the most prominent are local mark-scan algorithms, as introduced by Martinez et al. [25]. But these cyclic reference counting algorithms are often inefficient, incurring significant scanning overheads during run-time and creating indeterminate pauses during program execution. Recent work has been done to reduce these cyclic scanning overheads [7,22,23], with regard to local mark-scan algorithm of cyclic reference counting.

However, these recent improvements have made only limited use of programming language semantics to predict the behavior of objects. Knowledge of this behavior could be applied to more quickly identify specific kinds of data structures, including cycles. The types of data structures that can be expected depend on the inter-object connectivity of a given program, which in turn is based on the reference fields among the objects in memory. In many languages, such as Java, the structure of each data type, and the types that it can link to, are very well defined, and can be used to predict the nature of the data structures in which it can participate. Our study shows that this strongly-typed reference feature of the Java language can be utilized to improve the cyclic reference counting algorithm effectively.

The contribution of this paper is threefold: to introduce a new classification in modeling the predictable inter-object connectivity; to present the concept of double reference counts for capturing the relation of references between objects; and to propose a new cyclic reference counting algorithm which can distinguish dispensable operations based on the previous two schemes to reduce the scanning overheads effectively. Moreover, we implement our proposed algorithm in Jikes RVM [1,2] and measure the performance over various Java benchmarks.

The rest of the paper is organized as follows. In Section 2, we describe the current cyclic reference counting algorithms, including their strengths and weaknesses. Next, in Section 3, we introduce a novel object classification model based on the inter-object connectivity. Then, in Sections 4 and 5, we present our double reference counts algorithm and analyze our experimental results. And then, we review prior related work in Section 6. Finally, we conclude the paper with some remarks in Section 7.

2. Cyclic reference counting

The cyclic data structures are the structures in which an object is reachable from itself, either directly or indirectly (through other objects). Generally, reference counting maintains a reference count (RC) for each object indicating the number of pointers that reference the object. For any object, adding or deleting references to the object will cause increment or decrement of its RC respectively. If an object’s RC is zero due to other reference deletions, it should be recycled. However, the counts of cyclic garbage objects never drop to zero; therefore a reference counting algorithm in its basic form cannot reclaim cyclic data structures. The remainder of this section is divided into two parts. We first review a popular cyclic reference counting collector: local mark-scan. Then we describe existing extensions and optimizations of this basic framework.

2.1. Local mark-scan algorithm

The local scan algorithm introduced by Martinez et al. [25] breaks garbage cycles by capitalizing on the following: when a cycle becomes unreachable from the active program space, at least one RC of the objects on the cycle will be decreased to a non-zero value. Whenever an object’s RC is decreased to a non-zero value, this object is potentially a node in a cyclic structure of garbage, or a potential cyclic root (PCR). Once this occurs, the local mark-scan algorithm must proceed to collect cyclic garbage.

Starting from a PCR O, this garbage collection is done in three stages: MarkGray, Scan, and CollectWhite. The MarkGray operation scans through all target objects reachable from O, decreases every target’s count by its related internal references (references that are reachable from O), and marks all the targets as potential garbage (gray). The Scan operation divides the targets according to their reference counts into two classes: objects with counts of zero are marked as garbage (white), and those that have non-zero counts with external references are marked as normal (black). The CollectWhite operation simply collects all white objects and marks them black. The pseudocodes for these operations are given in Fig. 1.

2.2. Previous enhancements to local mark-scan

The lazy mark-scan algorithm is suggested by Lins [21], which is modified by the original local mark-scan algorithm. It reduces redundancy by placing PCRs into a buffer instead of performing the scan operation immediately. When this buffer is full, or the free list is full, or the collector decides for some other reason, all PCRs are scanned for cyclic garbage. Later Bacon and Rajan [7] further improve the process in two ways. First, every object has a flag to record if it has been added into the buffer to avoid needless scanning. Second, all PCRs are treated as a single graph and MarkGray operation performs on all cyclic roots concurrently. This reduces the worst-case time spent in scanning from quadratic to linear.

