Copying Garbage Collection

Michael Prinz

***Zusammenfassung*—Copying Garbage Collection ist eine einfache und doch effiziente Methode zur automatischen Speicherverwaltung. Im Rahmen dieser Ausarbeitung wird die grundlegende Idee hinter Copying Garbage Collection beschrieben, anschließend der Algorithmus von Cheney genauer betrachtet. Danach werden einige Themen wie billige Speicher Allokation, „Multiple-area collection", Effizienz und die Lokalitätsthematik angesprochen bevor die Gruppierungsstrategien das Thema abrunden. Abschließend wird Copying Garbage Collection den anderen Methoden gegenübergestellt und dessen Vorteile/Nachteile erläutert.**

I. Einleitung

Die Methode des Copying Garbage Collection hat seinen Ursprung bereits im Jahr 1963. Die erste Implementierung stammte damals von Marvin Minsky, diese wird auch heute noch teilweise oder zumindest als Grundlage modernerer Algorithmen verwendet. 1969 arbeiteten Robert R. Fenichel und Jerome C. Yochelson an einer überarbeiteten Version, aber erst Cheneys Entdeckung einer iterativen Umsetzung des Algorithmus macht Copying Garbage Collection 1970 bekannt. Sie bildet auch die Grundlage für weitere Garbage Collectoren wie der inkrementellen oder generierenden.
Die Seminarsarbeit ist wie folgt gegliedert:
In Abschnitt 2 wird die grundlegende Idee hinter Copying Garbage Collection beschrieben, Abschnitt 3 behandelt den Algorithmus von Cheney und im 4. Abschnitt wird die Allokation diskutiert. Abschnitt 5 bringt die „Multiple-area collection" näher während im 6. Abschnitt die Effizienz betrachtet wird. Kapitel 7 befasst sich mit der Lokalität und Kapitel 8 mit den Gruppierungs-Strategien. Den Abschluss bildet Kapitel 9 mit dem Vergleich von Copying Garbage Collection zu anderen Garbage Collection Methoden.

II. Die Idee hinter Copying GC

Copying Garbage Collection (CGC) teilt den Heap in zwei Semi-Spaces, einer davon enthält die aktuellen und der andere die überflüssigen Daten. Der Vorgang startet mit dem Tauschen der beiden Semi-Spaces. Anschließend durchläuft der Collector die aktive Datenstruktur im alten Semi-Space, dem so genannten Fromspace, und kopierte jede „lebende" Zelle in den neuen Semi-Space, auch Tospace genannt. Nachdem alle aktiven Zellen abgetastet sind, existiert nun eine genaue Kopie im Tospace und das Programm wird neu gestartet.
Da inaktive Zellen einfach ignoriert werden im Fromspace, werden diese copying collectors oft auch „Live Nodes Collectors" oder „Scavengers" genannt – sie picken interessante Objekte aus dem Müll und tragen sie weg.
Ein netter Nebeneffekt des CGC ist die nach dem Kopiervorgang entstandene Struktur im Tospace. Waren vorher noch alle Zellen lose im Fromspace, befinden sie sich nun in geordneter Folge im Tospace, womit keine Fragmentierung mehr nötig ist.
Beim Kopieren muss einzig und allein darauf geachtet werden, ob im Tospace noch genügend Platz ist für das anstehende Objekt. Dies geschieht einfach durch einen Pointer Vergleich, wie folgendes Codestück zeigt:

```
init() =
        Tospace = Heap_Bottom
        space_size = Heap_Size / 2
        top_of_space = Tospace + space_size
        Fromspace = top_of_space + 1
        free = Tospace

New(n) =
        if free + n > top_of_space
                flip()
        if free + n > top_of_space
                abort "Memory exhausted"
        newcell = free + n
        free  = free + n
        return newcell
```

Der Kopiervorgang selbst wird durch ein flip eingeleitet, dabei werden Tospace, Fromspace und top_of_space neu gesetzt. Danach wird jede aktive Zelle mittels eines einfachen rekursiven Algorithmus vom Fromspace in den Tospace kopiert. Dabei ist zu beachten dass die Topologie der geteilten Struktur erhalten bleibt, d.h. dass keine Objekte mehrmals in den Tospace kopiert werden. Dies lässt sich mit Hilfe von „Forwarding Adressen" umsetzen. Beim Kopieren wird eine solche Adresse im Fromspace zurückgelassen, welche dann auf das kopierte Objekt im Tospace zeigt. Immer wenn eine Zelle im Fromspace markiert ist, wird zuerst überprüft ob diese eine Forwarding Adresse zurückgibt, ist das nicht der Fall wird Speicher im Tospace allokiert. Folgendes

Codefragment zeigt den Kopieralgorithmus nach Fenichel-Yochelson:

```
Copy(P) =
        If atomic(p) or P == nil
                Return P

        If not forwarded (P)
                N = size(P)
                P' = free
                Free = free + n
                Temp = P[0]
                Forwarding_address(P) = P'
                P'[0] = copy(temp)
                For I = 1 to n-1
                        P'[i] = copy(P[i])
        Return forwarding_address(P)
```

Abbildung 1 zeigt sehr deutlich wie Tospace und Fromspace vor und nach dem Kopieren aussehen.

