REPORT CSG-35 AUGUST, 1984

COORDINATED SCIENCE LABORATORY
COMPUTER SYSTEMS GROUP

# PARALLEL GARBAGE COLLECTION WITHOUT SYNCHRONIZATION OVERHEAD

ASHWIN RAM
JANAK H. PATEL



REPORT R-1019 UILU-ENG 84-2213

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>N/A |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>N/A | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>R-1019 (CSG-35); UILU-ENG 84-2213 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>N/A |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Coordinated Science Laboratory<br>University of Illinois<br>6b. OFFICE SYMBOL (If applicable)<br>N/A | 7a. NAME OF MONITORING ORGANIZATION<br>Joint Services Electronics Program |
| 6c. ADDRESS (City, State and ZIP Code)<br>1101 West Springfield Avenue<br>Urbana, IL 61801 | 7b. ADDRESS (City, State and ZIP Code)<br>Office of Naval Research<br>800 North Quincy Street<br>Arlington, VA |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>Joint Services Electronics Prog.<br>8b. OFFICE SYMBOL (If applicable)<br>N/A | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>N00014-84-C-0149 |
| 8c. ADDRESS (City, State and ZIP Code)<br>Office of Naval Research<br>800 North Quincy Street<br>Arlington, VA | 10. SOURCE OF FUNDING NOS.<br>PROGRAM ELEMENT NO.: N/A<br>PROJECT NO.: N/A<br>TASK NO.: N/A<br>WORK UNIT NO.: N/A |
| 11. TITLE (Include Security Classification) Parallel Garbage Collection Without Synchronization Overhead | |
| 12. PERSONAL AUTHOR(S)<br>Ram, Ashwin and Patel, Janak H. | |
| 13a. TYPE OF REPORT<br>Technical<br>13b. TIME COVERED<br>FROM _____ TO _____ | 14. DATE OF REPORT (Yr., Mo., Day)<br>August 1984<br>15. PAGE COUNT<br>17 |
| 16. SUPPLEMENTARY NOTATION<br>N/A | |
| 17. COSATI CODES<br>FIELD / GROUP / SUB. GR. | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)<br>symbolic processing, LISP, LISP machines, garbage collection, parallel garbage collection |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Incremental garbage collection schemes incur substantial overhead which is directly translated as reduced execution efficiency for the user. Parallel garbage collection schemes implemented via time-slicing on a serial processor also incur this overhead, which might even be aggravated due to context switching. It is useful, therefore, to examine the possibility of implementing a parallel garbage collection algorithm using a separate processor operating asynchronously with the main list processor. The overhead in such a scheme arises from the synchronization necessary to manage the two processors, maintaining memory consistency.

In this paper, we present an architecture and supporting parallel garbage collection algorithms designed for a virtual memory system with separate processors for list processing and for garbage collection. Each processor has its own primary memory; in addition, there is a small common memory which both processors may access. Individual memories

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐ | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE NUMBER (Include Area Code)<br>22c. OFFICE SYMBOL<br>N/A |

19. ABSTRACT, continued

swap off a common secondary memory, but no locking mechanism is required. In particular, a page may reside in both memories simultaneously, and indeed may be accessed and modified freely by each processor. A secondary memory controller ensures consistency without necessitating numerous lockouts on the pages.

# PARALLEL GARBAGE COLLECTION WITHOUT SYNCHRONIZATION OVERHEAD

Ashwin Ram
Janak H. Patel

Technical Paper R-1019 (CSG-35); UILU-ENG 84-2213

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, Illinois 61801

1984

Acknowledgment: This work was supported by the Joint Services Electronics Program under Contract N00014-84-C-0149.

# Parallel Garbage Collection Without Synchronization Overhead

Ashwin Ram
Janak H. Patel

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

## Abstract

Incremental garbage collection schemes incur substantial overhead which is directly translated as reduced execution efficiency for the user. Parallel garbage collection schemes implemented via time-slicing on a serial processor also incur this overhead, which might even be aggravated due to context switching. It is useful, therefore, to examine the possibility of implementing a parallel garbage collection algorithm using a separate processor operating asynchronously with the main list processor. The overhead in such a scheme arises from the synchronization necessary to manage the two processors, maintaining memory consistency.

