A Simple and Efficient Algorithm for Cycle Collection

Chin-Yang Lin
Department of Engineering Science
National Cheng Kung University
No 1, Ta-Hsueh Rd., Tainan City 701, Taiwan, R.O.C.
chinyang@nc.es.ncku.edu.tw

Ting-Wei Hou
Department of Engineering Science
National Cheng Kung University
No 1, Ta-Hsueh Rd., Tainan City 701, Taiwan, R.O.C.
houtw@mail.ncku.edu.tw

Abstract

The lack of collecting cyclic garbage is generally considered the major weakness of reference counting. Reference counted systems handle this problem by incorporating either a global tracing collector, or a ”partial” tracing collector that considers only the cycle candidates but needs several traces on them. In particular, the latter has become a preferred one as it has better scalability and locality (no need to scan the entire heap). This paper presents a new ”lightweight” cyclic reference counting algorithm, which is based on partial tracing and deals with the cycle problem in a simpler and more efficient way. By exploiting the lightweight hypothesis that considers a single sub-graph, instead of individual cycles, as the basic unit of cycle collection, the algorithm can detect garbage cycles in a single trace. In addition, we propose a technique for eliminating redundant scans over garbage objects, thus improving the efficiency of the algorithm. The pseudocode and its correctness proof are also presented. Finally, an implementation based on Jikes Research Virtual Machine is provided to demonstrate the effectiveness of the new algorithm.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Garbage collection, reference counting, cycle collection, Java

1 Introduction

Reference counting, a classical garbage collection scheme, was first developed by Collins [5]. Its simplicity brings the potential for widespread adoption [8]. In particular, the work of reference counting is interleaved with the running program’s execution (mutations). Such an incremental property makes it easy to maintain the responsiveness (i.e. shorter pauses), thus suitable for highly interactive systems. It has also great potential for distributed environments [15].

Unfortunately, reference counting algorithms can not reclaim cyclic garbage [18] as reference counts never reach zero in a dead cycle. This problem is generally considered the greatest drawback of reference counting, since memory leaks may occur if the mutator generates lots of garbage cycles. To solve this problem, an intuitive solution is to use a mixture of reference counting and mark-sweep [6, 19]. With a global mark-sweep collector invoked occasionally, the cyclic garbage can be recycled in the sweep phase. However, such a global tracing approach may have tough time because it needs to trace the entire heap, which can also hinder the scalability for distributed systems [16].

On the other hand, a more widely used solution is the ”partial” (local) tracing approach, which incorporates a local tracing algorithm performed on the sub-graphs suspected to contain cyclic garbage [4, 17]. It only considers the cycle candidates instead of the entire heap and therefore takes a smaller sub-graph as input. Since Christopher [4] introduced this approach, several related studies, either for uni-processor or for multi-processor, have been presented in the literature [2, 3, 9, 11, 12, 13, 14, 17, 20].

Lins’ lazy cyclic reference counting [14] was the first widely acknowledged algorithm. It extended the work of Martinez et al [17] by performing search lazily, in which many redundant local searches are eliminated by batching tracing on the associated (suspected) sub-graphs. The local search involves a well-known cycle detection procedure [17], trial deletion (TD), which normally traces objects three times. More recently, Bacon et al. [3, 2] improved Lins’ algorithm [14] by performing the tracing of all candidates simultaneously, in which all the suspected sub-graphs are considered as a whole and the number of traced objects is thus bounded. This algorithm was also extended to a concurrent cycle collection algorithm. In [20], the sliding views technique [10] was incorporated into Bacon and Rajan’s previous work [3], which reduces the cost of tracing again by further filtering out unnecessary cycle candidates.

A partial tracing collector is always confronted with a smaller input than a global one. However, with the adoption of the multiple-trace scheme (e.g. TD), the higher constant factor may also incur a significant delay if the cycle candidates are involved in a large data structure. In addition, the repeated tracing also brings more difficulties in making a collector concurrent (noticed in [3, 20]).

Most of the partial tracing algorithms were derived from the trial deletion, and several recent works [3, 2, 20] have still been concentrating on how to better choose cycle candidates for lowering tracing overhead. Few studies resort to the cycle collection procedure itself. In [22], the algorithm takes one trace over the sub-graph for gathering the dependency information among objects; a filtering procedure is then started for discovering garbage. This method traces objects only once, but the filtering procedure causes a performance bottleneck ($O(n^3)$ in the worst case). In our recent work [11], a lightweight cyclic reference counting algorithm (LW) based on partial tracing was proposed to collect garbage cycles in a simpler way. Key to the algorithm is the removal of multiple traces on the cycle candidates, which detects cyclic garbage with a single trace over the cycle candidates and hence reduces tracing cost alternatively. Its drawback, actually a theoretical problem noticed in [11], is that *completeness* can not be guaranteed, so some garbage cycles may not be timely collected.

Following the direction of [11], the algorithm proposed in this paper attempts to improve the original lightweight algorithm (LW) by reducing the effect of the incompleteness, in which more garbage cycles can, in theory, be reclaimed in a collection cycle, thereby reducing the extra scan overhead. We have implemented the algorithm in the Jikes Research Virtual Machine [1], where a set of benchmark programs from SPECjvm98 [21] were applied. The results show that the new algorithm can cope with the cycle problem for large programs with better efficiency, attaining a better application performance on average around 6% faster, when compared to a modern cycle collector [3] based on trial deletion. In addition, the results also demonstrate the effectiveness of the new algorithm, which can discover more garbage cycles (when compared to the original one), and even results in a better performance.