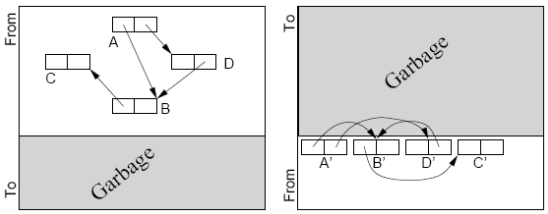

Abbildung 1: Vor und nach dem Kopieren

III. Cheneys Copying Collector

Der bisher gezeigte Algorithmus von Fenichel und Yochelson durchläuft den Graphen rekursiv. Das hat eine katastrophale Laufzeit zur Folge. Des Weiteren benötigt der Rekursionsstack wertvollen Speicher und Rekursion riskiert einen Stack-Overflow. Cheney hingegen beschreibt einen Algorithmus, der iterativ arbeitet und deshalb nur konstanten Laufzeit-Stack benötigt. Im Gegensatz zu seinen Vorgängern speichert Cheney die Objekte nicht in einen Stack sondern in einer Queue. Die Zeiger scan und free zeigen jeweils an ein Ende der Queue. Anstatt zusätzlichen Speicher zu brauchen, legt Cheney die Queue in den Tospace.
Seine Version ist die effizienteste Implementierung von Copying GC.

A. *„The tricolour abstraction“*

Cheney benutzt drei verschiedene Farben, welche die Objekte annehmen können:
Schwarz: zeigt an, dass der Knoten und seine direkten Nachfolger bereits besucht wurden und der Garbage Collector diese somit abhackt. Schwarze Objekte befinden sich links von dem scan-Zeiger.
Grau: Knoten, die bereits besucht aber noch nicht gescannt wurden, werden grau markiert.
Weiß: Knoten dieser Farbe wurden nicht besucht und werden am Ende des Kopiervorgangs zu „Müll“. Danach können sie einfach überschrieben werden.

Ein Garbage Collection Durchlauf endet dann, wenn alle erreichbaren Knoten gescannt wurden, also schwarz gefärbt sind, und keine grauen, ungescannten Knoten übrig bleiben. Alle weißen Knoten werden am Ende freigegeben und können ab sofort jederzeit überschrieben werden.

B. *Der Algorithmus*

Der Algorithmus beginnt mit dem Tauschen der Rollen von Fromspace und Tospace und dem Initialisieren der beiden Zeiger scan und free, welche ab sofort in Tospace zeigen. Free soll ab sofort hinter das letzte kopierte Objekt zeigen und das Ende der Queue repräsentieren. Scan zeigt auf das nächste zu scannende Objekt und stellt den Anfang der Queue dar.
Nun werden die Wurzelknoten des Graphen in den Tospace kopiert. Bei jeder Iteration der Kopierschleife wird die nächste graue Zelle (auf die scan zeigt) auf Pointer, die auf Objekte im Fromspace zeigen, die noch nicht kopiert wurden, untersucht. Wird eine solche Zelle gefunden, wird sie an die Stelle im Tospace kopiert, wo free hinzeigt und eine Forwarding Adresse zurückgelassen. Diese Adresse wird normalerweise ins erste Feld des betreffenden Objekts im Fromspace gespeichert. Die Tospace Nachfolger-Zeiger müssen ebenfalls umgelenkt werden, damit sie auf die grauen Kopien zeigen anstatt auf die Fromspace-Objekte. Anschließend werden free und scan jeweils um die Länge des kopierten Knotens bzw. des gescannten Objekts nach rechts weiterbewegt. Das gescannte Objekt ist nun schwarz und muss deshalb nicht mehr beachtet werden. Der Algorithmus ist dann beendet, wenn keine grauen Zellen mehr übrig sind, d.h. keine weiteren Knoten mehr beachtet werden müssen und somit scan und free an derselben Position stehen.
Voraussetzung bei diesem Algorithmus ist die Annahme, dass alle Objekte Header besitzen und dass die Komponenten der Objekte über diese Header erreichbar sind. Weiters wird angenommen, dass alle Heap Zeiger auf den Head eines Objekts deuten, d.h. interne Zeiger sind verboten.
Der Algorithmus kann auch durch folgende Codezeilen beschrieben werden:

```
flip() =  Fromespace,Tospace=Tospace,Fromspace
          top_of_space = Tospace + space_size
          scan = free = Tospace
          for R in Roots
                  R = copy(R)
          while scan < free
                  for p in Children (scan)
```

```
                    *P = copy(*P)
        scan = scan + size(scan)
copy(P) =
        if forwarded (P)
                return forwarding_address(P)
        else
                addr = free
                move (P,free)
                free = free + size(P)
                forwarding_address(P) = addr
                return addr
```