In this paper, we present an architecture and supporting parallel garbage collection algorithms designed for a virtual memory system with separate processors for list processing and for garbage collection. Each processor has its own primary memory; in addition, there is a small common memory which both processors may access. Individual memories swap off a common secondary memory, but no locking mechanism is required. In particular, a page may reside in both memories simultaneously, and indeed may be accessed and modified freely by each processor. A secondary memory controller ensures consistency without necessitating numerous lockouts on the pages.

## 1. Introduction

List processing systems use dynamic storage management techniques, and it is desirable to ease the burden on the user by providing automatic storage allocation and reclamation. An excellent survey of various garbage collection algorithms can be found in [Cohen81]. There are two basic alternatives for automatic reclamation of storage: reference counting and mark-and-trace methods. The reference count method [Deutsch76] cannot detect self-referential structures inaccessible from the outside; in addition, space and time overheads are incurred in maintaining these counts. Mark-and-trace garbage collection [Knuth73], on the other hand, traditionally caused a complete disruption of computation while storage was reclaimed, which could take a substantial amount of time. Attempts to distribute the garbage collection time over useful list processing time to make it less unbearable led to incremental garbage collection algorithms

[Deutsch76] and parallel garbage collection algorithms [Baker78, Dijkstra78, Hibino80, Lamport76, Lieberman83, Newman82, Steele75]. However, incremental schemes, as well as parallel schemes implemented via time-slicing on a serial computer, suffer from substantial overhead which is less noticeable since it gets distributed over useful processing time, but annoying nevertheless. The source of the overhead is not completely obvious; in extended virtual memory systems, the garbage collection routines could increase page swapping by substantial amounts.

True parallel garbage collection systems dedicate a separate processor to garbage collection. While these overcome the CPU time overhead for garbage collection, they suffer from the necessity of synchronization between the two processors. If the processors share primary memory, they must have exclusive access to the memory; in addition, the paging overhead caused by the garbage collector affects directly the performance of the list processor. If the two processors maintain separate primary memories, pages from secondary memory must be locked out from simultaneous access to avoid memory inconsistency.

In this paper, we present an architecture and supporting parallel garbage collection algorithms designed for a virtual memory system with separate processors for list processing and garbage collection. Each processor has its own primary memory; in addition, there is a small common memory which both processors may access. Individual memories swap off a common secondary memory, but no locking mechanism is required. In particular, a page may reside in both memories simultaneously, and indeed may be accessed and modified freely by each processor. A secondary memory controller ensures consistency without necessitating numerous lockouts on the pages.

## 2. Overview of the system

The proposed system, as shown in Figure 1, consists of two independent processors, the *list processor* (LP) (often known as the mutator), and the *garbage collector* (GC). These processors have independent primary memories, in which pages of a common virtual space are maintained. In addition, there is a peripheral processor called the *disk controller* (DC), through which the LP and GC interface to the common secondary memory. The LP and GC may work on their own local copies of the same page of the virtual