The approaches illustrated here are quite efficient, but model objects only generically, as containers of references. Object oriented languages have type information in their class definitions that can be used effectively to reduce the run-time cost of these local-scan algorithms. These characteristics of objects rely heavily on the use of typed fields in many object oriented languages, like Java. In the following two sections, we will propose our two modifications on cyclic reference counting algorithm, which exploit such characteristics of objects and the strongly-typed features in the Java language, so as to reduce the scanning overheads and the run-time pauses during program execution.
3. Classification models

The Java programming language is a strongly-typed language, which means that every variable and every reference has a type that is known at compile time. This feature of the Java language allows us to formulate much more detailed properties of objects than what we can get from an arbitrary directed graph. In this section, we will present a new classification model for object connectivity to make best utilization of the fixed types of reference fields in the Java language.

3.1. Object classification

Object classification refers to the categorization of data and references based on the types of connectivity that they can potentially exhibit. In the existing research on cyclic garbage collection, the most prevalent object classification is to categorize the connectivity into two types: cyclic and acyclic [5]. However, this categorization is not sufficient to fully describe the basic behavior of objects that is relevant to the cycle scanning algorithm. Therefore, in order to better utilize the behavior of objects, all objects are further divided into three types: cyclic type, terminating type, and linking type. For the purpose of clarity, we describe the definitions of object types as follows.

Definition 1. An object is cyclic if it is reachable from itself.

Definition 2. An object is acyclic if it is not cyclic.

Definition 3. An object $x$ is linking if it is acyclic and satisfies both of the following requirements:

(a) there exists a cyclic object that is reachable from $x$ and

(b) there exists a cyclic object that reaches $x$.

Definition 4. An object is terminating if it is acyclic and satisfies at most one of the requirements (a) and (b).

According to these definitions, the acyclic type is partitioned into the terminating type and the linking type. A linking object links two cyclic structures and terminating objects cannot form a link between two cyclic structures. Also note that all entities in the root-set of an application, such as static class variables and the execution/operand stack, can be considered to be the terminating type. This is because the root-set is always alive, making any references to it irrelevant. Fig. 2 illustrates the relations of these connectivity types.

During mutator activity, linking and terminating objects can be treated identically: they do not need to be buffered as possible garbage cycles if their RC are decremented to a non-zero value. However, during cycle collection, linking and cyclic objects can be treated identically: some terminating objects can be ignored in this phase, while linking objects must be scanned, in order to keep the worst-case scanning requirement linear with respect to the number of objects in the heap. Fig. 3 shows a worst-case example of scanning. In the case illustrated here, even if all objects in the PCR buffer are scanned concurrently, links between cycles must also be scanned. If this does not occur, either uncollected garbage cycles will be
discarded, resulting in memory leak, or all garbage cycles except the rightmost will have to be re-scanned later, resulting in a quadratic total scanning requirement as this process is repeated.

3.2. A new object classification model

Generally, more rigorous analysis will yield superior accuracy, and thus superior performance. An object classification will analyze a program to determine the types of all objects. This analysis can be performed by searching along the types of instance fields of a program’s classes for cycles, as done in [3,12]. Thus we make a thorough and experimental object analysis for our selected SPECjvm98 and DaCapo benchmarks to verify the efficiency of this aggressive classification.

Table 1 gives the results of the aggressive classification, including the frequency of cyclic, linking, and terminating objects. The classification is performed according to an exhaustive scan of the non-static reference fields of the objects for which objects are instantiated. From this table, we observe that the percentage of the linking type among all the objects is quite small. Besides, the task of the aggressive analysis is very time-consuming. These facts reveal that it is not economic for dividing objects into three categories. For the selected Java programs, we can just focus on cyclic and terminating objects to improve the efficiency of garbage collection. Therefore, in the following section, we will suggest a reference counting algorithm based on another object classification: terminating or non-terminating objects. Such a categorization is more simple and efficient than aggressive one. The definitions of non-terminating object is described as follows.

Definition 5. An object is non-terminating, if it is not terminating.

From the above definition, we have that all objects are partitioned into terminating and non-terminating types, and non-terminating type consists of cyclic and linking objects.

4. The double reference counts algorithm

Besides the new object classification model, we bring up another more effective approach to improve the local mark-scan algorithm, which is the double reference counts (DRC) algorithm. In this section, we will introduce the structures used in the DRC algorithm and the detailed description for this algorithm.

4.1. Double reference counts

The basic idea of our improvement is the utilization of double reference counts: each object is associated with one or two fields counting for the number of reference of the relevant type(s). Terminating objects require a reference count for...
recording the number of all references to themselves, while non-terminating objects require two fields to count terminating references and non-terminating references separately. Let $\text{RCT}$ and $\text{RCN}$ be terminating reference count and non-terminating reference count respectively. A brief example is shown in Fig. 4.

