Distributed Cyclic Reference Counting *

Frank Dehne †
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
dehne@scs.carleton.ca

Rafael D. Lins ‡
Departamento de Informática
Univ. Federal de Pernambuco
50732-970 Recife, Pe, Brasil
rdl@di.ufpe.br

Abstract

We present a distributed cyclic reference counting algorithm which incorporates both, the correct management of cyclic data structures and the improvement of lazy mark-scan. The algorithm allows processors to run local mark-scan simultaneously without any need of synchronisation between phases of different local mark-scans either on the same processor or on different processors.

1 Introduction

In distributed memory multiprocessors, each processor is responsible for reclaiming unused structures residing in its local memory (distributed garbage collection). As in the case of uni-processors, the algorithms are usually based on (global) mark-scan or on reference counting [5].

A number of algorithms use *(global) mark-scan* in distributed architectures [11, 1, 12]. The major disadvantage of these methods is that, as in the sequential case, the application is suspended during the garbage collection phase. One attempt [21] to improve this uses dual processors on each local memory and transfers objects with non-local references. This can however

*This work was done while the first author was visiting the Universidade Federal de Pernambuco, Brasil.

†Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

‡Research partially supported by CNPq research grant no. 40.9110/88.4.

create very high communication overhead, depending on the size of the objects being transfered, and it may be unable to reclaim large cyclic structures that span over several processors.

Reference counting has the advantage that it is performed in small steps interleaved with application computation and, hence, does not need to suspend the application. The major drawback of standard reference counting [6] is its inability to reclaim cyclic data structures. For the sequential domain, Friedman & Wise [8], Bobrow [4], and Hughes [13] have solved this problem in the context of implementing Lisp and functional languages such as Miranda. A general uniprocessor algorithm for cyclic reference counting with local mark-scan was presented in [19] and substantially improved in [17].

Two algorithms made standard reference counting suitable for use in loosely-coupled multiprocessor architectures: weighted reference counting [3, 24] and generational reference counting [9]. However , as in sequential standard reference counting, these both algorithms are also not able to reclaim cyclic data structures. A multiprocessor algorithm that merges weighted reference counting with Lins' cyclic reference counting with lazy mark-scan [17] was described in [18]. This method can reclaim cyclic data structures. Plainfossé and Shapiro [20] pointed out that this algorithm is simple but does not allow several processors to invoke mark-scan simultaneously. A first attempt to solve this problem is presented in [14]. However, this new algorithm needs global synchronisation between the phases of the local mark scan procedures on the same processor as well as on different processors.

In this paper, a distributed cyclic reference counting algorithm is presented. This algorithm allows processors to run local mark-scan simultaneously without any need of synchronisation between phases of different local mark-scans either on the same processor or on different processors. It incorporates both, the correct management of cyclic data structures [19] and the improvement of lazy mark-scan [17]. As shown in [17], lazy mark-scan evaluation can considerably reduce the garbage collection overhead by avoiding unnecessary local mark-scans.

We concentrate on the distributed computing issues of the algorithm without taki ng into account the choice of a suitable communication protocol as described in [16]. As an alternative, the algorithm presented can easily be modified to work with weighted reference conting [3, 24].

The remainder of this paper is organized as follows. In Section 2 we present our algorithm and in Section 3 we prove its correctness.

2 The Algorithm

For reference counting with local mark-scan, the basic interface between the application and the garbage collection consists of three procedures New(R), Copy(R,<S,T>), and Delete(<R,S>) used by the application to allocate a new cell, copy a new pointer, and delete a pointer, respectively. Unused cells, i.e. cells which can not be referenced any more by the application, are organised in a *free list*.

Every cell S has a reference couter RC(S) which counts the number of pointers to S as well as a colour which could be yellow, green, black, red, pink , or blue. All cells in the free list are coloured yellow.