2 Background

In this paper, a strongly connected component (***SCC***) in the object graph is used to represent a cyclic structure or cycle. A *SCC* is a maximal set of objects in the graph such that every pair of objects is mutually reachable. A *SCC* is **trivial** if it contains a single object without self loops. **Garbage** is objects that are unreachable from any of the system roots. For a *SCC*, we call the edges in the sub-graph induced by the *SCC* **internal pointers** and the edges from outside **external pointers**. A *SCC* is called a **garbage cycle** if there are no external pointers to it.

A basic assumption under the reference counting is that garbage (object or cycle) may only be generated on deleting a pointer (i.e. decrement mutation). Consider a deleted pointer p that points to an object S. The base case is when S is a trivial *SCC*, since it forms no cycle. In this case, S will become garbage if p is the last pointer to it. When S belongs to a nontrivial *SCC*, say C, there are two cases to consider. Either p is an internal pointer, or p is an external pointer. If p is external to C, deleting p could completely isolate C from the system roots since p may be the last external pointer. If p is an internal pointer, C may be decomposed into two smaller *SCC*s. Assume S is thus contained in a new smaller "nontrivial" *SCC*, say C'. C' will become a garbage cycle if p is the only one pointer making S reachable from the system roots before p is really deleted. As shown in Figure 1, deleting p will make the new decomposed *SCC*, C', become a garbage cycle.

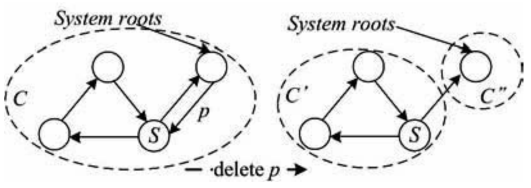


Figure 1. Pointer deletion that creates a garbage cycle

The above indicates that a garbage cycle may only be created on deleting a pointer to an object belonging to a nontrivial *SCC*. That is, garbage cycles may only be created when an object's reference count is decremented to a nonzero value (i.e., the object belongs to a nontrivial *SCC*).

Based on this observation, lots of prior work [2, 3, 9, 11, 12, 13, 14, 17, 20] take the sub-graphs below the deleted pointers as input. All potential garbage cycles are thus considered for collection. Since a starting object may be the first encountered member of a garbage cycle, it is called a **candidate root** (of a garbage cycle), written ***CR***. The computation graph (the potential sub-graphs that may include garbage cycles) can be defined as the transitive closure of all candidate roots discovered throughout the program execution. All these cycle collection algorithms are called partial tracing algorithms since they only consider the cycle candidates instead of the entire heap.

To efficiently manage the potential sub-graphs that may include garbage cycles, the algorithm proposed herein uses a similar strategy to the prior work [3, 14] - all the *CR*s (candidate roots) are placed into a buffer, denoted Roots (different from the system roots); periodically, *CR*s in Roots are batch processed for cycle collection.

In fact, different strategies for managing or extracting *CR*s applied to partial tracing algorithms can be applied to our algorithm (due to the same observation). Different trigger points for cycle detection (i.e. processing *CR*s in Roots) can also be easily incorporated, e.g., the buffer overflows, CPU is idle, memory is exhausted or maybe some heuristics will be used, etc. This paper will not cover these details. Unless otherwise noted, the related settings will refer to the synchronous cycle collection algorithm from Bacon and Rajan [3].

3 The Algorithm

The new algorithm is based on our previous work [11], the lightweight cyclic reference counting algorithm (LW). This section focuses on describing how the *completeness* of the original algorithm can be significantly improved. We will first review the original algorithm (LW) in terms of the computation graph. Then, we describe the problem of LW, followed by a solution. Finally, we present the pseudocode of the new algorithm, including its correctness proof.

3.1 The Computation Graph

The basic hypothesis of LW is that a garbage cycle can be reclaimed as soon as the sub-graph containing the cycle is detected as garbage, where the sub-graph is an object graph rooted at a candidate root. Its main idea is to consider a single sub-graph, instead of individual cycles, as the basic unit of cycle collection. Let G_i be a sub-graph considered for cycle collection, and SCC_1, ..., SCC_n be the *SCCs* contained in G_i. During the collection, LW reclaims G_i only if SCC_1, ..., SCC_n are all detected to be unreachable (dead). Under this concept, G_i is considered for collection as a whole, and thus it will no longer matter how many cycles are contained in G_i and which of these cycles are dead.

We now further abstract the computation graph for LW. Figure 2 illustrates the abstract view of the computation graph within a collection cycle, where a **collection cycle** is defined as a single complete execution of the cycle collector (starting at a specific trigger point). The entire work of a collection cycle is divided into n **work units**, where n is the number of live *CR*s in Roots and each work unit is a single execution of a cycle detection procedure starting from a *CR* in Roots. The *ith* work unit is written W_i. It is assumed that each *CR* can only be queued once per collection cycle (i.e. no duplicate *CR*s in Roots) and the collector will process *CR*s in the order CR_1, CR_2, ..., CR_n. For every $1 \leq i \leq n$, T_i is the transitive closure of CR_i and specially the sub-graph (rooted at CR_i) that is really traced (considered) during W_i is denoted G_i, where there could be a directed path from G_i to G_j, if $i > j$. Additionally, every object in the computation graph may be reachable from any of the system roots. Those pointers are all considered as external pointers.