To apply the concept of double reference counts to the local mark-scan algorithm, we identify and allocate storage of reference counts for every object, and we further modify the updating of reference counts correspondingly. For example, if a non-terminating object gets/loses a terminating reference, the non-terminating object's $\text{RCN}$ needs to be increased/decreased by 1. In Fig. 4(c), when the terminating reference from $T$ is released, $A.\text{RCT}$ becomes 0. The pseudocode for decrement of reference counts is described in Fig. 5 and the related increment operation can be obtained in the similar pattern.

### 4.2 Basic ideas of the DRC algorithm

Based on the partition of terminating and non-terminating objects, the main principles in our DRC algorithm is described as follows:

1. If an object is terminating, it need not be buffered as a potential root of cyclic data when any reference count of the object is decreased to a non-zero value.

#### Table 1
Percentage of instantiated objects for the selected benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Instantiated objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>jess</td>
<td>A Java expert shell system based on NASA’s CLIPS expert shell system</td>
<td>82.33 17.60 0.07</td>
</tr>
<tr>
<td>db</td>
<td>Performs DB functions on memory resident database</td>
<td>94.60 5.29 0.11</td>
</tr>
<tr>
<td>mtrt</td>
<td>A dual-threaded program that ray traces an image file</td>
<td>83.43 16.51 0.06</td>
</tr>
<tr>
<td>jack</td>
<td>A Java parser generator with lexical analyzers</td>
<td>63.12 36.83 0.05</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>A mpg-3 audio stream decoder</td>
<td>91.65 7.88 0.48</td>
</tr>
<tr>
<td>raytrace</td>
<td>Works on a scene depicting a dinosaur</td>
<td>84.26 15.68 0.06</td>
</tr>
<tr>
<td>antlr</td>
<td>Parses several grammar files and generates a parser and lexical analyzer for each</td>
<td>60.62 37.60 1.78</td>
</tr>
<tr>
<td>fop</td>
<td>Formats an XSL-FO file and generates a PDF file</td>
<td>59.38 38.82 1.80</td>
</tr>
<tr>
<td>hsqldb</td>
<td>Executes a number of transactions against a model of a banking application</td>
<td>68.72 29.73 1.55</td>
</tr>
<tr>
<td>luindex</td>
<td>Uses lucene to do a text search of keywords over a corpus of data</td>
<td>65.41 32.94 1.65</td>
</tr>
<tr>
<td>pmd</td>
<td>Analyzes a set of Java classes for a range of source code problems</td>
<td>63.35 34.46 2.19</td>
</tr>
<tr>
<td>xalan</td>
<td>Transforms XML documents into HTML</td>
<td>61.17 37.21 1.62</td>
</tr>
</tbody>
</table>

Note: SPECjvm98 (top seven) and DaCapo (bottom seven) benchmarks are considered in this paper.
2. If a non-terminating object is referenced by at least one terminating references, it can be ignored in the MarkGray operation and its descendants also can be ignored.

The reason of the first principle is that a terminating object is not included in a cycle. Therefore, it is very nature to ignore these objects in the process of cycle collection. To explain the second principle in more detail, next we discuss further on the operations of cycle collection in the original local mark-scan algorithm.

The main steps of cycle collection are MarkGray, Scan and CollectWhite. The MarkGray operation performs a depth-first search of the graph starting at a possible cyclic root O, marks all visited objects as potential garbage (gray), and removes internal reference count for each reachable objects. The Scan operation divides the targets according to their reference counts into two classes: objects with counts of zero are marked as garbage (white), and those that have non-zero counts with external references are marked as normal (black). The CollectWhite operation simply collects all white objects and marks them as black. It is worth noting that, during the progress of cycle collection, some reachable objects (from root O) which are not garbage are colored gray and then re-colored to black. If we can identify such objects in advance and ignore them in the MarkGray operation, much redundancy can be removed. Moreover, the scanning process also can be skipped for their descendants.

In Fig. 4 all objects, except G, are first colored gray, and then the objects A, B, C, D and E are re-colored white and the objects F, H, I and J are re-colored black in the Scan operation. Because the non-terminating object F has a terminating reference from a live object G, the object F and its descendants can not be recycled. On the other hand, the non-terminating object D has a non-terminating reference from a garbage object C. Thus D should be recycled.