The garbage collection code uses three data structures. A control heap CH is used for implementing the lazy evaluation scheme. It is a data structure created locally on each processor. mr-Q(S) denotes a queue of mark-red(S) processes attempting to mark red a cell S , and rec-Q(S) denotes a queue of reclaim(S) processes, where procedures mark-red(S) and reclaim(S) are described below. The mr-Q(S) and rec-Q(S) for all cells S on one processor can be easily implemented as one data structure, each.

A call of Delete(<R,S>) spawns an independent process $\text{reclaim}_p(S)$ on the same processor which attempts to recover unused cells. The index p denotes a process number given to the reclaim process, and it will later be used to determine if this process has terminated. For the remainder, all indices attached to procedure calls refer to such process numbers.

New(R) =
  select a new cell S from the free-list
  RC(S):=1
  colour(S):=green
  mr-Q(S):=nil
  rec-Q(S):=nil
  create a pointer from R to S

Copy(R,<S,T>) =
  create a pointer from R to T
  RC(T):= RC(T)+1

Delete(<R,S>) =
remove the pointer <R,S>
RC(T):= RC(T)-1
spawn a new process reclaim$_p$(S)
for some new process number p

Procedure reclaim$_p$(S) checks if the reference counter of S is zero, in which case S can be immediately linked to the free list. Otherwise, a reference to S is added to the control heap CH (if not already ther e) for later evaluation on whether S is unreachable by the application. Sons(S) refers to all cells T such that there exists a pointer from S to T.

reclaim$_p$(S) =
if colour(S) is green or black then
if RC(S)=0 then
for all cells T in Sons(S) do
Delete(<S,T>)
colour(S):=yellow
link S to the free-list
else
if colour(S) is green then
colour(S):=black
add S to the control heap CH
else
add p to rec-Q(S)

Control Heap CH
Continuously select the *next best* S from CH and do:
remove S from CH
if colour(S) is black then
spawn a process mark-red$_{p_1}$(S)
for some new process number p_1
suspend until no active process with number p_1 exists
spawn a process scan$_{p_2}$(S)
for some new process number p_2
suspend until no active process with number p_2 exists
spawn a process collect-blue(S)

When a cell S is taken from the control heap CH, there are three processes started at S: mark-red$_{p_1}$(S), scan$_{p_2}$(S), and collect-blue(S). As these processes spawn other subprocesses, the process numbers are used to determine their termination.

Procedure mark-red$_{p_1}$(S) paints red S and all cells in the subgraph of cells reachable from S. It also decrements the reference counters of the cells visited, so that fianl reference counts are associated only with pointers from outside the subgraph (external references).

```
mark-red_p(S) =
    if colour(S) is green or black then
        colour(S):=red
        for all cells T in Sons(S) do
            mark-red(T)
        for all cells T in Sons(S) do
            RC(T):=RC(T)-1
    else
        add p to mr-Q(S)
```

scan$_p$(S) seaches the subgraph now painted red and repaints green all cells reachable from external references. All other cells are painted blue.

```
scan_p(S) =
    if colour(S) is red then
        if RC(S)>0 then
            colour(s) := pink
            spawn a process scan-green_p(S)
        else
            colour(S):=blue
            for all cells T in Sons(S) do
                spawn a process scan_p(T)
```

```
scan-green_p(S) =
    for all cells T in Sons(S) do
        RC(T):= RC(T)+1
        colour-green(T)
    for all cells T in Sons(S) do
        if colour(T) is red or blue then
            spawn a process scan-green_p(T)


colour-green(S) =
    if mr-Q(S) not nil then
        colour(S) := red
        take q from mr-Q(S)
        spawn a process mark-red_q(S)
    else
        colour(S) := green
        if rec-Q(S) not nil then
            take q from rec-Q(S)
            spawn a process reclaim_q(S)
```

Procedure collect-blue(S) links to the free list all cells that are blue after the previous $\text{scan}_p(S)$ has terminated.

```
collect-blue(S) =
    if colour(S)=pink then
        colour-green(S)
    if colour(S)=blue then
        colour(S):=yellow
        Temp := Sons(S)
        link S to free-list
        for all cells T in Temp do
            spawn a process collect-blue(T)
```

3 Proof of Correctness

We now show the correctness of our distributed reference counting algorithm. In particular, we show that (a) if some cell S is collected for the free list,

then S is not in use and (b) every unused cell is collected for the free list.