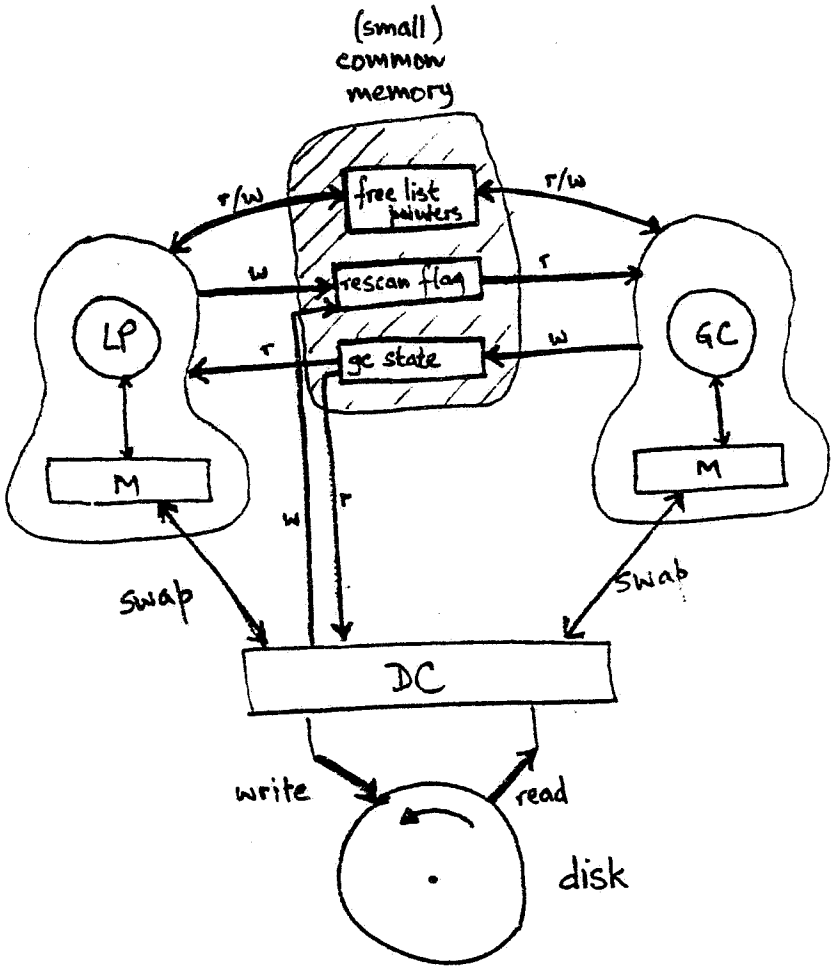


Figure 1: System architecture

address space. Swapping is done through the DC, which ensures that a page written to secondary memory by either processor is *merged* with the version of the page that is already there (and which may have been altered by the other processor during the time it resided in the swapping processor's primary memory). The merging process resolves differences between the two versions of the page, and guarantees memory consistency.

## 2.1. Terminology

We adopt LISP terminology to refer to data objects and operations. The basic data object is a fixed size *list cell*, or simply *cell*. A list cell stores two pointers, called the *CAR* and *CDR*. *Lists* are created by chaining list cells into binary (spanning) trees, the leaves of which are the *atoms* in the system. Primitive operations are provided to access the pointers in a cell (*CAR* and *CDR*), test equality of two pointers (*EQ*), test whether an object is an atom or a list cell (*ATOM*), create a new list cell with specified pointer fields (*CONS*), and replace the CAR or CDR pointers in a cell (*REPLACA* and *REPLACD* respectively). Cells are accessed by tracing the list structures starting from one of the *roots* in the system. A root is typically an entry on the stack, or an atom on the OBLIST (which stores pointers to all the atoms in the system, thus guaranteeing their accessibility and uniqueness). A cell that cannot be reached from any root is *garbage*, and needs to be returned to the pool of unused cells (the *free list*) so that it can be reallocated if needed.

## 2.2. The List Processor

The LP needs to perform basic list processing primitives on its data. These correspond to the LISP functions CONS, CAR, CDR, RPLACA, RPLACD, EQ and ATOM. In our system, CAR, CDR, EQ and ATOM present no particular difficulty; the interesting operations are CONS and REPLACE (which includes RPLACA and RPLACD). CONS causes a new list cell to be allocated from the free list, and returns a pointer to this cell. REPLACE replaces the pointer address in (one of the fields of) a cell, causing it to point to a cell different from the one to which it pointed before the operation (which may possibly become garbage).

## 2.3. The Free List

The unallocated or *free* cells in the system are chained together in a *free list*, each pointing to the next. New cells are allocated from this list, and unused cells are returned to this list after they become garbage. The LP may request new cells via CONS at the same time as the GC is returning unused cells to the free list. To allow these two operations to be performed simultaneously, separate pointers to the head and the tail of the free list are kept. Allocations are done from the head, while reclaimed cells are returned

to the tail. The head and tail pointers are kept in the common memory, and may be accessed simultaneously. The only case in which the LP needs to wait for the GC is when the free list runs out, i.e., the memory is exhausted; in such a case, however, the user program is probably too large for the available virtual space anyway.

### 2.4. The Garbage Collector

The GC processor uses a basic mark-and-sweep algorithm to trace all the reachable cells in the system. The collection is performed in two phases:

1) The *mark phase*, in which all the reachable cells in memory are marked.