Investigating the connectivity of objects in cycle collection, we observe that a non-terminating object referenced by a terminating object must be live (i.e. not garbage) since the terminating object can not be reachable from the potential cyclic root which triggers this cycle collection. Next we will prove that if a non-terminating object X is referenced by a terminating object, the terminating object is not reachable from any non-terminating object.

**Theorem 1.** Let T be a terminating object reachable from a non-terminating object N. If an object X is reachable from T, then X is terminating.

**Proof.** Suppose that X is non-terminating. We have X is cyclic or linking. If X is linking, there is a cyclic object C reachable from X, and hence from T. This implies that T is non-terminating, which is contradictory. If X is cyclic, there is a cyclic object (i.e. X) reachable from T. Thus T is not terminating. It is also a contradiction. Therefore X is terminating. □

**Corollary 1.** If there is a non-terminating object X referenced by a terminating object T, then T is not reachable from any non-terminating object.

**Proof.** Suppose that T is reachable from a non-terminating object N. Then Theorem 1 implies that X is terminating, which is a contradiction. This completes the proof. □

From Corollary 1, we obtain that any non-terminating object with at least one terminating reference can be omitted in MarkGray operation and its descendants also can be omitted. The pseudocodes of two important operations in our DRC algorithm are given in Fig. 6. The remainder of operations in local mark-scan algorithm can be modified by similar patterns. Note that our NewMarkGray will stop at F in the case of Fig. 4.

5. **Analysis and implementation**

In this section, we will present the implementation details for this algorithm, as well as its experimental results. We implement our proposed algorithm in Jikes RVM and measure the performance on the selected SPECjvm98 and DaCapo benchmarks. Our experiments show that the DRC algorithm can significantly reduce the pause time and the scanning overheads of the existing local mark-scan algorithm.

5.1. **Implementation of the DRC algorithm**

The proposed DRC algorithm is implemented into the standard MMTk in Jikes RVM [1,2] version 2.9.1, FastAdaptiveRefCount configuration, and built on Fedora Linux version 5.0. MMTk uses a simple high level algorithm to implement various popular non-concurrent collectors [10]. The Jikes RVM object model and FastAdaptiveRefCount configuration are aggressively optimized based on [7], which have been implemented in MMTk.

To apply the innate characteristics of Java objects, we propose a simplified classification for implementation. The terminating objects include scalars, references to objects that are both terminating and final, and arrays of either of the previous objects. All objects which are not terminating belong to non-terminating type. Since the number of terminating references are found to be relatively small, the terminating reference count can be embedded into the upper 6 bits of the reference count field to avoid incurring storage overheads. All references stemming from the root-set of an application, the boot space, and the immortal space are counted as terminating references.

The required size of a count field is implementation dependent; we suggest 6 bits to be sufficient for storing the terminating reference count. This approach can avoid incurring memory overhead. However, it is possible to use 32 bits for
this count field which can reduce the possibility of overflow in this count field. Any other static or run-time overhead is implementation dependent as well.

Efficient operation of the chosen model requires specialized write barriers for the different connectivity types. A Putfield operation requires one of two write barriers: one for terminating fields and the other for non-terminating fields. An Arraystore operation requires one of three write barriers, selected according to the following. If the compile-time type of the array is terminating, a specialized terminating write barrier can be used. Otherwise, if the possibility of inheritance can be statically eliminated, or if the array’s innermost element type (without regard to dimensionality) is non-terminating, then a non-terminating write barrier is used. If none of these cases hold, then a type-blind write barrier which performs a run-time check is used. For the non-terminating and blind write-barriers, a pointer boundary check is performed, in case the field being updated resides outside of reference counted space, which would implicitly make it a terminating reference.

Beyond the additional write barrier requirements, the RVM requires separate increment/decrement buffers for the different connectivity types, bitmaps for each class to denote the connectivity types of reference fields, and a 1-byte field for each class, array type, and fields to denote the connectivity type.

Both the modified and default local mark-scan algorithm are implemented in RVM to record the number of buffered objects, and the number of analyzed objects and traced pointers in the MarkGray, Scan, ScanBlack and CollectWhite phase. The costs of this implementation are as follows: the implementation must include two write barriers (and in the case of an interpreted VM, two Putfield bytecodes), one for terminating fields and the other for non-terminating fields. An Arraystore operation requires one of three write barriers, selected according to the following. If the compile-time type of the array is terminating, a specialized terminating write barrier can be used. Otherwise, if the possibility of inheritance can be statically eliminated, or if the array’s innermost element type (without regard to dimensionality) is non-terminating, then a non-terminating write barrier is used. If none of these cases hold, then a type-blind write barrier which performs a run-time check is used. For the non-terminating and blind write-barriers, a pointer boundary check is performed, in case the field being updated resides outside of reference counted space, which would implicitly make it a terminating reference.