C. *Ein Beispiel*

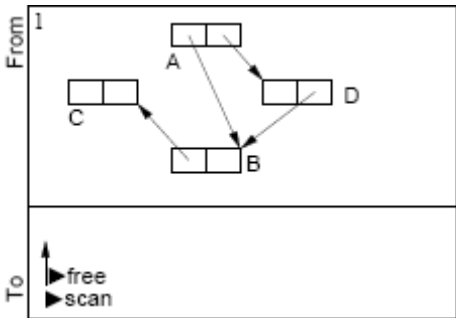

Abbildung 2: Ausgangssituation nach dem flip

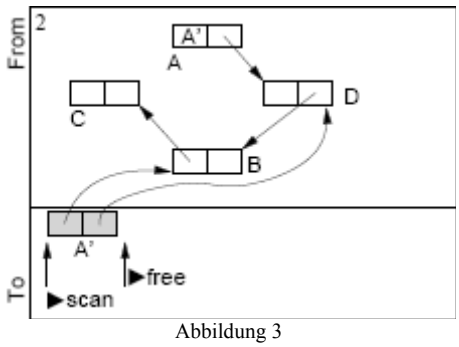

Abbildung 3

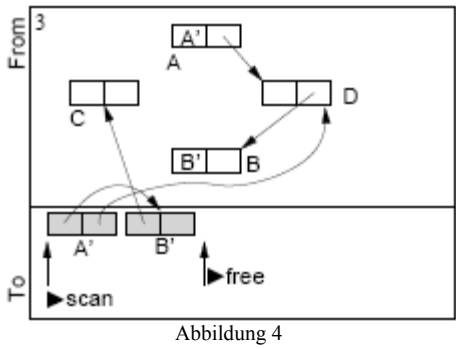

Abbildung 4

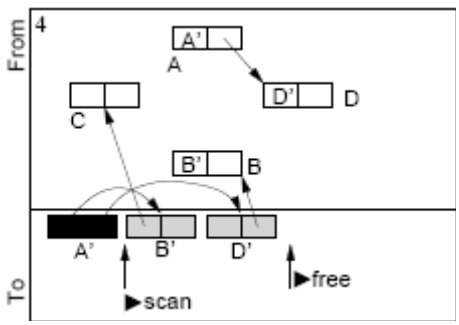

Abbildung 5

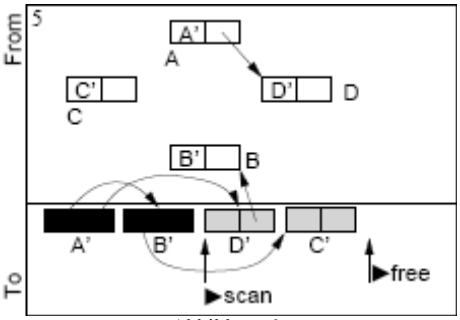

Abbildung 6

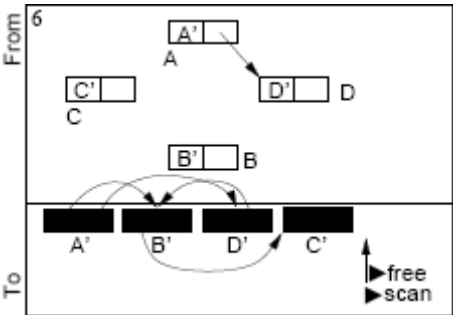

Abbildung 7

Abbildung 2 zeigt die Ausgangssituation nach dem Tauschen von Fromspace und Tospace. Die beiden Zeiger free und scan deuten in den Tospace.
Abbildung 3: Die Wurzel der Struktur, A, wird in Tospace an die Stelle, auf die free zeigt, kopiert. Die Zeiger der Kopie, A', weisen aber noch immer auf die Objekte im Fromspace. Eine Forwarding Adresse wird in das erste Feld von A geschrieben, wobei die Referenz zu B verloren geht. Da jedoch A' bereits auf B zeigt ist das nicht mehr von Belang. Aus dieser Abbildung geht nicht heraus, dass auch der Root-Zeiger auf A' umgelenkt wurde.
Abbildung 4: Der Knoten B' wird hinzugefügt, die Referenz von B auf C gelöscht, dafür eine von B' auf C hinzugefügt. Im ersten Feld von B wird die Forwarding Adresse zurückgelassen und damit auch der Zeiger von A' auf B' verwiesen. Schließlich wandert der free Zeiger um ein Objekt nach rechts.
Abbildung 5: Scan kann noch nicht weiter, da A' in den Fromspace auf D zeigt. Deshalb wird dieses Objekt als D' zuerst in den Tospace kopiert. In D bleibt wiederum nur die Forwarding Adresse bestehen. Diese leitet A' auf D' um. Schließlich kann der scan Zeiger A' schwarz färben und somit eine Position weiterrücken.
Abbildung 6: Wie im Bild zuvor muss zuerst das Objekt C kopiert werden damit scan durchgeführt werden kann. Dort wird die Forwarding Adresse zurückgelassen. Anschließend wird B' gescannt.
Abbildung 7: Es sind bereits alle Knoten übertragen, deshalb ändert sich der free Zeiger nicht mehr. Scan biegt den Zeiger von D' zu B auf B' um, und erreicht schließlich die Position von free.
Nach dem letzten Schritt erfolgt die Vertauschung von Fromspace und Tospace.

IV. Allokation

Die CPU-Kosten von stop-and-go Garbage Collection sind generell niedrig, da nur aktive Knoten besucht werden und das Kopieren von kleinen Objekten billig ist. Einzig und allein was noch gemacht werden muss ist das Überprüfen des Speichers im Semi-Space, inkrementieren des free Zeigers und die Rückgabe der Adresse von der neuen Zelle. Heap Allokation ist somit nicht mehr teurer als Stack Allokation.

Die Allokation von neuem Speicher zwischen zwei Garbage Collection Zyklen erfolgt linear, hierzu kann der Zeiger free benutzt werden (der ja, nach Vertauschen der Rollen, jetzt im Fromspace liegt). Neuer Speicher wird dann allokiert, indem der aktuelle free als Anfang des Speicherblockes zurückgegeben und dann hinter das Ende des Blockes verschoben wird. (diese Methode ist auf den Algorithmus von Cheney bezogen)

