Cyclic Weighted Reference Counting without Delay

Richard E. Jones and Rafael D. Lins*

Computing Laboratory
University of Kent at Canterbury
Canterbury, England

email: rej,rdl@ukc.ac.uk
Phone: +44 227 764000 ext. 7550
FAX: +44 227 762811

11 November 1992

Abstract

Weighted Reference Counting is a low communication distributed storage reclamation scheme for loosely-couple multiprocessors. The algorithm we present herein extends weighted reference counting to allow the collection of cyclic data structures. To do so, the algorithm identifies candidate objects that may be part of cycles and performs a tricolour mark-scan on their subgraph in a lazy manner to discover whether the subgraph is still in use. The algorithm is concurrent in the sense that multiple useful computation processes and garbage collection processes can be performed simultaneously.
Keywords: Memory management, Distributed memory, Reference counting, Garbage collection.

Introduction

Computation on distributed systems involving several processors is already a reality. In a distributed multiprocessor system each processor is responsible for allocating and reclaiming structures residing in its local memory; interprocessor communication is far less efficient than local memory access. In order to make this approach viable for a much larger number of applications in computer science, a number of problems must be addressed. One of these fundamental problems is how to manage distributed memory with low communication costs, and with acceptable space-time overheads.

We present an improved algorithm for distributed memory management, based on reference counting yet able to collect cycles. Our scheme was inspired by two algorithms. Weighted Reference Counting [35; 1] makes reference counting practical for use in distributed systems by reducing the amount of communication necessary; Lazy Cyclic Reference Counting [19] extended the standard reference counting algorithm [9] to deal with cyclic structures simply and efficiently in a uni-processor environment. David Plainfossé and Marc Shapiro [28] emphasised that an earlier version of our algorithm [21] was simple but not truly concurrent — under certain circumstances, some of the mutators were suspended during garbage collection. Furthermore it was not possible to allow all processing nodes to garbage collect independently and asynchronously.

*also: Dept.de Informática - U.F.PE. - 50.739 - Recife - PE - Brazil

In this paper we address these problems in a scheme for Cyclic Weighted Reference Counting without Delay, a simple concurrent algorithm for cyclic reference counting on loosely-coupled multi-processor architectures. Section 1 examines other proposals for distributed memory management; sections 2 and 3 examine weighted reference counting and lazy cyclic reference counting in more detail before we present our new algorithm in section 4. We set out the requirements our algorithm makes of the communication network in section 6 and summarise our work in section 7.

1 Background

To solve the problem of managing distributed memory, several authors have proposed distributed algorithms based on mark-scan garbage collectors. Hudak and Keller's algorithm for garbage collection and task deletion in distributed applicative systems works in two steps [14]. First, all processors co-operate to mark all accessible cells, and then all processors collect irrelevant tasks and unmarked cells. Although marking is a process global to all processors, it can be performed concurrently with normal execution. This algorithm makes extensive use of mechanisms for locking or releasing cells, and queuing processes including an arbitrary number of marking tasks, which is expensive in both processing time and in space used for buffering.

Mohamed-Ali presented two different approaches to the problem, using variants of mark-scan [25]. 'Local' garbage collection is performed independently and asynchronously. On completion each local garbage collector informs all other processors of the remote pointers it retains, and they then treat these as roots that must be marked during their own local garbage collection. No attempt is made to recover cycles that span processing nodes. Such cycles are dealt with by a 'global' garbage collection, during which useful computation is suspended. Hughes describes a similar algorithm which is capable of recovering cyclic structures [15]. His algorithm has lower storage overheads than Mohamed-Ali's, although it is likely to take longer to recover remotely-referenced garbage. None of these algorithms is concurrent.

Shapiro *et al.*'s algorithm uses mark-scan garbage collection locally in each processor [30]. External references are avoided by transferring an object between processors whenever a local mark-scan discovers that there are no locally held references to the object. Eventually all members of a cycle will be transferred to the same processing element and can be dealt with by a local mark-scan. Although this increases locality of reference, large objects would present high communication overheads for this algorithm.

However for all distributed memory systems, mark-scan is an expensive process. Marking requires substantial communication between processors and it is necessary to detect termination. It is also difficult, though not impossible, to make mark-scan algorithms incremental. Reference counting suggests itself as a naturally attractive method for real-time storage management in distributed systems. The relative merits of reference counting *vis à vis* garbage collection have been thoroughly discussed (see, for example, [8]). Nevertheless it is of interest for a number of reasons.