2) The *reclaim phase*, in which all the unmarked (and hence unreachable) cells are returned to the free list.

Since the GC maintains its own local primary memory, the LP need incur no overhead due to the computation performed by the GC. In addition, page faults caused when the GC traces through the list structures marking reachable cells are localized to the GC's primary memory, and do not cause swapping overhead as far as the LP is concerned.

### 2.5. The Disk Controller

Since the LP and the GC maintain local copies of virtual memory pages, they could simultaneously modify memory contents which could ostensibly lead to inconsistency. This seeming inconsistency is easy to resolve, however, since the manipulations performed by the LP and the GC are constrained by the very definitions of their tasks. The DC has the responsibility of maintaining consistency by *merging* a page when it is written out by the LP or the GC with the version that is currently on disk. The merging operation can be performed with little overhead given appropriate hardware.

The merging algorithm depends on whether the page is being written by the GC or the LP; in addition, it depends on the phase of the GC (i.e., mark or reclaim). A flag for the GC's phase is therefore kept in the common memory area, which is accessed on a mutually exclusive basis by the GC and the DC. However, the GC need access the flag only when it changes from one phase to another, a relatively

infrequent situation, and thus there is almost no synchronization overhead due to this.

### 2.6. Cell colorings

List cells in the system are *colored* to indicate their status. We need two bits per cell to store the color. These are assigned as follows:

**Green:** A cell which is on the free list is unallocated, and is colored green.

**Black:** A black cell is known to be accessible from one of the roots in the system.

**Yellow:** During the mark phase of the GC, cells that have been reached but have yet to be traced or explored further are colored yellow.

**White:** During the reclaim phase of the GC, cells that are white are inaccessible and therefore garbage.

## 3. Algorithms for the processors

We now develop the algorithms used by the three processors, and indicate how they interact. Extensions to the basic algorithms to improve performance will be treated in the next section.

### 3.1. The List Processor

As mentioned above, the primitive list operations that the LP needs to perform are CONS, CAR, CDR, REPLACA, REPLACD, EQ and ATOM. Since the GC performs no relocation, CAR, CDR, EQ and ATOM present no difficulty. We now consider CONS and REPLACE (RPLACA and RPLACD) in detail.

#### 3.1.1. The CONS primitive

The CONS function gets an unused cell from the free list and returns a pointer to this cell. To do this, it gets the next green cell, following the free list head pointer. This cell is blackened to mark it as being reachable, and the free list head pointer is updated to point to the next free cell on the list.

To avoid synchronization conflicts with the GC (which may be returning cells to the free list), we allocate cells from the head of the free list and return reclaimed cells to the tail of the free list. The head and tail pointers can be accessed independently. The only situation in which the LP needs to wait for the

GC is when the free list is empty, which is obviously unavoidable.

### 3.1.2. The REPLACE primitives

A REPLACE operation modifies one of the pointers in a cell X to point to another cell Z. The cell that was originally pointed to may become garbage as a result of this operation.

If the GC is in the mark phase, the cell Z should be marked as being reachable. If Z is already black, nothing need be done, otherwise it should be marked yellow (since it may need to be traced further). The LP must also indicate to the GC that it needs to trace the cell further. There are several ways in which this could be implemented. The simplest method is to set a *rescan* flag, and to let the GC continually scan the memory until there is no rescan request. This process will eventually halt as all the reachable cells get colored black. Modifications of this algorithm to reduce rescanning and page faults are examined in the next section.

If the GC is in the reclaim phase, Z would already have been marked as being reachable. If the page Z lies on has already been processed by the GC, it would have been marked white, otherwise it would be black (this is explained in the next section). In either case, the LP does not need to alter the color of Z. The LP could check which phase the GC is in (thus avoiding the need to access Z in the reclaim phase), but this would require a mutually exclusive access to a common flag. Since the LP and DC require read-only access to this flag, we need only ensure that neither reads the flag while the GC is altering it. This is guaranteed with current memory technology since it provides indivisible reads and indivisible writes. The GC changes from one phase to another only at long intervals, and therefore this would cause negligible overhead.