V. Multiple-area collection

GCG kopiert, wie bereits erwähnt, nur die „lebenden“ Daten von einem Semi-Space in den anderen. Die CPU-Kosten dieser Methode hängen je nachdem von der Größe des Objektes ab, für kleine Objekte können die Kosten teilweise gleich sein wie wenn es mit einem Bitmap gekennzeichnet würde. Generell gilt, wenn kleine Objekte nur eine kurze Lebensdauer haben, d.h. wenn sie entstehen und verenden zwischen zwei folgenden Garbage Collections, können sie ohne Kosten entfernt werden. Im Gegensatz dazu sollte das Kopieren großer Objekte vermieden werden. Wird beim Start oder kurz danach ein Objekt geladen, das bis zum Ende nicht mehr entfernt wird, so ist es nicht zielführend bei jeder Garbage Collection dieses Objekt von einem Semi-Space in den anderen zu kopieren. Für Spezialfälle dieser Art sind eigene Bereiche vorgesehen.

A. *Statische Bereiche*

Der Sammelaufwand kann reduziert werden, wenn große und vor allem langlebige Objekte speziell behandelt werden. Dafür wird der Heap in einige separate verwaltete Regionen unterteilt. Objekte, von denen bekannt ist, dass sie sich länger im System aufhalten werden, werden in einem so genannten statischen Bereich allokiert. Auch wenn diese statischen Daten Zeiger auf Objekte im Heap aber außerhalb des statischen Bereiches beinhalten, dürfen sie nicht verschoben werden.

B. *Bereiche für große Objekte*

Ähnlich wie oben werden große Objekte in einen eigenen Bereich (large object area) transferiert. Eine Möglichkeit der Verwaltung wäre, diese Objekte in einen kleinen Header und einen Datenteil zu separieren. Der Header würde in dem Teil des Heaps bleiben, das von dem Semi-Space Copying Collector verwaltet wird, während der Datenkörper in der „large object area“ aufbewahrt würde. Dieser große Bereich wird normalerweise durch einen unbewegten Collector verwaltet (z.B. Mark and Sweep) um die Kosten des Kopierens einzusparen, wobei es manchmal nötig ist diesen Bereich zu komprimieren um die Fragmentierung zu reduzieren.

C. „*Compacting Garbage Collection*“

Das Unterteilen des Heaps in mehrere separate Bereiche bringt mehrere Vorteile mit sich. Einer davon ist das Eliminieren von Fragmentierung durch das Komprimieren des Heaps. Der Nachteil daran liegt in den Kosten für den zweiten Semi-Space. Eine Möglichkeit die Vorteile des Komprimierens zu nutzen, ohne die Semi-Space Kosten des totalen Kopierens zu zahlen und dem Zeitproblem von Mark & Compact zu entgehen, wäre Teile des Heaps inkrementell zu zusammenzufügen.