- It is incremental. Reclamation is performed in small steps interleaved with computation.
- It has good locality of reference.
- It does not degrade with occupancy (unlike mark-scan garbage collection).
- It is relatively simple to implement and prove correct, even in distributed systems.

On the other hand, reference counting suffers from a number of problems. The major drawback of the standard reference counting algorithm is its inability to reclaim cyclic data structures [24]. Several authors have solved this problem in the context of implementing Lisp and functional languages on uni-processors [11; 3; 16]. Language independent algorithms based on Brownbridge's notion of partitioning the set of pointers into strong and weak (cycle-closing) pointers have also been proposed. However these have either been incorrect [4], non-terminating in pathological cases [29] or extremely expensive to implement [26; 33]. A more promising scheme for uni-processors has been developed and substantially improved by Lins and his colleagues [23; 19]. A drawback of Lins' algorithm is that it is not concurrent. In order to collect cyclic structures, Lins must perform a *local* mark-scan garbage collection on the appropriate subgraph. In essence, his can be thought of as a hybrid algorithm — it switches between reference counting and mark-scan garbage collection. It is important to realise that Lins' mark-scan is local rather than global (as in the traditional mark-scan algorithm). Lins simply mark-scans the transitive closure of cells that are suspected to be part of an isolated cycle (and hence garbage).

Highly parallel distributed systems complicate storage reclamation. A regime based upon the standard reference counting algorithm requires messages to be sent whenever a pointer to a remote object is copied or deleted. Special care must be taken to avoid an object being reclaimed while references to it still exist. This may happen if reference counting operations are not executed in the order in which they were spawned, for instance if the message deleting the last reference to an object overtakes a copying message. Lermen and Maurer describe a communication protocol which provides a correct distributed reference count scheme but at the cost of three messages per inter-processor reference [18].

The Weighted Reference Counting algorithm makes reference counting practical for use in loosely-coupled multiprocessor architectures [1; 35]. by significantly reduces communication overhead, only sending messages between processors when references to remote objects are deleted. No synchronisation between messages is required. Goldberg's Generational Reference Counting algorithm [12] has the same communication overhead as Weighted Reference Counting. However, as Goldberg admits, the major drawback of his algorithm in relation to weighted reference counting is its space efficiency. None of these algorithms is able to deal with cyclic structures.

Recently Lang *et al.* proposed an elegant system which combines mark-scan for local garbage collection and reference counting for handling remote references [17]. To perform global garbage collections, processing nodes are dynamically grouped together, and Christopher's algorithm is used to remove dead cycles [6]. Unfortunately, details that are important for a full understanding are omitted.

2 Weighted Reference counting

In this section we explain the original weighted reference counting algorithm [1; 35]. Non-atomic *objects* (or *cells*) have a number of fields which store pointers to other objects. In the standard reference counting algorithm, each object has an additional field, the *reference count*, which holds the number of pointers which refer to this object [9]. In the weighted reference counting algorithm, a *weight* is associated with each pointer. A cell's reference count field contains the total weight of all pointers that refer to the cell. We denote a pointer from a cell `R` to cell `S` by `<R,S>`, its weight by `Weight(<R,S>)` and the reference count of cell `S` by `RC(S)`. For all cells N, X the following invariant is maintained

$$RC(N) = \sum_{N} Weight(< X, N >)$$

For simplicity we assume that the graph formed by the objects in use has a single starting point, which we call *root*. All cells in use (*active* cells) are transitively connected to root. Similarly we also assume that *free* cells are held in a *free-list*. Any cells that cannot be reached either from the free-list or by tracing pointers from root are *garbage*. We describe the algorithms in this paper in terms of three primitive operations:

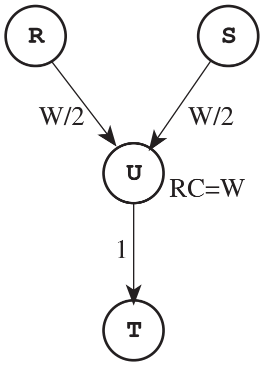
`New(R)` gets a fresh cell `U` from the free-list, initialises its reference count to some arbitrary (fixed) value W and links it into the graph. The weight of the pointer `<R,U>` is also set to W.