## 3.2. The Garbage Collector

The GC is a separate processor dedicated to the task of garbage collection. It has its own primary memory; in addition, it can access the common memory. The GC continually cycles through two phases, the *mark phase* and the *reclaim phase*. There is a flag in the common memory which indicates the phase that the GC is in (as described in the previous section).

### 3.2.1. Mark Phase

In the mark phase, the GC scans the memory linearly looking for reachable cells. At the outset, all cells are colored white, except for the cells on the free list which are green. To start off, all the roots in the system, i.e., cells which are directly reachable, such as atoms in the oblist and stack variables, are marked yellow. The GC then scans the memory linearly from top to bottom, looking for yellow cells. From each yellow cell, the GC traces the list structure recursively, terminating its scan whenever it reaches the end of the list or a cell that has already been marked black. Each cell thus visited is marked black. The GC continues this linear scan of the memory as long as the rescan flag is set, taking care to reset the flag before each scan.

At this point, there are no yellow cells in secondary memory, since they have all been explored and marked black. However, the LP could have yellow cells in its own primary memory on pages that had not been written out to disk after the cells were colored yellow. The list structures starting at these cells must also be traced. To do this, the GC now scans the LP's memory looking for yellow cells. Starting at the addresses of these yellow cells, the GC traces the list structures within its own memory, marking them black as before. If in this time the LP writes out a yellow cell to disk (detected by the DC), the cell will have to be traced further in the usual way. In a naive scheme this involves another iteration of both parts of the mark phase.

In the next section, we explore some alternative schemes with the intention of eliminating unnecessary rescans, as well as reducing the page faults incurred by the GC during the marking process.

### 3.2.2. Reclaim Phase

In the reclaim phase, the GC scans the memory linearly from top to bottom, looking for white cells, which did not get marked during the mark phase. These cells are colored green and returned to the tail of the free list. All other cells are either reachable or known to be on the free list, and hence would be either black or green after the mark phase. The black cells are colored white in preparation for the next GC cycle; the green cells are left unaltered. The GC should ensure it scans a page completely before it returns it to disk.

After the reclaim phase is over, the GC must write out all its pages to disk, so that any changes made within the GC's local memory are actually recorded before it switches to the mark phase of the next garbage collection cycle.

### 3.3. The Disk Controller

When a page is written to disk by either processor, the DC maintains memory consistency by reading the page from the disk and merging the colors before writing it back to the disk. This could be done using a fast buffer; as an alternative, we could use a disk with two heads, the write head being positioned some distance after the read head. In this case, the DC would read the data as it passed under the read head, merge it with the new data being written out, and write it back to disk as the sector passed under the write head, without any overhead being caused by this operation in comparison with a normal write. Since the merge requires very little processing, this implementation is feasible.

The color of the *merged cell* depends on the color of the corresponding cell on disk (the *old cell*) as well as on the color of the *new cell* being written out. It also depends on which processor (the LP or the GC) is doing the writing, and which phase the GC is in (since that determines the meaning of the colors). The DC needs read-only access to the GC phase flag, which as discussed earlier causes negligible overhead. It should also be noted that all combinations of colors are not encountered since they are constrained by the very nature of the tasks of the processors. This is explained further in the following sections.

#### 3.3.1. Mark Phase

When the GC is in the mark phase, only the following color changes can occur. The LP can color cells black if they were originally green (during the CONS operation), and yellow if they were originally white (during the REPLACE operations). The GC can only blacken cells that were originally white or yellow.

##### 3.3.1.1. Page received From LP

The LP can color only green cells black; thus, if a black cell is received from the LP, it must either have been black already (no merging needed), or alternatively it could have been a green cell on the free

list which has now been allocated, in which case it should be colored black. Thus when a black cell is written back, the merged cell is black.

A white cell received from the LP must have been white when the LP read it (since the LP never colors anything white); the old cell on disk can only be white (untouched) or black (having been colored black in the meantime by the GC). In either case, the color of the old cell is taken as the color of the merged cell.

If a green cell is received from the LP, the corresponding cell on the disk must also be green, and no merging is needed.