In the original sense [14], every work unit is performed independently, in which the transitive closure of the current *CR* is always supposed to be traced. Thus, the considered sub-graph in W_i may include the sub-graphs that have been considered in the previous work units $W_1, \ldots, W_{i-1}$. For example in Figure 2, suppose W_1 reclaims

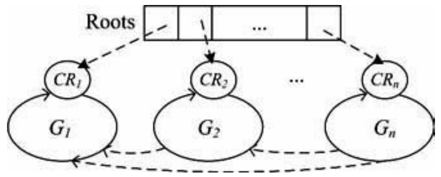


Figure 2. The Computation Graph

no garbage and there exists a one way path from CR_2 to CR_1. It can be seen that a larger sub-graph will be considered for cycle collection in W_2. In theory, this can cause the worst case that is quadratic in the size of graph.

Our algorithm avoids such a quadratic complexity by considering each object only once per collection cycle, in which every object traced in a work unit will not be traced again in subsequent work units (per collection cycle). In the implementation, each object traced would be marked as visited and remain this state till the end of the current collection cycle. Hence, it is essential to separate G_i from T_i in our work. More precisely, let V_i and C_i be, respectively, the set of objects in G_i and the set of objects in T_i. Then, $V_1 = C_1$, and for every $2 \leq i \leq n$, $V_i \subseteq C_i$. In addition, $V_i \cap V_j = \emptyset$ if $i \neq j$, and

$$\bigcup_{i=1}^{n} V_i = \bigcup_{i=1}^{n} C_i.$$

This indicates that the set of objects considered in a collection cycle is the set of objects in the transitive closure of Roots. The algorithm is thus linear in the size of the computation graph. In the same example just mentioned, our algorithm will not consider G_1 when performing W_2.

Compared to the algorithms based on trial deletion (TD), LW has two key benefits. First, it takes only one traversal over the computation graph, reducing the complexity from $3O(n)$ to $O(n)$, and secondly, it is linear in the size of the computation graph, avoiding the worst case of Lins' algorithm [14].

Technically, LW employs a special labeling scheme for the sub-graphs below Roots, with each sub-graph being associated with a distinct *ID*. Each work unit, say W_i, in a collection cycle is processed as follows. The collector first performs a depth-first search on G_i, the sub-graph (rooted at CR_i) considered in W_i, in which each visited object will be labeled by the *ID* assigned to G_i (i.e. $ID(CR_i)$), and the number of external pointers to G_i, written $\alpha(G_i)$, will also be computed. The collector then reclaims G_i only if $\alpha(G_i) = 0$ (i.e. G_i is completely unreachable.).

3.2 The Problem

As described in section 3.1, LW achieves linear complexity by isolating objects in different work units, in which the sub-graphs considered in a collection cycle are handled independently. Unfortunately, this design may cause a lack of efficiency.

Consider two "dead" sub-graphs G_x and G_y, which are processed in W_x and W_y respectively. Suppose G_x is processed earlier than G_y (i.e. $x < y$) and there exists a one way path from G_y to G_x (i.e. weakly-connected), as shown in Figure 3. Both G_x and G_y would be expected to be reclaimed in the same collection cycle. However, due to the isolation design, G_x will not be reclaimed in this collection cycle even though it will become garbage after the reclamation of G_y. That is, during a collection cycle, a dead sub-graph (considered in a work unit) will not be reclaimed in the current collection cycle if there exists a path from any object considered in a subsequent work unit.

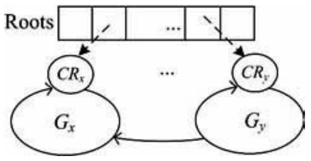


Figure 3. Two weakly connected sub-graphs considered in two different work units

In the above case, the reclamation of G_x will be postponed to a subsequent collection cycle, which clearly reveals a negative impact on performance as more potential traces over G_x would be required. It also affects the *completeness* of the algorithm.

More precisely, let H be an object graph that contains a set of unreachable (dead) sub-graphs considered in the same collection cycle, such that for each pair of sub-graphs G_i and G_j in H, there is a path from G_j to G_i if G_i is processed earlier than G_j in the collection cycle. The full reclamation of H requires more than one collection cycle, depending on the number of sub-graphs included in H. Assume H consists of k sub-graphs. The total number of involved work units for collecting H is

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2},$$

which is quadratic in terms of the number of passes over a dead sub-graph and is actually the worst-case.

3.3 The Solution

A main idea to address the problem pointed out in section 3.2 is to keep track of the number of external pointers (i.e. the *alpha* value (α)) for each sub-graph considered in a collection cycle, so that the collector can determine whether a previously processed sub-graph is still alive when reclaiming a dead sub-graph.