Lemma 1 *If colour(S)=green then RC(S) has the actual value.*

Proof. When S is created by a New(R), then S is coloured green and RC(S) has the actual value. The only procedures possibly changing RC(S) are Copy, Delete, mark-red, and scan-green. Obviously, changes made by Copy and Delete are always correct. A mark-red(S) changes colour(S) to red and, hence, the assumption of this lemma does not hold for S. Assume that S is coloured green by colour-green(S) called within scan-green(S). We observe that scan and scan-green can only reach nodes which have previously been reached by a mark-red started at some deleted pointer, and that this entire mark-red process has been terminated. Hence, RC(S) has first been decremented by one in a mark-red(S), and is now being incremented by one in a scan-green(S). Assuming, by induction, that RC(S) was correct before mark-red(S) changed it, it has now again the actual value. □

Lemma 2 *If colour(S) is not green then in finite time either*

(a) colour(S) will change to green (if S is in use) or

(b) colour(S) will change to yellow (if S is not in use) and S will be collected for the free list.

Proof. (a) Assume S is in use. If colour(S)=red then a scan process will colour it blue, green, or pink by usin g the same path as the mark-red process which coloured S red. If colour(S)=blue then it lies on a cycle and is reachable from a cell that is in use. Hence, a subsequent scan-green process will colour S green. If colour(S)=black then it is on the control heap CH, and after finite time it will be taken from CH and coloured red. As shown above, all red cells (in use) are eventually coloured green. If colour(S)=pink and S is part of a cycle, then the scan-green process called by the scan that coloured S pink will colour S green. If colour(S)=pink and S is not part of a cycle, the collect-blue phase following the scan that marked S pink will colour S green.

(b) Assume S is not in use. If colour(S)=blue then the collect-blue process following the scan that made S blue will change it to yellow and collect it for the free list. If colour(S)=red then the subsequent scan process will change it to blue, and from there it will be changed to yellow as indicated above. If colour(S)=black then it is on the control heap CH, and after finite

time it will be taken from CH and coloured red. As shown above, all red cells (not in use) are eventually coloured yellow and collected for the free list. If colour(S)=pink and S is not in use, then the mark-red originated at the pointer whose deletion made S useless has not yet reached S. However, it is reachable from that pointer and will therefore be set to red by that mark-red process. □

Lemma 3 *If colour(S)=black then RC(S) has the actual value.*

Proof. The only procedure to change colour(S) to black is reclaim, and reclaim changes colour(S) from green to black. Hence, by Lemma 1, RC(S) has the actual value. For black cells, only Copy and Delete can change their reference counters, and both procedures obviously keep the reference counter at the actual value. □

Theorem 1 *If S is collected for the free list, then S is not in use.*

Proof. S can be collected either by reclaim or by collect-blue. Reclaim deletes S only if colour(S) is green or black and RC(S)=0. Hence, it follows from Lemma 1 and Lemma 3 that S is not in use. S is collected by collect-blue only if colour(S)=blue and collect-blue(S) is called. This implies that the previous mark-red and scan phases for the same deleted pointer have terminated (see Control Heap CH). Thus, the previous scan marked S blue and could not repaint it green. Therefore, S is part of an unreachable cycle. □

Theorem 2 *Every cell not in use is collected for the free list.*

Proof. Assume that cell S is not in use. If colour(S) is not green then, by Lemma 2, after finite time its colour will be set to yellow and S will be collected for the free list. Assume that colour(S)=green, then S is reachable from a cell T for which a reclaim(T) is still in process. Hence, this reclaim(T) process will change colour(S). □

4 Conclusion

We presented a distributed cyclic reference counting algorithm which solves the previously open problem of performing cyclic reference counting which recovers unused cyclic structure and at the same time allowing multiple such processes without synchronisations between these processes.