If a yellow cell is received from the LP, there are several possibilities. If the corresponding cell on the disk (the old cell) is white, it must have been white when the LP read the page and have been colored yellow by the LP; in this case, the merged cell should be yellow. If the old cell is black, it could have been either white or yellow when the LP read the page, having been blackened by the GC in the meantime; in this case, the merged cell should be black. The only other possibility is that the cell was originally yellow and still is; the merged cell in this case is obviously yellow.

This can be summed up in the following table, where blank entries correspond to impossible combinations of colors:

| Color of merged cell | | | | |
|---|---|---|---|---|
| Color of new cell in LP's page: | Black | White | Green | Yellow |
| Color of old cell on disk: | | | | |
| Black | Black | Black | | Black |
| White | | White | | Yellow |
| Green | Black | | Green | |
| Yellow | | | | Yellow |

The DC, finally, needs to implement only the following rules, which can easily be done by a logical circuit:

1. Black merged over any other color results in black.
2. Any color merged over black results in black.
3. Yellow merged over white results in yellow.

The other function performed by the DC is the detection of yellow cells being written out by the LP while the GC is scanning the LP's memory, as explained earlier.

### 3.3.1.2. Page Received From GC

In the mark phase, the GC can only color cells black that were originally white or yellow. Other differences found in the colors are therefore due to the LP. If the cell received from the GC is yellow, the corresponding cell on the disk must be yellow too, and no merging is needed in this case.

If a black cell is received from the GC, the cell could originally have been black, white (and since then traced by the GC, possibly yellowed meanwhile by the LP) or yellow (and since traced by the GC). In either case, the merged cell should be black.

A white cell received from the GC may originally have been white, in which case it stays white, or yellow, in which case the LP has colored it yellow after the GC read the page. In the latter case, the merged cell should be yellow.

If a green cell is received from the GC, it must originally have been green. The corresponding cell in memory at this time, therefore, would still be green unless the LP has in the meantime requested this cell (via a CONS) and colored it black. In the former case, the merged cell stays green; in the latter case, the merged cell is black.

This can be summed up in the following table:

| Color of merged cell | | | | |
|---|---|---|---|---|
| Color of new cell in GC's page: | Black | White | Green | Yellow |
| Color of old cell on disk: | | | | |
| Black | Black | | Black | |
| White | Black | White | | |
| Green | | | Green | |
| Yellow | Black | Yellow | | Yellow |

This can be implemented by the DC by a logical circuit realizing the following rules:

1. Black merged over any other color results in black.
2. Any color merged over black results in black.

3. White merged over yellow results in yellow.

### 3.3.2. Reclaim Phase

In a similar manner, the merging rules for the reclaim phase of the GC can be formulated. The tables describing these rules are:

#### 3.3.2.1. Page Received From LP

| Color of merged cell | | | | |
|---|---|---|---|---|
| Color of new cell in LP's page: | Black | White | Green | Yellow |
| Color of old cell on disk: | | | | |
| Black | Black | | | |
| White | White | White | | White |
| Green | Black | Green | Green | |
| Yellow | | | | |

#### 3.3.2.2. Page Received From GC

Recall that during the reclaim phase, once the GC has started coloring black cells white and white cells green in a certain page, it must complete the operation on that page before returning it to the disk.

| Color of merged cell | | | | |
|---|---|---|---|---|
| Color of new cell in GC's page: | Black | White | Green | Yellow |
| Color of old cell on disk: | | | | |
| Black | | White | White | |
| White | | | Green | |
| Green | | | Green | |
| Yellow | | | | |

### 3.3.3. Merging revisited

We have presented rules for merging the colors of the new cells being written back to disk with the colors of the corresponding cells already on disk. All these rules can collectively be implemented as a logical operation on the color of the cell being received, the color of the corresponding cell on disk, the

processor writing out the page, and the phase of the GC.

We also need to consider the merging of the data in the cells, i.e., the pointers in the CAR and CDR fields of the cells. It is easy to figure out which of the cells, the old cell on the disk or the new one being written out, contains the correct data to be placed in the merged cell. The only time the GC changes the data in a cell is when it colors a white cell green and adds it to the free list during the reclaim phase, thus making the green cell which was previously on the end of the free list point to this one. When the GC writes this green cell back to disk, it finds either a white or a green cell there (depending on when this cell was reclaimed). In either case, the merged cell should get its data from the new cell. In all other cases, the old cell will either have the same data as the new cell, or would have been modified by the LP after the page was read by the GC. The merged cell, then, would get the data from the old cell.