Beyond the additional write barrier requirements, the RVM requires separate increment/decrement buffers for the different connectivity types, bitmaps for each class to denote the connectivity types of reference fields, and a 1-byte field for each class, array type, and fields to denote the connectivity type.

The costs of this implementation are as follows: the implementation must include two write barriers (and in the case of an interpreted VM, two Putfield bytecodes), one for terminating fields and the other for non-terminating fields. The static cost in terms of space required is one additional bit per field per class. Static computation cost is unchanged from the default RVM. The run-time space requirement is represented by the additional on-object reference fields. The number of additional count fields per object is dependent on the object’s connectivity type. The required size of a count field is implementation dependent; we find 6 bits to be sufficient for storing the terminating reference count. Any other static or run-time overhead is implementation dependent as well.

5.2. Experimental results

The benchmarks are selected from the SPECjvm98 and DaCapo suites. The 12 benchmarks are run with their related minimum-sized heap [8,15] on both RVM implementations. The final results presented in Figs. 7 and 8 are the averaged results of these ten runs.

Fig. 7 compares the total objects traced over all collection phases, between the default RVM and the DRC improvement. The collection phase refers to all collection phases in a single run. The proposed referencing counting algorithm is executed during the collection phase. For references, all benchmarks tested buffer at least 260,000 PCRs for scanning in the default RVM, and at least 240,000 in modified RVM. This difference is because of the “else if (A is terminating)” test in NewDecrement of Fig. 6. As Fig. 7 shows, nearly all phases in cycle collection are reduced by using the DRC algorithm, especially the MarkGray phase and the ScanBlack phase. The number of buffered objects in MarkGray phase is reduced significantly, and so does that in the ScanBlack phase. The reduction in buffered/traced objects ranges from 6.1% to 45.0%, with an average over the twelve benchmarks of 26.4%. If the objects buffered as PCRs are ignored, the reduction in traced objects ranges from 7.1% to 69.4%, with an average of 31.4%.
If we record the counters in field level, we can get the number of traced references in each cycle collection phase. Fig. 8 shows the total references traced during each collection phase. As Fig. 8 shows, the number of traced references is also reduced effectively by the DRC algorithm. The reduction ranges from 1.2% to 64.4%, with an average of 35.2%.

The total application execution time composed of mutator time and garbage-collection time (or GC time). Table 2 gives the time spent in reference counting (called per-collection time) and the total execution time for two implementations respectively. Each of the results is an average of ten runs. This per-collection time includes the execution time for all the procedures in Double Reference Counts Algorithm (in Section 4). In Table 2, the average per-collection time reduction across all applications is 37.9% and the average execution time reduction is 31.4%. The db benchmark is the only application that does not benefit from the proposed scheme. The main reason is that this database application involves most of the objects which are long-lived cyclic objects. These objects are still the candidates of cycle scanning in our proposed scheme.

According to the data above, the DRC algorithm is quite effective for reducing the scanning overheads in cycle collection and the total time spent in garbage collection. Therefore, the run-time pauses during program execution are reduced.
6. Prior work

Aside from local mark-scan algorithms, cyclic reference counting was addressed by Brownbridge [12] with weak and strong references, which was extended in [3] to optimize scanning in databases. Although Brownbridge also used multiple reference types and corresponding reference counts, his reference types were based only on the structure of the heap at the time objects and references are created, and make no use of class connectivity.

Local mark-scan algorithms were explored previously in [3,7,18,21,25,29]. The research [13] used an algorithm similar in methodology to local mark-scan, except that each cycle collection must be performed across the entire heap. Local mark-scan algorithm was first proposed in [25]. Later a more efficient algorithm, called lazy mark-scan, was suggested in [21] by using the cyclic buffer to reduce repeated scanning of the same nodes. The work [7] further devised a concurrent algorithm, based on lazy mark-scan, which eliminates the possibility of quadratic time required for scanning. Besides, the paper [29] adapted local mark-scan methodology to distributed systems for minimizing costly inter node tracing in garbage collection. The algorithm presented in [3] extended the standard local mark-scan algorithm to reduce tracing in a distributed environment and introduced the use of a schema graph to perform cyclic classification.