For this, we organized a two-dimensional table, named *External Pointer Table* (**EPT**), to store the α values for the sub-graphs considered during a collection cycle. In addition to the α value, each row in EPT also contains the root of the corresponding sub-graph. Each time an α value reaches zero, the collector performs a reclamation procedure starting from the root of the corresponding sub-graph.

More precisely, let G be a sub-graph that has just been traced for a given work unit. At this point, if $\alpha(G) \neq 0$, both $\alpha(G)$ and the root of G will be stored in EPT. Then, $\alpha(G)$ is monitored (by the collector) until it reaches zero or the collection cycle finishes, in which $\alpha(G)$ may be updated (decreased) because of the deletion of any (external) pointer coming from objects outside of G. During the collection cycle, G will be identified as garbage, as soon as $\alpha(G)$ becomes zero.

In the previous example in Figure 3, after G_x is traced, EPT must retain $\alpha(G_x)$ and CR_x since $\alpha(G_x) = 1$ (trivial). G_y is subsequently detected as garbage, since $\alpha(G_y) = 0$ (trivial). When reclaiming G_y, the collector frees each object reachable from CR_y such that the object was labeled with $ID(CR_y)$. It is clear that, in the meantime, the only pointer to G_x, say p, must be examined and removed eventually. Let T be the target object of p. Since T is contained in G_x instead of G_y, $ID(T) = ID(CR_x)$; this means that T is supposed to be freed with the reclamation of G_x. In this case, deleting the pointer p will cause $\alpha(G_x)$ in EPT to be decremented to "zero", where the collector can easily find the corresponding row in EPT for G_x because both objects T and CR_x were labeled with the same *ID*. That is, the collector can now identify G_x as garbage during the reclamation of G_y, since $\alpha(G_x)$

becomes zero after the deletion of p. Another reclamation procedure starting from CR_x can thus be performed for collecting G_x.

Consequently, any dead sub-graph in the same computation graph can be reclaimed in the same collection cycle. In the worst case that we analyzed for the original algorithm (LW), the number of involved work units is now limited to k. In general, the main benefit of the new algorithm is that more garbage cycles will be instantly reclaimed per collection cycle. In theory, it further avoids the potentially repeated traversals over a dead sub-graph. The efficiency is thus improved.

3.4 Pseudocode and Explanation

This section presents the pseudocode and explanations regarding the new cycle collection algorithm. The new algorithm is derived from the lightweight algorithm (LW) [11]. To have a better insight into the new algorithm, we choose not to duplicate some details that have been well addressed in [11] and only focus on describing the major refinements.

For ease of presentation, the labeling scheme, an integral part of LW, (which, as noted earlier, is employed to assist the graph traversal as well as to avoid revisiting objects per collection cycle) is assumed to be well settled. The details embedded in the traversal procedure are thus omitted here, including how the *alpha* value can be computed. The *ID* used for the labeling scheme is also maintained implicitly (by an increasing number actually). In addition, we assume the existence of some utility functions, which are all related to the manipulation of EPT (a table for managing the reachability information of previously processed sub-graphs). These include:

- EPT.add(*alpha*, *CR*): it adds a new row (*alpha*, *CR*) to EPT.
- EPT.update(S): it updates EPT by decrementing the *alpha* value of the sub-graph to which S belongs, and then the updated *alpha* value is returned. (It returns -1, in case EPT does not contain the row associated with the sub-graph to which S belongs.)
- EPT.getCR(S): it returns the root of the sub-graph to which S belongs.

Figure 4 shows the key procedures of the new algorithm. The input is the computation graph defined in section 3.1, and a global buffer for queuing candidate roots is called Roots. Invoking the procedure CollectCycles corresponds to triggering a collection cycle, which is aimed at processing *CR*s in Roots (with each being related to a work unit). For each work unit to be processed, the corresponding *CR* must be alive and have never been labeled in the current collection cycle. The process starts with a depth-first search (DFS) from the *CR*, which involves labeling the reached objects, computing the *alpha* value, and additionally setting up a flag *cyclicCR*. The flag is added to signify whether there is a back edge to *CR*, where the back edge is a pointer coming from an object visited in the DFS (including self-loops). Note that a "back edge" naturally implies a "cyclic structure". Hence, only the *CR*s really involved in cyclic structure may be retained in EPT, which can be seen at line 8 of CollectCycles (In the implementation, it is easy to detect a back edge through a DFS.).

After the DFS, the procedure CollectGarbage is invoked to reclaim cyclic garbage if the *alpha* value is zero; otherwise the *alpha* value and the *CR* are both added to EPT if *cyclicCR* is *true*. In the algorithm, CollectGarbage proceeds in a recursive fashion, which, in theory, may result in more than one sub-graphs being collected. During the execution of CollectGarbage, each object in the sub-graph (rooted at the *CR*) associated with the same *ID* (i.e. *DeadID*) is reclaimed as soon as it is reached; particularly all of the (outgoing) pointers from the dead sub-graph are treated separately (at line 7 of CollectGarbage), which is done by the procedure Decrement.

For a pointer examined by Decrement, its target object, say S, is first examined whether the reachability information for the sub-graph to which S belongs has been stored in EPT. If so, the *alpha* value of the sub-graph is decremented and then returned. If the returned value (i.e. *updated_alpha*) is zero, the sub-graph must be garbage and then another reclamation procedure (CollectGarbage) is performed, at line 3 of Decrement. Otherwise, $RC(S)$ is normally decremented, at line 5 of Decrement.