`Copy(R,<S,T>)` links cell `R` to `T` by duplicating the pointer `<S,T>`. The pointer's weight is split equally between `<R,T>` and `<S,T>`. Thus the invariant is maintained without having to communicate with `T` to change its reference count.

Delete (<R,S>) deletes the pointer <R,S>. In order to maintain the invariant, a message is sent to the target cell S instructing it to decrement its reference count by the weight of the pointer <R,S>. If the reference count drops to zero, the cell cannot be in use any longer and so is returned to the free-list.

Delete is the only operation that requires communication. An added benefit of the algorithm is that these messages do not need to be synchronised.

From this description it can be seen that pointer weights are powers of 2. This permits a practical technique for implementation: each pointer stores the *logarithm* of its weight. Indirection cells are used when copying pointers of weight one: to execute Copy(R,<S,T>) when Weight(<S,T>) = 1, an indirection U cell is created (in the same processing element as S so that no communication is necessary). The indirection cell simply contains a pointer to the target T — the pointer's weight is one so need not be stored. R and S are both set to refer to the indirection cell, each with pointer weight $W/2$. Notice that the reference count of T need not be changed — no communication is necessary.



Piquer's algorithm avoids the need for indirection cells by maintaining indirect reference counts at each node [27]. Like weighted reference counting it does not send increment messages, and decrement messages do not require synchronisation. The chief drawback to both these methods is their inability to deal with cyclic structures. Lins' algorithms, which we examine next, provide a sequential solution to this problem.

3 Lazy Cyclic Reference Counting

A promising scheme for uni-processors has been developed by Lins and his colleagues [23]. Its most sophisticated form, the Lazy Cyclic Reference Counting algorithm, combines reference counting with lazy four-colour mark-scan garbage collection [19]. The New and Copy operations are similar to those of the standard reference counting algorithm, as is the deletion of the last reference to an object. However, if the target of the pointer being deleted is shared (it has reference count greater than one), then it may be part of an isolated cycle and hence garbage. In this case, the colour of the object is changed from *green* (active) to *black* (a candidate for garbage collection) and a reference to it is placed on a *control queue*. No further action is taken until either the free-list becomes empty or the control queue is full.

When either of these events occur, cells are popped from the control queue until a black one is found (it will become apparent below how a non-black cell may appear in the control queue). Cells in the transitive closure of this cell are then marked and scanned to find any external references (i.e. from cells external to this subgraph). If none are found then the cells in the subgraph are garbage and can be returned to the free-list.

Garbage collection proceeds in three phases. In the first phase, mark_red, all cells in the closure are painted *red*. Each time a cell is visited by mark_red its reference count is decremented. On completion of this phase only those cells that are the (direct) target of an external reference will have non-zero reference counts.

The task of the second phase, `scan`, is to discover any such cells. These cells and their descendants are re-painted green by `scan_green` which also increments the reference count of each cell it visits. All other cells are painted *blue*. Finally, the third phase, `collect_blue`, returns all blue cells to the free-list.

It is important to realise that Lins' only marks, scans and collects from the subgraph whose root is the cell popped from the control queue. A *global* mark-scan of the entire active data structure is not necessary. His algorithm takes advantage of the observation that, for LISP and functional languages at least, the overwhelming majority of objects have small reference counts [7; 32]. It is also likely that by the time that a cell is popped from the control queue its colour will have reverted to green in which case it is not subjected to garbage collection. (This is an optimisation that only applies to a sequential environment; it is not applicable if messages may be delayed indefinitely.) Its last reference may have been deleted, in which case it is either in the free-list or has been recycled, but either way it is green. Alternatively, if it is active, a pointer to it may have been copied in which case the cell will be painted green. Finally it may already have been examined by an earlier garbage collection and reprieved (painted green) by `scan_green`.

Lins' algorithm suffers from two deficiencies which it shares with other garbage collection algorithms. First, reclamation of garbage cycles may be delayed indefinitely. Standard reference counting, on the other hand, recycles cells as soon as they become garbage although of course it cannot reclaim cycles. In order to collect cyclic structures, Lins must perform a *local* mark-scan garbage collection on the appropriate subgraph. In essence, his can be thought of as a hybrid algorithm — it switches between reference counting and mark-scan garbage collection. It is important to realise that Lins' mark-scan is local rather than global (as in the traditional mark-scan algorithm). Lins simply mark-scans the transitive closure of cells that are suspected to be part of an isolated cycle (and hence garbage). Nevertheless, this delay may be controlled by appropriate management of the control set (see section 5). Again unlike the standard reference counting algorithm, Lins' algorithm is not truly concurrent: 'useful' processing must stop for local mark-scan garbage collection. Our Cyclic Weighted Reference Counting scheme also suffers from the former deficiency (although to a lesser extent) but not the latter — it is truly concurrent.