Lang und Dupont teilten 1987 den Heap in n + 1 gleich große Segmente. Bei jedem Garbage Collection Zyklus werden zwei dieser Segmente als ein Paar Semi-Spaces behandelt und von einem Copying Collector verwaltet, während der Rest des Heaps mittels Mark & Sweep bearbeitet wird.

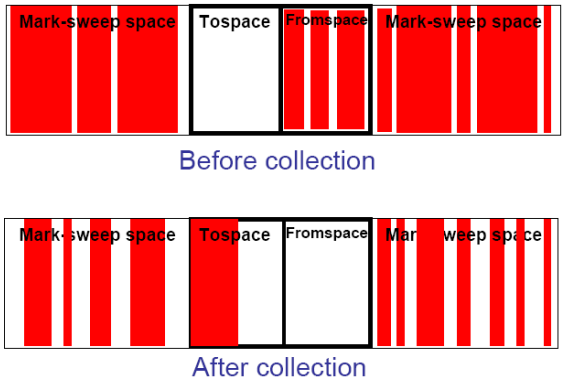

Abbildung 8: Lang und Dupont

Abbildung 8 zeigt wie der Heap vor und nach dem Bearbeiten aussieht. Am einfachsten ist es, wenn die beiden Semi-Spaces direkt nebeneinander liegen, etwa i und i + 1. Das Segment i ist zu Beginn leer und wird zum Tospace, i + 1 zum Fromspace. Wenn der Kollektor beginnt den Graphen zu traversieren, werden besuchte Zellen markiert (mittels Mark-Bit), es sei denn diese Zellen liegen im Fromspace, denn diese Objekte werden in den Tospace kopiert und lassen wie gewohnt nur eine Forwarding Adresse zurück. Referenzen zu diesen Objekten, sei es von Knoten aus dem Tospace oder dem Mark-Sweep-Space (alle außer i und i + 1), müssen umgelenkt werden, damit sie zukünftig auf die Tospace Kopien zeigen. Am Ende des Durchlaufes befinden sich alle

aktiven Zellen aus dem Segment i + 1 zusammengepackt in Segment i. Für den nächsten Durchlauf kann dann i + 1 als Tospace verwendet werden und die Daten von (i + 2) modulo n aufnehmen. Das Fragment des Tospace das unbenutzt bleibt kann der free-Liste hinzugefügt werden.

Wie in der Abbildung 8 zu sehen ist, besteht der Heap aus zwei Datenstrukturen, dem Mark & Sweep- und dem Copying-Teil. Der Kollektor hat nun die Wahl, welchen Weg er einschlägt. Entweder wird dem Marking Stack ein Objekt entnommen oder aber der scan Zeiger weiterbewegt. Lang und Dupont meinten man solle den Mark-Sweep immer dem Copying Kollektor vorziehen um das Wachstum des Stacks zu limitieren.

VI. Garbage Collector Effizienz

Appel argumentierte 1987, dass CGC durch das erweitern der Heapgrösse willkürlich billig gemacht werden kann. Um ein Objekt zu sichern muss der Kollektor dieses erst in den Tospace kopieren und danach seinen Ursprung „scavangen“. Beträgt die Anzahl der erreichbaren Knoten R, dann benötigt die Kopiermethode nach Cheney cR Operationen. Die Konstante c ist abhängig vom Verarbeiten einer Zelle und der durchschnittlichen Anzahl an Zeigern die sich darin befinden. Die Menge an Zellen, die zwischen den Garbage Collections allokiert werden, beträgt M / s – R, wobei M die Größe jedes Semi-Spaces und s die durchschnittliche Größe jeder Zelle ist. Das entspricht auch der Anzahl der zu entfernenden Zellen bei jeder Kollektion, falls die Menge von erreichbaren Zellen konstant bleibt. Die CPU-Kosten pro Garbage Collection einer Zelle werden folgendermaßen berechnet:

$$g = (cR / ((M/s) – R) = (c / ((M/sR) – 1))$$

In der Theorie kann g beliebig klein werden indem M vergrößert wird. Appel meinte, mit genügend Speicher sei es billiger eine Zelle zu kopieren als sie explizit aufzulösen, auch wenn die Kosten dafür nur eine Instruktion betragen.