For mark-sweep garbage collectors, the paper [4] proposed an on-the-fly algorithm based on the sliding views mechanism to have smaller pause times than those of the stop-the-world collector. This work executes garbage collector on a separate thread so that the program threads (the mutators) can run concurrently on a different processor (in a SMP system). Besides, the research work [30] focused on the overhead and concurrency issues in concurrent garbage collection algorithm. It presented correctness-preserving transformations can be combined to derive safe collection algorithms. Our research deals with reference counting strategy which has shorter pause time comparing to mark-sweep garbage collectors.

The recent study [20] intended to improve the efficiency of tracing cyclic garbage in reference counting. It suggested a lightweight cycle reference counting algorithm that only exam a single sub-graph (instead of individual cycles) as the basic unit of cycle collection. In addition, it also introduced a technique to avoid redundant scans over garbage objects. Our research also intends to reduce the scanning overhead in reference counting algorithm. But, we exploit the strongly-typed reference features of the Java program.

For static analysis, the work [16] presented a series of optimization techniques that reduce the execution time of a nondeferred reference-counting garbage collection, based on C# strongly-typed object classification. In [17], a framework for nondeferred reference-counting garbage collector was proposed, which unified several proposed optimizations based on the idea of overlapping roots.

Lins’ Jump-stack further improved on lazy mark scan by eliminating the Scan operation [22]. The Jump-stack is used to store all objects found to have a non-zero reference count during the MarkGray phase. This allows for the ScanBlack operation to be performed directly, with no intermediate step required to find externally referenced objects. More recently, a new algorithm [23] was suggested to postpone the local mark-scan operation until there is the risk of isolating a cyclic data structure causing a space-leak.

7. Conclusions

Cyclic reference counting offers a naturally incremental, fully correct garbage collection algorithm. While the basic local mark-scan algorithm suffers from poor efficiency, there are a variety of methods by which this efficiency may be improved. This paper presents a new method to model the object connectivity in Java programs, which can categorize the objects and references in a more effective way, so as to improve the efficacy of cycle collection phase potentially. Besides, this paper introduces an effective enhancement to the local mark-scan algorithm by applying double reference fields in Java objects. The correctness of the suggested strategies is argued.

Table 2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Per-collection time</th>
<th>Total execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RVM-default</td>
<td>RVM-DRC</td>
</tr>
<tr>
<td>jess</td>
<td>7490</td>
<td>2270</td>
</tr>
<tr>
<td>db</td>
<td>10,815</td>
<td>11,158</td>
</tr>
<tr>
<td>mtrt</td>
<td>8887</td>
<td>5387</td>
</tr>
<tr>
<td>jack</td>
<td>6700</td>
<td>2605</td>
</tr>
<tr>
<td>mpegudio</td>
<td>2106</td>
<td>1052</td>
</tr>
<tr>
<td>raytrace</td>
<td>5508</td>
<td>3164</td>
</tr>
<tr>
<td>antlr</td>
<td>545</td>
<td>417</td>
</tr>
<tr>
<td>fop</td>
<td>1106</td>
<td>905</td>
</tr>
<tr>
<td>hsqldb</td>
<td>4116</td>
<td>2004</td>
</tr>
<tr>
<td>luindex</td>
<td>3126</td>
<td>1990</td>
</tr>
<tr>
<td>pmd</td>
<td>1715</td>
<td>847</td>
</tr>
<tr>
<td>xalan</td>
<td>1145</td>
<td>967</td>
</tr>
</tbody>
</table>

---

Table 2: Performance comparisons of RVM default model and DRC implementation (in millions of CPU-cycle).
Combining these two new strategies, we propose the DRC algorithm which can improve the performance of cyclic reference counting. The main improvements come from identifying and disregarding unnecessary operations to reduce the scanning overheads on our new classification model. The DRC algorithm has been verified via a real implementation on Jikes RVM and the performance improvement on a set of selected SPECjvm98 and DaCapo benchmarks are reported. Based on our experimental analysis, the DRC algorithm is proved to be very effective, and demonstrates an average reduction of 37.9% in object scanning of the local mark-scan algorithm.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0296131 (ITR) 0219870 (ITR) and 0098235. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[24] Paz H, Bacon DF, Kolodner EK, Petrank E, Rajan VT. An efficient on-the-fly cycle collection. ACM Transactions on Programming Languages and Systems 2007;29(4) [article 20].