4 Cyclic Weighted Reference Counting without Delay

Our new algorithm combines the benefits of standard Weighted Reference Counting (SWRC) — low communication overheads — and Lins' Cyclic Reference Counting (CRC) — efficient handling of cycles. As in SWRC pointers have weights and cells have reference counts. We modify the condition on the invariant so that it is guaranteed to hold when no processor is performing garbage collection. Cells are assigned one of three colours *green, red, blue*; the meanings are the same as in Lins' algorithm. In addition, each cell also has a second reference count which is only used by the garbage collector. Instead of using a control queue, we use a set. Although a set is more expensive to maintain, it has the advantage of greater precision: we wish to avoid unnecessary mark-scans as far as possible. Since the evidence suggests that the control set will be comparatively small, the cost of maintaining it is worth the price [22].

We impose the usual condition that every cell except root is initially on the free-list. For the moment, we say nothing about the characteristics required of the communications protocol (they are few), nor of how the control set should be managed (there are several possibilities). We return to these matters later.

New

`New` behaves in the same way as in SWRC and in CRC. A new cell is allocated from the free-list, given an arbitrary (fixed) weight W and painted green. The cell is connected to the graph by a pointer whose weight is also W. (We abstract away from the details of the synchronisation required to manage the free-list.)

```
New(R) =
    allocate(U)
    RC(U) = W
    Weight(<R,U>) = W
    colour(U) := colour(R)
    SRC(U) := 0
```

where `allocate(U)` obtains a fresh cell `U` from the free-list, `green` paints a cell or pointer green, `RC(U)` is the reference count of `U`, `SRC(U)` is the secondary reference count and `Weight(<R,U>)` is the weight of the pointer `<R,U>`. We take care to simulate the effect of any garbage collection phase that might be in operation.

Copy

`Copy(R,<S,T>` links `R` to `T` by copying pointer `<S,T>`. It is similar to the SWRC scheme unless `S` is currently subject to garbage collection; we discuss this problem later. There are also several optimisations that we might consider adding to our scheme. One would be to short-circuit indirection cells (since it must communicate with the target cell in any case).

```
Copy(R,<S,T>) =
    if Weight(<S,T>) > 1
        Weight(<S,T>) := Weight(<S,T>) / 2
        Weight(<R,T>) := Weight(<S,T>)
    else
        allocate(U)
        RC(U) := W
        SRC(U) := 0
        colour(U) := colour(S)
        Weight(<U,T>) := 1
        Weight(<S,U>) := W / 2
        Weight(<R,U>) := W / 2
        insert(U, control_set)
    if red(S) and mark_red_phase
        send retract(<S,T>) to T
```

Delete

`Delete(<R,S>)` is treated in a similar same way to Lins' CRC. A message is sent to the target cell `S` causing it to decrement its reference count by the weight of the deleted pointer. If the count becomes zero, `Delete` is called on the sons of the cell. The cell is then returned to the free-list by `free`. Otherwise the cell becomes a candidate for garbage collection and is added to the control set.

```
Delete(<R,S>) =
    send Message_Delete(<R,S>) to S
    remove <R,S>

Handle_Delete(<R,S>) =
    RC(S) := RC(S) - Weight(<R,S>)
    if RC(S) = 0
```

```
            for T in Sons(S)
                send Message_Delete(<S,T>) to T
            green(S)
            free(S)
        else
            insert(S, control_set)
```

Mark-scan

When a pointer to a shared object is deleted, a reference to the target is added to the control set rather than being mark-scanned immediately (as in [23]). Lins' idea was to use the control queue to delay the mark-scan in the hope that it will prove unnecessary to garbage collect the cell and its subgraph: either its last reference will have been deleted or another call to `scan_green` will prove that it is still active. This approach has proved successful. Lins and Vasques have found that, with appropriate management of the control queue, no unnecessary calls to mark-scan are made: garbage collection was only performed on cells that proved to be members of cycles [22]. Our incremental algorithm also uses the control set in a manner similar to Steele's algorithm [31]. Steele used a stack for communication between mutator and collector: whenever the connectivity of the graph was changed, a reference to the target cell was pushed onto the stack. Our algorithm requires much less communication between mutator and collector. Like Steele we keep a record of targets: our algorithm may be classified as incremental update, write-barrier based. The difference is that we just record the target of `Delete` operations, and then only if the target is shared.