Abbildung 9 und 10 illustrieren den Effekt eines vergrößerten Heaps. Wird ein Programm zwei Mal ausgeführt, einmal mit einem Semi-Space von 350MB, anschließend mit einem Semi-Space von 700MB. Um die Annahme zu vereinfachen wird vorausgesetzt, dass die Menge an aktiven Speichern annähernd konstant ist, in diesem Fall 100MB. Insgesamt werden vom Programm 1800MB allokiert. Um diese Datenmenge mit dem kleineren Semi-Space abzuarbeiten muss der Garbage Collector sechsmal agieren (6 x 100MB = 600MB). Der Durchgang mit dem größeren Semi-Space hingegen muss nur zwei Mal angestoßen werden (2 x 100MB = 200MB)

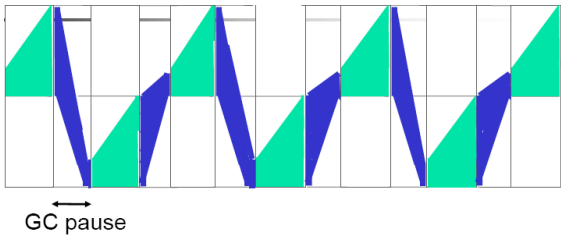

Abbildung 9: Semi-Space mit 350MB

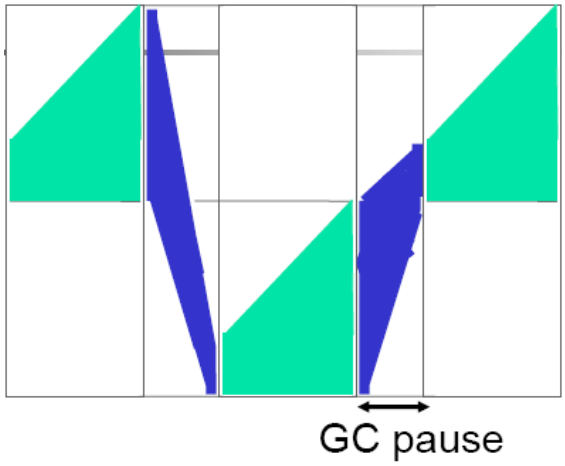

Abbildung 10: Semi-Space mit 700MB

VII. Lokalität

In der Realität spielt der Virtuelle Speicher und dessen Verhalten solange eine wichtige Rolle bezüglich der Gesamtleistung des Systems, bis der gesamte Heap im Hauptspeicher gelagert werden kann. Des Weiteren spielen auch Cache Verfehlungen eine wichtige, wenn auch weniger signifikante Rolle. Wird der Heap zu groß, können die Vorteile der reduzierten Anzahl an gebrauchten Collections nicht mehr mit der gestiegenen Anzahl an Seitenaufrufen (paging) aufgewogen werden. Seitdem die Kosten eines Seitenfehlers (page fault) eine Unmenge an Zyklen betragen, wird zusätzliche CPU Leistung dafür aufgewendet Seitenfehler zu vermeiden.

Das Speicher Management System, bestehend aus Garbage Collector und Allokator, behandelt jede Seite im Tospace in jedem Collection Durchgang. Wird der Heap vergrößert, steigt auch die Zahl der Seiten, die in jedem Durchgang abgearbeitet werden. Weiters ordnet das Kopieren die Objekte im Heap neu an, mit der Konsequenz, dass die Lokalität der Heap Datenstruktur verändert wird.

Zorn verglich 1989 das Paging Verhalten von „generational copying“ und „generational mark-sweep“ Collectoren, dabei lieferte letztere Variante ein spürbar besseres Speicherverhalten. Wilson argumentierte, der Grund dieser Ungleichheit läge im regelmäßigen Gebrauch der beiden Semi-Space. Dieses Muster der zyklischen Wiederverwendung besagt, die nächste zu allokierende Seite ist die am

längsten nicht mehr benutzte Seite. Dieses Muster widerspricht aber den Seitenersetzungsstrategien des Virtuellen Speichers, die normalerweise die am wenigsten frequentierten Seiten auslagert, in der Annahme dass sie auch am längsten nicht mehr gebraucht werden. Wenn die Seiten im reellen Speicher nicht mehr ausreichen um die beiden Semi-Space gleichzeitig aufrechtzuerhalten, werden die Tospace Seiten immer vorher ausgelagert bevor sie allokiert werden. Am effektivsten ist es jedoch von vornherein zu kontrollieren, ob beide Semi-Spaces in den Hauptspeicher passen. Ist der Heap zu groß dafür, kann er in mehrere kleine Segmente unterteilt werden, die unabhängig von einander bearbeitet werden.

A. Unterstützung des Betriebssystems

Paging kann auch reduziert werden durch Mithilfe des Betriebssystems. Nach einem Collection Durchgang sind alle Daten auf Fromspace-Seiten und alle Tospace-Seiten mit Adressen, die rechts von free liegen, Müll. Wenn eine neue ungeladene Tospace-Seite allokiert wird, werden die Daten auf der ausgelagerten Seite in den Hauptspeicher geladen obwohl es nur Müll enthält. Das Laden dieser kann das Aussetzen einer Fromspace Seite zur Folge haben. Die Seite wird wahrscheinlich als „schmutzig“ markiert, entweder weil der Mutator seinen Inhalt geändert oder der Kollektor Forwarding Adressen daraufgeschrieben hat. In beiden Fällen kopiert der Virtuelle Speicher die Seiteninhalte auf die „Swap disk“. Egal ob aus der Sicht des Mutators oder des Kollektors ist dieser Datenverkehr unnötig.