When a page is written out by the LP, a similar argument holds. When it writes out a white cell over a green cell in the GC's reclaim phase, the old cell has the correct data. In other cases, the new cell will have the correct data to be put into the merged cell.

## 4. Extensions and modifications

Since the disk is a shared limited-bandwidth resource, excessive page faults in the GC would eventually reduce the performance of the LP. It is desirable, therefore, to design the GC algorithms with a view towards minimizing the page fault rate as far as possible. Further, we would like the GC to be, on the average, "faster" in some sense than the LP, so that garbage is reclaimed faster than it is generated. Reducing the GC's page fault rate helps in this respect; in addition, unnecessary rescanning of memory during the GC's mark phase should be eliminated if possible. In this section, we explore some extensions and modifications to the basic algorithms presented above in order to improve further the performance of the system.

### 4.1. Eliminating unnecessary rescans

In the mark phase, the GC scans the memory linearly looking for yellow nodes which need to be traced further. When the LP replaces a pointer in a cell X to point to another cell Z, it colors Z yellow and

requests a rescan by setting the rescan flag. However, if Z lies on a page that has not yet been scanned by the GC, it will eventually be taken care of anyway when the GC gets to that page, and hence the rescan request flag need only be set if Z lies on a page that the GC has already scanned.

Another alternative is to maintain a separate table for the addresses of the cells which need further examination. In this scheme, a cell is treated as being yellow if it is colored yellow or if it appears in this *yellow table*. The yellow table can be a fixed size high speed memory, preferably a content-addressable (associative) memory. The GC would examine this first to look for yellow cells which need to be traced before scanning the memory linearly. If the table fills up, the original scheme (mark the cell yellow and set the rescan flag) is resorted to until the table has vacancies again. Care should be taken that the table manipulation does not cause the LP any overhead. Since the table is shared by the LP and the GC, mutually exclusive access needs to be guaranteed, and this scheme may actually be worse as far as the LP is concerned due to this. However, in the yellow table method, the LP doesn't need to access the cell Z itself, as it would if it were to color it yellow. If Z lies on a page which is not currently in primary memory, the coloring process would cause a page fault. If Z is used soon after the REPLACE, this fault does not incur too much overhead, but otherwise the yellow table method is superior in this regard. Dynamic measurements would be useful in judging the comparative utility of these schemes; measurements performed in [Clark77, Clark79] help but are insufficient for the current purpose.

After scanning the disk completely in the mark phase, the GC, as explained earlier, must scan the LP's memory looking for yellow cells that were never written out to disk (which, therefore, the GC did not know about). The GC needs only to get the addresses of these cells, after which it traces the list structures from these cells within its own memory. During this scan, the LP may write out a yellow cell to disk. The DC must detect this and request a disk rescan if this is so. To avoid rescanning the entire disk, the DC can save the address of the yellow cell being written out in a table (the yellow table described above would suffice). A disk rescan is then requested only if this table is full. The table size can be substantially reduced by saving only the page number of the page being written out rather than the entire address of the yellow cell. The GC then has to scan this page to find the yellow cell, but this does not cause any extra

page faults.

### 4.2. Reducing disk contention

As mentioned above, the disk is a shared limited-bandwidth resource, and therefore it is desirable to reduce disk contention by reducing the page faults incurred by the GC. During the mark phase, the GC traces down list structures which could be scattered through memory. In the basic scheme presented above, the GC follows these structures down till the ends of the lists. If a part of the structure being recursively traced lies on a different page, that page will have to be brought in to the GC's primary memory. While the GC has this page, it could as well examine the entire page rather than look only at the yellow cells which are part of the list it was originally tracing. Put another way, all the cells in the pages currently in the GC's primary memory should be examined and traced as far as possible within these pages. If a yellow cell points out to a page not currently in primary memory, it is colored black as usual, but the cell it points to is not brought in; instead, its address is stored in a table, and the GC continues tracing other yellow cells within its primary memory. When no yellow cells remain, the pointers stored in the table are traced further by bringing in their respective pages.