We discuss control set strategy later; for the moment let us assume that the garbage collector `gc` must be called periodically to remove items from the control set and initiate a garbage collection is on the subgraph of which this cell is the root. It is possible that other processing elements may also wish to perform garbage collection. Care must be taken to ensure that simultaneous garbage collections do not interfere with each other thereby losing colour and weight information.

Again there are several approaches that may be taken. One option would be to require the processing elements to take it in turn to initiate garbage collection. No processor would be allowed to call `mark_red` before the node currently garbage collecting had completed `collect_blue`. This could easily be implemented by imposing an order on nodes in the network and using a token-passing mechanism.

A less restrictive method is to synchronise the phases of garbage collection [20]. In the `mark_red` phase, any processor that wishes to do so is allowed to initiate marking; no processor is allowed to start either scanning or collecting until there are no processors marking red. Notice that only a subset of processing elements need be involved in this garbage collection at any time. When the `mark_red` phase is complete (including `retract`, the processors garbage collecting move onto the scan phase. The rules for this phase are similar, and likewise for the final `collect_blue` phase. We discuss the mechanism necessary for this synchronisation later.

```
gc() =
    S := pop(control_set)
    RENDEZVOUS
    mark_red(S)
    RENDEZVOUS
    scan(S)
    RENDEZVOUS
    collect_blue(S)
```

`mark_red` traces the transitive closure of its starting point, reddening cells and initialising secondary reference counts as it does so. As each pointer is traversed, the secondary reference count of its target cell is decremented by the weight of the pointer. On completion, red cells will only have a non-zero secondary reference count if there are references to them from cells external to the subgraph that have not been visited by this, or another, `mark_red`. We emphasise that only a subset of the graph is marked (in contrast to standard marking algorithms). The subgraph may span processing elements. The invariant for *secondary* reference counts becomes:

> 'the secondary reference count of a red cell is equal to its reference count less the sum of the weights of pointers to it which have been traced by `mark_red`.'

```
mark_red(S) =
    if not red(S)
        redden(S)
        SRC(S) := RC(S)
        for T in Sons(S)
            send Message_mark_red(<S,T>) to T

Handle_mark_red(<S,T>) =
    if not red(T)
        redden(T)
        SRC(T) := RC(T) - Weight(<S,T>)
        for U in Sons(T)
            send Message_mark_red(<T,U>) to U
    else
        SRC(T) := SRC(T) - Weight(<S,T>)


Handle_retract(<S,T>) =
    RC(T) := RC(T) + Weight(<S,T>) / 2
```

Copying a pointer from a red cell during a `mark_red` garbage collection phase introduces a subtle problem. `mark_red` would have decremented the target's secondary reference count by the weight of the pointer, but this value has now been halved. The SRC invariant would be violated. `retract` is used during the `mark_red` phase to preserve the invariant.

Having removed the effect of pointers internal to the subgraph from secondary reference counts, `scan` is called to search for cells with external references (non-zero secondary reference counts). Such cells must be considered to be roots of active subgraphs and be re-painted green. This is done by `scan_green`. Any cells not visited by `scan_green` are garbage and are marked blue accordingly.

```
scan(S) =
    if red(R)
        if RC(S) > 0
            scan_green(S)
        else
            blue(S)
            for T in Sons(S)
                send Message_scan(T) to T

Handle_scan(T) =
    scan(T)


scan_green(S) =
    if not green(S)
        green(S)
        for T in Sons(S)
            send Message_scan_green(<S,T>) to T
```

```
Handle_scan_green(<S,T>) =
    scan_green(T)
```

Finally, `collect_blue` is called to return all blue (garbage) cells to the free-list. Any pointer from a blue cell to a non-blue (green) one must be removed and the reference count diminished appropriately.

```
collect_blue(S) =
    if blue(S)
        for T in Sons(S)
            send Message_collect_blue(<S,T>) to T
            remove <S,T>
    free(S)

Handle_collect_blue(<S,T>) =
    if blue(T)
        collect_blue(T)
    else
        Handle_Delete(<S,T>)
```

Aside: the algorithms for `scan` and `collect_blue` have been presented recursively. The alternative would be to let each processing element scan its heap iteratively, looking for red (blue) cells, thus removing the need for any inter-processor communication.