Eine bessere Lösung wäre die Zusammenarbeit des Betriebssystems mit dem Dynamischen-Speicher-Manager damit der Seitenrahmen im reellen Speicher zugehörig zu der Fromspace Seite einfach auf die Tospace Seite abgebildet wird ohne dabei Disk Operationen durchzuführen.

Es können einige Garbage-Collected-Prozesse mit großen Heaps gleichzeitig laufen. Jeder dieser Prozesse will einen maximalen Fortschritt erzielen und erweitert deshalb seinen Heap so weit wie möglich ohne dabei mit anderen zu kollidieren. Dies bringt einen Streit um den reellen Speicher mit sich. Deshalb haben Alonso und Appel 1990 vorgeschlagen die Heap Größen zentral zu verwalten. Bei jeder Collection soll jeder Prozess eine zentrale Anlaufstelle befragen, ob er seinen Heap vergrößern oder verkleinern soll. Die Kriterien für diese Entscheidung der Anlaufstelle sind einerseits die Zeit, die jeder Prozess für nützliche Arbeit und Garbage Collecting aufgewendet hat, andererseits die minimalen Speicheranforderungen jedes Prozesses.

VIII. Umgruppierungsstrategien

Es ist wünschenswert, dass die Beziehungen zwischen den Daten auch in ihrer Struktur im Heap wiedergegeben werden. Je enger Daten miteinander verbunden sind, desto enger sollten sie auch im Heap platziert werden. Diese Beziehungen zwischen den Datensätzen des Mutators können strukturell (die Knoten sind Teil einer gemeinsamen Struktur) oder temporal (der Mutator greift auf die Objekte zu ähnlichen Zeiten zu) sein. Werden verwandte Daten auf denselben Seiten platziert reduziert sich die Paging Häufigkeit, denn wenn ein Objekt in den Hauptspeicher geladen wird werden auch seine Nachbarn geladen, und diese werden mit hoher Wahrscheinlichkeit bald gebraucht.

Untersuchungen von Hayes 1991 zeigen, dass Objekte normalerweise in Clustern erzeugt und zerstört werden. Über 60 Prozent der langlebigsten Objekte werden innerhalb von nur einem Kilobyte allokiert, diese Verbindung ist sogar noch stärker wenn junge Objekte beachtet werden. Schließlich sterben diese Objekte in Clustern, das heißt dass das Anfangslayout der Objekte im Heap spätere Zugriffsmuster durch den Benutzer widerspiegelt.

Die Anordnung der Objekte im Heap wird beim Kopieren angepasst. Je nachdem wie der Graph traversiert wird ändert sich die Art und Weise wie die Daten umgruppiert werden. Die einfachsten Traversiermethoden sind die Tiefensuche („depth-first“) und die Breitensuche („breadth-first“). Die Depth-first-Methode besucht zuerst alle Nachfolger eines Knotens bevor er dessen Geschwister besucht. Im Gegensatz dazu durchläuft die Breadth-first-Methode zuerst immer die Geschwister und dann erst die Nachfolger. Der Copying Collector nach Cheney bedient sich des Breadth-First Algorithmus während rekursive Varianten, wie etwa Mark & Sweep oder Fenichel-Yochelson, den Graphen mittels Tiefensuche traversieren.

Untersuchungen an der Verwendung des Garbage Collectors zur Verbesserung der Referenzlokalität lieferten zwei Versionen:

1. *Statische Umgruppierung (static regrouping)*

Diese Methode untersucht die Topologie der Heap-Datenstruktur um struktur-verwandte Objekte besser anzuordnen. Statisch wird es deswegen genannt, da sie Struktur des Graphen zur Collection-Zeit analysiert und weniger beachtet wie der Mutator auf die Daten zugreift.

2. *Dynamische Umgruppierung*

„Dynamic Regrouping“ clustert die Objekte je nach Art des Zugangsmusters zu den Daten durch den Mutator. Das benötigt eine Umgruppierung der Objekte zur Laufzeit durch einen inkrementierenden Copying Collector.

A. *Tiefensuche vs Breitensuche*

Moon fand 1984 heraus, dass Tiefensuche eine bessere Lokalität liefert als Breitensuche. Erklären lässt sich das durch die Platzierung von Eltern und Kindern auf dieselbe Seite, besonders wenn Datenstrukturen eher dazu neigen in die Breite zu gehen als an Tiefe zu gewinnen.