In summary, the new algorithm is different from the original one (LW) in two aspects. First, it incorporates a technique to monitor the number of external pointers for processed sub-graphs in the same collection cycle, solving the worst case of LW that we mentioned earlier. Second, in consequence of the above new scheme, the reclamation procedure is rewritten in a recursive form; both CollectGarbage and Decrement collaborate to update EPT and reclaim garbage. In contrast with that in LW, the new algorithm needs a DFS on dead sub-graphs. However, the extra buffer for queuing the references of potential garbage objects considered in a work unit is now no longer needed, thus reducing the space required in a collection cycle.

3.5 Proof of Correctness

Based on the original lightweight cycle collection algorithm (LW), the new algorithm proposed herein further reduces the potentially repeated scans over a dead sub-graph by tracking the reachability information of processed sub-graphs (per collection cycle). A rigorous proof of correctness for LW has been presented in [11], in which the difficult case that we mentioned earlier to cause the lack of *completeness* is described and addressed in greater detail. In this paper, we will not go into those details. Instead, we prove that the *safety* of the new algorithm remains.

One can think that the new algorithm optimizes LW and keeps the invariants unchanged (i.e. the special labeling scheme employed by LW). This, therefore, indicates that the following facts remain:

Remark 1. *Let R be a candidate root considered for cycle collection. When the DFS starting from R finishes, each visited object must be labeled by the same ID number.*

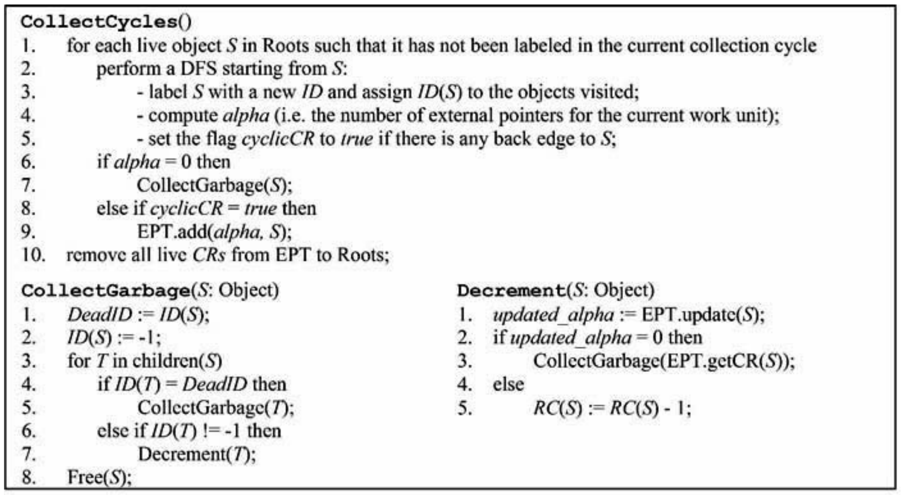
Remark 2. *Let R be a candidate root considered for cycle collection and G be the sub-graph induced by the objects labeled during the DFS starting from R. When the DFS finishes,* $\alpha(G)$ *is correctly computed and G must be garbage if* $\alpha(G) = 0$.

Proposition 1. *Let R be a candidate root considered for cycle collection and G be the sub-graph induced by the objects labeled during the DFS starting from R.* CollectGarbage(R) *can reclaim each object in G.*

Proof: By Remark 1, the objects labeled during the DFS starting from R must be assigned the same *ID* (i.e. $ID(R)$). Also, it can be easily seen that CollectGarbage(R) performs a simple recursive procedure that attempts to free those objects labeled by $ID(R)$ (i.e. *DeadID*). The proposition is thus proved.

Lemma 1. *During the execution of* Decrement*, if the updated alpha (i.e. updated_alpha) is zero, the sub-graph associated with this value must be garbage.*

Proof: The hypothesis implies that there must be an *alpha* value ever retained in EPT and later decremented to zero. Suppose G is the



```
CollectCycles()
1.  for each live object S in Roots such that it has not been labeled in the current collection cycle
2.      perform a DFS starting from S:
3.          - label S with a new ID and assign ID(S) to the objects visited;
4.          - compute alpha (i.e. the number of external pointers for the current work unit);
5.          - set the flag cyclicCR to true if there is any back edge to S;
6.      if alpha = 0 then
7.          CollectGarbage(S);
8.      else if cyclicCR = true then
9.          EPT.add(alpha, S);
10. remove all live CRs from EPT to Roots;

CollectGarbage(S: Object)
1.  DeadID := ID(S);
2.  ID(S) := -1;
3.  for T in children(S)
4.      if ID(T) = DeadID then
5.          CollectGarbage(T);
6.      else if ID(T) != -1 then
7.          Decrement(T);
8.  Free(S);

Decrement(S: Object)
1.  updated_alpha := EPT.update(S);
2.  if updated_alpha = 0 then
3.      CollectGarbage(EPT.getCR(S));
4.  else
5.      RC(S) := RC(S) - 1;
```