### 4.3. Alternative free list organizations

The unallocated or free cells in the system are chained together in a free list, each pointing to the next. When the LP needs a new cell during the CONS operation, it has to get the next green cell from the head of the free list. This cell, however, may lie on a different page from the one the LP is currently working on, and on the average this increases the page faults incurred by the LP. In addition, lists built up this way tend to be scattered through memory, which increases the number of page faults both when they are used by the LP as well as when they are traced by the GC. To improve the locality of pointers in the system, we present an alternative organization for the free list.

Instead of maintaining one free list for all the free cells, it is possible to keep a separate free list for each page of memory. The head and tail of a page's free list are kept in the *page header*. New cells requested via CONS are allocated from the head of the free list of the current page, and reclaimed cells are

returned to the tail of the free list of the page they lie in. Since we do not want to lock pages out from simultaneous access by the GC and the LP, only the free list head and tail pointers are kept, not its size. To avoid inconsistency, the free list has a dummy header which is never allocated. The CONS algorithm, then, allocates a cell from the current page unless the head and tail pointers are equal. In this case the free list is empty, and a free cell needs to be allocated from another page. This page could be the next page already in primary memory, or the next page according to sequential number, or, to avoid clustering of full pages during execution, a page chosen at random.

Returning reclaimed cells to the tail of the free list poses no particular difficulty. The merging algorithm, however, must ensure that the free list head and tail pointers within page headers are merged correctly. This is a simple extension to the merging rules presented earlier.

## 5. Conclusion

We have presented a design for a parallel garbage collection scheme in an extended virtual memory environment which almost completely eliminates synchronization overhead. The garbage collection is done by a dedicated processor with a separate primary memory, leaving the list processor to perform its own task. A disk controller maintains memory consistency in situations where the list processor and garbage collector simultaneously modify their local copies of the same page. The main advantages of the proposed scheme is that the list processor does not incur overhead due to garbage collection, either directly (sharing CPU cycles, for example) or indirectly (increased page fault rate due to swapping necessitated by the garbage collector's trace through the list structures in memory).

## References


[Baker78] H. G. Baker, "List Processing in Real Time on a Serial Computer," *Communications of the Association for Computing Machinery 21*, 4 (April 1978), 280-294.

[Clark77] D. W. Clark and C. C. Green, "An Empirical Study of List Structure in Lisp," *Communications of the Association for Computing Machinery 20*, 2 (Feb 1977), 78-87.

[Clark79] D. W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *Institute of Electrical and Electronics Engineers transactions on Software Engineering SE-5*, 1 (Jan 1979), 51-59.

[Cohen81] J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys 13*, 3 (Sept 1981), 341-367.

[Deutsch76] L. P. Deutsch and D. G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," *Communications of the Association for Computing Machinery 19*, 9 (Sept 1976), 522-526.

[Dijkstra78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten and E. F. M. Steffens, "On-The-Fly Garbage Collection: An Exercise in Co-operation," *Communications of the Association for Computing Machinery 21*, 11 (Nov 1978), 966-975.

[Hibino80] Y. Hibino, "A Practical Parallel Garbage Collection Algorithm and its Implementation," *Proceedings of the Seventh Annual Symposium on Computer Architecture*, May 1980, 113-120.

[Knuth73] D. E. Knuth, *The art of computer programming, vol. I: Fundamental algorithms*, Addison-Wesley, Reading, MA, 1973.

[Lamport76] L. Lamport, "Garbage Collection with Multiple Processes: An Exercise in Parallelism," *Proceedings of the Institute of Electrical and Electronics Engineers International Conference on Parallel Processing*, 1976, 50-54.

[Lieberman83] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the Association for Computing Machinery 26*, 6 (June 1983), 419-429.

[Newman82] I. A. Newman and M. C. Woodward, "Alternative Approaches to Multiprocessor Garbage Collection," *Proceedings of the Institute of Electrical and Electronics Engineers International Conference on Parallel Processing*, Aug 1982, 205-210.

[Steele75] G. L. Steele, "Multiprocessing Compactifying Garbage Collection," *Communications of the Association for Computing Machinery 18*, 9 (Sept 1975), 495-508.