5 Control set strategy

Lins and Vasques examined several strategies for managing the control queue in a sequential context [22]. They obtained the best results by dynamically allocating the queue in the heap rather than using a fixed length queue (various sizes in proportion to total heap memory were tried). They also examined a number of access disciplines for the control queue such as FIFO, LIFO, mark-scanning the entire queue at once instead of one element at a time. The preferred method was FIFO access. They also found that, with appropriate management of the control queue, no unnecessary calls to mark-scan-collect were made: all cells on which garbage collection was invoked proved to be members of cycles.

In their sequential uni-processor environment, `gc` was called either by `New` when a new cell was requested but the free-list was empty, or by `Delete` after it had pushed a reference onto the control queue and thus filled the queue. We could choose to use the same mechanism in a multi-processor environment. Another possibility would be to run `gc` regularly (for example when the length of the free-list drops below a pre-determined size).

A more attractive alternative that would give a truly real-time performance might be to run `gc` continuously on a second processor [10; 20]. In this way, 'useful' processing would not be disrupted. By using a combination of reference counting and mark-scan in this way we expect to obtain a better performance than the mutator-collector architectures analysed in [34] and [13]. Notice that even with a single processor, we are able to run garbage collection incrementally. It is not necessary, for example, to run `mark_red` to completion before returning to useful processing; we could allow it to execute one cell at a time.

Finally, notice that cells are only added to the control set by `Delete` and are only removed by `gc`. Since `Delete` requires communication with the processing element which contains the target cell, we can arrange that cells are added to the control set on their local processor. Thus the control set can be distributed: each control set only contains references to cells on the same node. When a cell is returned to the free-list, `free` should remove any reference to it from the control set.

6 Network

The standard Weighted Reference Counting algorithm only required communication between nodes when a reference to a (remote) object was deleted. In that case a message was sent to reduce the reference count of the target cell by the weight of the pointer. The advantage of that scheme is that no synchronisation is needed.

Our garbage collection scheme requires that the ends of its three phases be synchronised: no processor is allowed to initiate the next stage if another process is still working on the current phase. This can be accomplished by using either of two multicast protocols. If one processor is nominated to control synchronisation, it can keep track of the number of processors working on the current phase of garbage collection. As processors complete a phase, they should send a message to this distinguished processor indicating that they have completed the phase and requesting permission to continue to the next. When there are no more processors working on the current phase, the controlling processor should broadcast permission to move to the next phase. This can be accomplished by using an atomic ordered multicast protocol [5]. Alternatively a more symmetric multicast technique could be adopted which does not require the assignment of a single distinguished processor: each node should replicate the count of processors working on the current garbage collection phase and all requests to proceed to the next phase should be broadcast to all processors. Birman's three-phase atomic ordered multicast protocol would be adequate for this purpose [2].

7 Discussion

We have presented a storage reclamation scheme for loosely-coupled multi-processors based upon weighted reference counting that is able to reclaim cyclic data structures. In order to reclaim cycles, a mark-scan-collect process is performed only upon the *subgraph* whose root is suspected to be part of a garbage cycle. Lins has demonstrated that, with appropriate tuning, this can be done efficiently. Our scheme also allows useful processing to continue during mark-scan-collect: it is not a stop-and-wait method. Colours and second reference counts are required to support garbage collection which gives our scheme higher storage overheads than the standard reference counting algorithm. For computations that do not create cycles, our scheme has communication overheads identical to those of the standard weighted reference counting algorithm, namely communication is only required when a reference to a remote object is deleted. Where cycles are present, our garbage collection scheme must trace subgraphs over the network[1]. The three phases of garbage collection must be synchronised, which requires atomic ordered multicast capability of the communications protocol. In contrast, the standard weighted reference algorithm makes no demands of the protocol. We believe these costs are an acceptable price to pay for the ability to reclaim cycles.