Figure 4. The New Cycle Collection Algorithm

sub-graph associated with the value, denoted $\alpha(G)$. We have initially $\alpha(G) > 0$, which means that there is at least one external pointer to G when processing G, at line 6 of CollectCycles. Let $\alpha(G) = m + n$, where m and n are the number of external pointers from, respectively, the system roots and the sub-graphs processed later than G in the current collection cycle. Since the external pointers from system roots are not involved in the computation graph of the algorithm, there is no possibility of tracking them. We hence have $m = 0$ in this case, and $n > 0$ (trivial). Consider any sub-graph G' that is processed later than G in the current collection cycle and is weakly connected to G such that there are k pointers from G' to G, $k \leq n$. Suppose G' is live. By Remark 2, it is clear that $\alpha(G') > 0$ when G' is processed. It also indicates that both CollectGarbage and Decrement will not be invoked for G' before the end of the current collection cycle. Also, Decrement is the only procedure able to update *alpha*. This clearly contradicts the hypothesis. Thus, G' must be garbage (unreachable from the system roots). That is, G cannot be reached from any of the system roots. G must be garbage.

Theorem 1 (Safety). *Only garbage objects are collected by the new algorithm.*

Proof: For each collection cycle (i.e. CollectCycles), CollectGarbage is the only place to reclaim objects, which, in the algorithm, can be invoked either when a DFS on a sub-graph finishes, yielding a zero *alpha*, or if an *alpha* value in EPT is decremented to zero. The former is the normal case (at line 7 of CollectCycles), in which the objects in the sub-graph reclaimed by CollectGarbage must be garbage, by Remark 2. In the latter case, a sub-graph associated with the *alpha* value is also reclaimed by CollectGarbage, at line 3 of Decrement. By Lemma 1, such a sub-graph must be garbage. The theorem is proved.

4 Experiments

In accordance with the experiments presented in [11], we have also implemented the new lightweight algorithm for evaluation of its effectiveness. We performed the experiments using the version 2.3.1 of Jikes Research Virtual Machine (Jikes RVM) [1] running on the platform: 1.4 GHz Pentium 4 and 512MB of physical memory running Linux 2.4.20; all the settings related to this experiment were configured as described in [11], unless otherwise noted.

All of our measurements were based on seven programs from the SPECjvm98 benchmark suite [21], with each running at the "problem size 100". We ran the seven benchmarks in RVM's default heap settings, starting with an initial size of 20 MB and allowing to grow to 100 MB. Each of the experiments was run five times and the average was reported. The measurement of mpegaudio is excluded since it does not perform meaningful allocation activity [7]. In fact, our execution of mpegaudio only triggers garbage collection twice. It means that the both triggers are due to invoking System.gc(), which is performed at the start and end of each SPECjvm98 benchmark.

We focus our attention on comparing the cycle collection operation for three algorithms: the original lightweight algorithm (LW) [11], the new lightweight algorithm (NLW) and the synchronous trial deletion algorithm (TD) [3]. In addition to cycle collection time and the total execution time of the application, they are also compared quantitatively in terms of several measures.

4.1 Performance

Figure 5 shows the cycle collection time ratio between using the lightweight algorithms and TD, which depicts both the ratio of TD to LW and the ratio of TD to NLW. Therefore, any ratio larger than 1 indicates that the lightweight one outperforms TD; when comparing LW and NLW, the higher ratio is better. In particular, the time reflects only the work for triggered cycle collections instead of triggered garbage collections. Notice that, in the implementation, the work of a triggered garbage collection includes not only collecting garbage cycles but also collecting non-cyclic garbage objects due to a particular reference counting implementation (i.e. deferred reference counting). A garbage collection is normally triggered when a low-memory condition occurs or System.gc() is invoked.

According to the results in Figure 5, both lightweight algorithms unsurprisingly deliver a substantial reduction in cycle collection time when compared to TD. That is because the tracing work actually dominates the amount of time spent on collecting garbage cycles, and also, unlike LW and NLW, TD always requires multiple passes over the computation graph, thus causing much more tracing overhead. In this experiment, the amount of tracing work, which you can see in column two of table 1 (addressed later), almost entirely reflects the improvement in cycle collection time. When comparing LW and NLW, there is no clear difference in terms of cycle collection time.

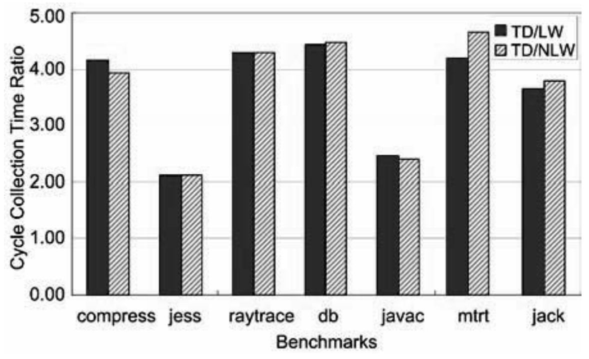


Figure 5. Cycle Collection Time Ratio

To explore the effect of the cycle collection algorithms on the application performance, we also report the execution time ratio, as shown in Figure 6. The results show that both lightweight algorithms outperform TD in almost all of the programs, at best 17% faster (raytrace). On average, LW and NLW are respectively 4% and 6% faster than TD. In particular, the dramatic improvement in the cycle collection of lightweight algorithms is not simply reflected in the execution time measure. That is because, in most of the programs, the time spent on cycle collection is relatively lower than the program's total execution time, and the overall garbage collection time depends on not only the cycle collection but also the implementation of the deferred reference counting. On the other hand, the results show that NLW is slightly better than LW in terms of the application performance.