References

[1] K.A.M.Ali. *Object-oriented storage management and garbage collection in distributed processing systems.* PhD thesis, Royal Institute of Technology, Stockholm, December 1984.

[2] M.Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[3] D.I.Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 176–187. Springer Verlag, LNCS 259, June 1987.

[4] D.G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3): 269–273, March 1980.

[5] J.Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[6] G.E. Collins A method for overlapping and erasure of lists *Communications of the ACM*, 3(12): 655-657, 1960.

[7] E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten & E.M.F.Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of ACM*, 21(11):966–975, November 1978.

[8] D.P.Friedman and D.S.Wise. Reference counting can manage the circular environment of mutual recursion. *Information Processing Letters*, 8(1):921–930, January 1979.

[9] B.Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Proceedings of SIGPLAN'89 Conference on Programming Languages Design and Implementation*, pages 313–321. ACM Press, June 1989.

[10] D.Gries. An exercise in proving parallel programs correct. *Communications of ACM*, 20(12):921–930, December 1977.

[11] P.Hudak and R.M.Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings of 1986 ACM*

Conference on Lisp and Functional Programming, pages 168–178, Pittsburg, August 1982.

[12] R.J.M.Hughes. A distributed garbage collection algorithm. In J. P. Jouannaud (Ed.), *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS 201, pages 256–272, 1985.

[13] R.J.M.Hughes. Managing reduction graphs with reference counts. Departmental Research Report CSC/87/R2, University of Glasgow, March 1987.

[14] R. Jones and R.D.Lins Cyclic weighted reference counting without delay In *In PARLE'93 Parallel Architectures and Languages Europe*, Springer Verlag, LNCS 694, Arndt Bode and Mike Reeve and Gottfried Wolf Editors, pp 512–515, June 1993.

[15] H.T.Kung and S.W.Song. An efficient parallel garbage collection system and its correctness proof. In *Proc. IEEE Symposium on Foundations of Computer Science*, pa ges 120–131, 1977.

[16] C-W.Lermen and D.Maurer. A protocol for distributed reference counting. In *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, pages 343–350, Cambridge, Massachusetts, August 1986.

[17] R.D.Lins. Cyclic reference counting with lazy mark-scan. *Information Processin Letters* 44:215–220, 1992.

[18] R.D.Lins and R. Jones Cyclic weighted reference counting In K.Boyanov (Ed.), Parallel and Distributed Processing'93 (WP&DP'93), Bulgarian Academy of Sciences, Sofia, 1993, pages 369–382, to be published by North Holland.

[19] A.D.Martinez, R.Wachenchauzer and R.D.Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[20] D.Plainfossé and M. Shapiro. Experience with a fault-tolerant garbage collector in a distributed Lisp system. in Y. Bekkers and J.Cohen (Eds.) Proceedings of *Memory Management - International Workshop*, St. Malo, France, 1992, volume LNCS 637, pages 116–133. Springer-Verlag, 1992.

[21] M.Shapiro, O.Gruber and D.Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, Rapports de Recherche, INRIA-Rocqencourt, November 1990.

[22] G.L.Steele. Multiprocessing compactifying garbage collection. *Communications of ACM*, 18(09):495–508, September 1975.

[23] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud (Ed.), *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS 201, pages 1–16, 1985.

[24] P.Watson and I.Watson. An efficient garbage collection scheme for parallel computer architectures. In *In PARLE'87 Parallel Architectures and Languages Europe*, Springer Verlag, LNCS 259, pages 432–443, June 1987.