Hudak and Keller's algorithm has one significant advantage over ours — it is able to detect and remove irrelevant tasks. However, our algorithm has a number of advantages over theirs. It only requires mark-scan to be performed on the relevant subgraph rather than the entire active data structure. Hudak and Keller must trace the entire graph twice before scanning the whole heap. We visit each cell in the subgraph three times in the worst case (once by `mark-red`, once by `scan_green` or `collect_blue` and at most once by `scan`). Other active cells are unaffected, as indeed are those other processors which hold no part of this subgraph. This is also in contrast to Lang *et al.*'s algorithm which similarly visits garbage cells but also marks all live cells; indeed pointer paths within a node may be traced more than once. Traditional marking schemes by definition do not visit garbage cells. On the other hand, on-the-fly schemes such as Dijkstra's and Steele's do mark garbage and in this respect our algorithm is no worse. Evidence suggests that, for LISP and functional languages at least, much storage reclamation will be done without recourse to garbage collection. Hudak and Keller also make extensive use of mechanisms for locking or releasing cells, and queuing processes including an arbitrary number of marking tasks. This is expensive in both processing time and in space used for buffering. Neither Mohamed-Ali's nor Hughes' algorithms are truly real-time since any particular computation may be delayed for a long time

[1] But even in this case we can reduce the amount of tracing involved if we know that the target of a pointer is an atomic object and so cannot be part of a cycle. This information could be coded into pointers in many although not all cases.

while its processor does a garbage collection. Hughes says that his algorithm may take a very long time indeed to recover remotely-referenced garbage in pathological cases. The major disadvantage of Ali's method is that it cannot deal with cycles.

Acknowledgements

Research reported herein has been sponsored jointly by the British Council HED link, CNPq (Brazil) grants No 40.9110/88.4 and 46.0782/89.4, and CAPES (Brazil) grant 2487/91-08.

References

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

[2] K. Birman. Exploiting virtual synchrony in distributed systems. *ACM Operating Systems Review*, 21(5):123–138, November 1987.

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

[4] David R. Brownbridge. Cyclic reference counting for combinator machines. In J.-P. Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, Berlin, 1985. Springer-Verlag.

[5] J. Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2:251–273, August 1984.

[6] T.W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[7] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, February 1977.

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

[9] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[10] Edsgar W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[11] D.P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inf Process. Lett.*, 8(1):41–45, Jan. 1979.

[12] Benjamin 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.

[13] T. Hickey and Jacques Cohen. Performance analysis of on-the-fly garbage collection. *Communications of the ACM*, 27(11):1143–1154, Nov. 1984.

[14] Paul R. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsnurgh, Pa.*, pages 168–78, August 1982.

[15] R. John M. Hughes. A distributed garbage collection algorithm. In J.-P. Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume LNCS 201, pages 256–272. Springer-Verlag, 1985.

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

[17] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming, Albuquerque*, pages 39–50, January 1992.

[18] C-W. Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, pages 343–350, Cambridge, Massachusetts, August 1986. ACM SIGPLAN/SIGACT/SIGART.

[19] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. Technical Report 75, The University of Kent at Canterbury Computing Laboratory, The University, Canterbury, Kent, July 1990. Submitted to Information Processing Letters.

[20] Rafael D. Lins. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 38:31–35, September 1992.

[21] Rafael D. Lins and Richard E. Jones. Cyclic weighted reference counting. Technical Report 95, The University of Kent at Canterbury Computing Laboratory, The University, Canterbury, Kent, December 1991.

[22] Rafael D. Lins and Márcio A. Vasques. A comparative study of algorithms for cyclic reference counting. Technical Report 92, The University of Kent at Canterbury Computing Laboratory, The University, Canterbury, Kent, August 1991. Submitted to Software Practice and Experience.

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

[24] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

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

[26] E.J.H. Pepels, M.C.J.D. van Eekelen, and M.J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical report 88–10, Computing Science Department, University of Nijmegen, 1988.

[27] J. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts *et al.*, editor, *PARLE'91 Parallel Architectures and Languages Europe*, Berlin, 1991. Springer Verlag, LNCS 505.

[28] David Plainfossé and Marc Shapiro. Experience with fault-tolerant garbage collection in a distributed Lisp system. In *Proceedings of International Workshop on Memory Management, St. Malo, France*, INRIA, France, September 16–18 1992.

[29] Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[30] Marc Shapiro, O. Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, November 1990. Also in ECOOP/OOPSLA'90 Workshop on Garbage Collection.

[31] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[32] Will R. Stoye, T.J.W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas*, pages 159–66, August 1984.

[33] Simon J. Thompson and Rafael D. Lins. Cyclic reference counting: a correction to Brownbridge's algorithm. unpublished notes, 1988.

[34] Philip L. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–608, September 1987.

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