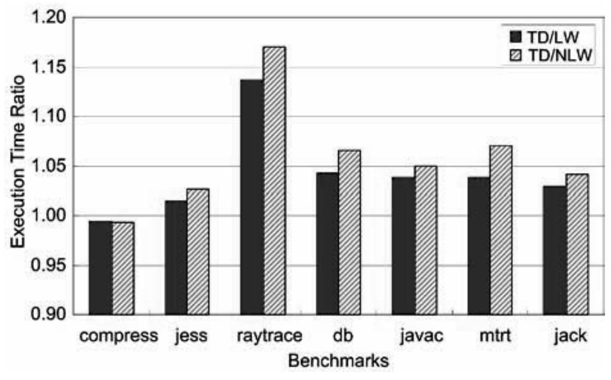


Figure 6. Execution Time Ratio

4.2 Cycle Collection

To get better understanding of the lightweight algorithms, we now present some more measurements. As shown in table 1, each main column, except for the last one ("Sub-graphs Reclaimed"), consists of three sub-columns, where the last two sub-columns separately normalize to the first one (TD).

First of all, the number of triggered cycle collections is very close among these algorithms, in which the number of triggers required in LW and NLW is at most two rounds more than that in TD (i.e. compress). As noted above, a garbage collection is generally triggered when a low memory condition happens. The slightly increased number of triggers thus indicates that the lightweight approach could be practical even if it may not collect all the cyclic objects immediately. More evidence for the practicality of the lightweight approach has been presented in [11].

Column two in table 1 shows the number of references visited when running the benchmarks. It reflects the amount of tracing work, and clearly the lightweight approach, including LW and NLW, considers substantially fewer references than TD. The reduction in tracing is up to 62% on average. As mentioned earlier, for any sub-graph considered for collection, TD needs to scan it three times, while the lightweight approach walks the sub-graph only in one round and at most twice (if the sub-graph is detected as garbage). That is, the lightweight algorithms can, in theory, save at most two passes over the same graph when compared to TD. In this case, the restoration phase of TD would take up a considerable portion of its tracing work.

Such a great reduction in tracing has also implied that not many cyclic objects would be freed when running the benchmarks, which can be seen in column three ("Cyclic Objects Reclaimed") of table 1, and actually has already been pointed out in [3]. Even so, it does not mean that the cycle collection makes no sense. For instance, few cyclic objects are actually reclaimed when running compress; however, those objects lead to many huge data structures, and if they can not be timely collected, the program will run out of memory.

Unsurprisingly, the number of cyclic objects reclaimed by LW and NLW is lower than that by TD, which reflects the lack of *completeness* that we mentioned earlier. In particular, the amount of cyclic garbage found varies widely among programs, at the range from 99% (compress) to 29% (jack) when compared with TD. This is quite application dependent. Moreover, the results show that only two programs (jess and jack) benefit from NLW, where the cyclic objects reclaimed by NLW is more than that by LW. Such a difference can be better seen in terms of the number of sub-graphs reclaimed (the last column of table 1). Apparently, the difference is relatively noticeable for jack, in which NLW significantly detects 25% more dead sub-graphs than LW, without incurring more tracing overhead.

Overall, the experiments show that the lightweight approach indeed provides a substantial reduction in tracing when compared to TD, which brings a great improvement in cycle collection time and thus has better application performance, even if some garbage cycles appear not to be timely collected. In addition, though not all of the programs greatly gain from NLW, the results also demonstrate that NLW can improve the *completeness* of LW, and even results in a better performance. This could be very application dependent.

5 Conclusions and Future Work

In this paper, a new cyclic reference counting algorithm based on the lightweight approach has been presented, including a better description in terms of its computation graph, pseudocode and proof of correctness. Conceptually, the new algorithm exploits the main hypothesis from the lightweight approach that considers a single sub-graph, instead of individual cycles, as the basic unit of cycle collection, and further optimizes the original one by eliminating redundant scans over garbage objects. An additional contribution is the improvement of the *completeness*, in which more garbage cycles would be instantly reclaimed per collection cycle. Moreover, our experiments not only provide more insight into the comparison of the lightweight approach and the trial deletion, but also show that the new algorithm significantly outperforms both the original lightweight algorithm as well as the trial deletion algorithm.

The proposed algorithm is compatible with the prior improvements on trial deletion. Those studies on how to better choose more significant

Table 1. Cycle Collection

Program	Cycle Collections			Refs. Traced			Cyclic Objects Reclaimed			Sub-graphs Reclaimed	
	TD	LW (norm)	NLW (norm)	TD	LW (norm)	NLW (norm)	TD	LW (norm)	NLW (norm)	LW	NLW
compress	10	1.2	1.2	158834	0.30	0.36	127	0.99	0.99	51	51
jess	20	1.00	1.00	2685825	0.66	0.66	5	0.40	0.80	2	3
raytrace	8	1.00	1.00	4922468	0.25	0.25	2	0.50	0.50	1	1
db	7	1.00	1.00	1538155	0.31	0.31	4	0.75	0.75	3	3
javac	13	1.08	1.08	5827995	0.53	0.55	2	0.50	0.50	1	1
mtrt	8	1.13	1.13	4556576	0.26	0.23	2	0.50	0.50	1	1
jack	11	1.00	1.00	735178	0.34	0.32	5456	0.29	0.36	80	106
mean	11	1.05	1.05	2917862	0.38	0.38	799	0.56	0.63	20	24

candidates can be naturally incorporated. We also plan to investigate adapting the algorithm to a concurrent cycle collector in the future. Furthermore, if the algorithm could be adapted to a distributed environment or, any environment that involves many disk accesses (e.g. the object-oriented database). The benefits from the lightweight algorithm would be clearer since the delay due to the network communication (sending messages) or the disk I/O is relatively small.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen, ”Implementing Jalape˜no in Java,” In Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Denver, Colorado, ACM SIGPLAN Notices, vol. 34(10), pp. 314-324, ACM Press, October, 1999.

[2] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith, ”Java without the coffee breaks: A nonintrusive multiprocessor garbage collector,” In Proceedings of the ACM SIGPLAN'01 Conference on Programming Languages Design and Implementation (PLDI), Snowbird, Utah, vol. 36(5), pp. 92-103, June 2001.

[3] D. F. Bacon and V. T. Rajan, ”Concurrent cycle collection in reference counted systems,” In J. L. Knudsen, editor, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, Springer-Verlag, LNCS 2072, pp. 207-235, June, 2001.

[4] T. W. Christopher, ”Reference count garbage collection,” Software Practice and Experience, vol. 14(6), pp. 503-507, 1984.

[5] G. E. Collins, ”A method for overlapping and erasure of lists,” Communications of the ACM, vol. 3(12), pp. 655-657, December, 1960.

[6] J. Detreville, ”Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64,” DEC Systems Research Center, Palo Alto, California, 1990.

[7] S. Dieckmann and U. Hölzle, ”A study of the allocation behavior of the SPECjvm98 Java benchmarks,” In Proceedings of ECOOP'99, Springer-Verlag, LNCS 1628, pp. 92–115, 1999.

[8] R. E. Jones and R. D. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, July, 1996.

[9] R. E. Jones and R. D. Lins, ”Cyclic weighted reference counting without delay,” In Proceedings of Parallel Architectures and Languages Europe (PARLE'93), A. Bode, M. Reeve, and G. Wolf, Eds., Springer-Verlag, LNCS 694, pp. 712-715, June 1993.

[10] Y. Levanoni and E. Petrank, ”An on-the-fly reference counting garbage collector for Java,” In ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, Tampa, FL, pp. 367-380, October, 2001.

[11] C.-Y. Lin and T.-W. Hou, ”A lightweight cyclic reference counting algorithm,” In Proceedings of the first International Conference on Grid and Pervasive Computing, Taichung, Taiwan, May 3-5, Springer-Verlag, LNCS 3947, pp. 346-359, 2006.

[12] R. D. Lins, ”An efficient algorithm for cyclic reference counting,” Information Processing Letters, vol. 83, pp. 145-150, 2002.

[13] R. D. Lins, ”Generational cyclic reference counting,” Information Processing Letters, vol. 46, pp. 19-20, 1993.

[14] R. D. Lins, ”Cyclic reference counting with lazy mark-scan,” Information Processing Letters, vol. 44, pp. 215-220, 1992.

[15] R. D. Lins and R. E. Jones, ”Cyclic weighted reference counting,” in: K. Boyanov (Ed.), Proceedings of International Workshop on Parallel and Distributed Processing, NH, pp. 369-382, May, 1993.

[16] M. Maeda, H. Konaka, Y. Ishikawa, T. Tomokiyo, A. Hori, and J. Nolte, ”On-the-fly global garbage collection based on partly mark-sweep,” Springer-Verlag, LNCS 986, pp. 283-296, 1995.

[17] A. D. Martinez, R. Wachenchauzer and R. D. Lins, ”Cyclic reference counting with local mark-scan,” Information Processing Letters, vol. 34, pp. 31-35, 1990.

[18] J. H. McBeth, ”On the reference counter method,” Communications of the ACM, vol. 6(9), pp. 575, September, 1963.

[19] J. McCarthy, ”Recursive functions of symbolic expressions and their computation by machine,” Communications of the ACM, vol. 3, pp. 184-195, 1960.

[20] H. Paz, E. Petrank, D. F. Bacon, V. T. Rajan, and E. K. Kolodner, ”Efficient on-the-fly cycle collection,” In Proceedings of the 14th International Conference on Compiler Construction, R. Bodik, editor, Springer- Verlag, LNCS 3443, pp. 156-171, April, 2005.

[21] Standard Performance Evaluation Corporation, ”SPECjvm98 Documentation”, release 1.03 edition, March, 1999.

[22] X. Ye and J. Keane, ”Collecting cyclic garbage in distributed systems,” In International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'97), Taipei, Taiwan, pp. 227-231, December, 1997.