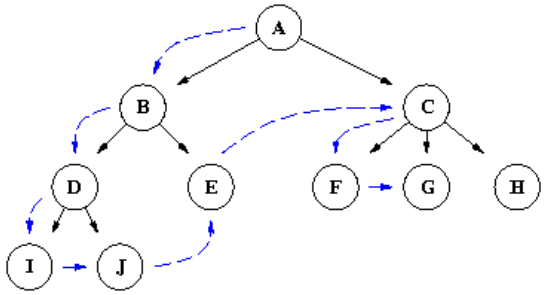

Abbildung 11: Tiefensuche

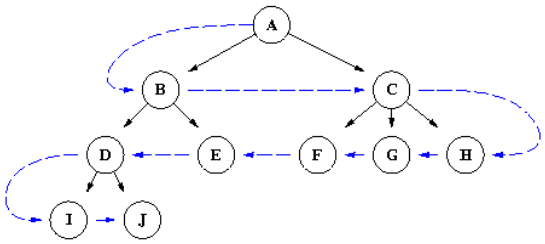

Abbildung 12: Breitensuche

In Abbildung 11 sieht man wie die Tiefensuche dazu tendiert Knoten auf eine Seite mit seinen Nachfolgern zu platzieren. Abbildung 12 hingegen illustriert wie die Breitensuche entfernt verwandte Knoten zusammenfügt. Diese Methode reduziert die Chance beim Laden eines Objektes in den reellen Speicher dabei Objekte mitzuladen, die vielleicht bald gebraucht werden könnten, eine erhöhte Seitenfehler-Wahrscheinlichkeit ist die Konsequenz. Generell wird bei der Breitensuche Level-weise kopiert, das heißt zuerst werden die Wurzeln kopiert, anschließend die Level-2, Level-3 usw. Nachfolger. Die erreichbaren Daten im Tospace sind dann vielmehr verzahnt als logisch gruppiert angeordnet.

Stamos und Blau verglichen die Wirkung verschiedener Gruppierungsmethoden auf Paging. Neben Tiefen- und Breitenanordnung wurden Objekte auch nach Typ, Erstellung, mittels Reference Count und zufällig gruppiert. Die Simulationen zeigten, dass Tiefen- und Breitenordnungsverfahren weniger Seitenfehler lieferten als etwa Zufallsanordnung, aber auch dass Tiefensuche nur minimal besser ist als Breitensuche, außer für sehr kleine reelle Speichergrößen. Beide Anordnungen lieferten hingegen eine schlechtere Lokalität als Optimale- oder Erstellungsanordnung. Größere Seiten lieferten weniger Seitenfehler als kleinere.

Wilson et al. kritisierten die Argumentation von Stamos und Blau, da diese die Topologie von typischen Programmen ignoriert haben. Viele Systemabbildungen enthalten einige sehr breite Wurzelknoten, besitzen dafür aber eine eher flache Struktur. Diese Wurzeln sind normalerweise Hashtabellen von allen Symbolen und Wurzeln. Hashtabelle gruppieren Daten in einer pseudo-zufälligen Anordnung. Für eine gute Leistung wurden sie so gestaltet, dass die Schlüssel in der Tabelle eher verteilt sind und nicht in Gruppen gespeichert sind.

Ein weiterer Punkt den Wilson et al. bemängeln ist das Ignorieren der katastrophalen Gruppierungseffekte auf die Lokalität durch das lineare Traversieren von Hashtabellen. Seitenfehler könnten reduziert werden, wenn Hashtabellen speziell und normale Datenstrukturen auf eine Art depth-first behandelt würden. Wie bereits erwähnt benutzt der Fenichel-Yochelson-Collector die Tiefensuche zum Kopieren, benötigt dafür jedoch einen zusätzlichen Speicher um den Rekursionsstack zu speichern und riskiert daher einen Stack-Overflow. In den folgenden beiden Abschnitten werden zwei Wege vorgestellt, die dieses Problem umgehen können.

B. *„Stackless recursive copying collection“*

Eine Möglichkeit das Stackproblem aus der Welt zu schaffen wäre die Zeiger-Umkehrung nach Deutsch-Schorr-Waite. Das Problem an dieser Methode liegt im zusätzlichen Speicher der benötigt wird für die Flag-Bits und der langsamen Geschwindigkeit, müssen doch in jeder Iteration Bits abgefragt und Zeiger manipuliert werden.

Thomas und Jones beschreiben einen rekursiven CGC für eine „shared environment closure reducer“, der weder extra Speicher benötigt noch interpretierend ist. Die Grundeinheit der Heap Allokation ist ein „frame of closures“ von variabler Länge. Jedes Closure enthält einen Code-Zeiger und einen Umgebungs-Zeiger auf einen Heap-Rahmen.

Zur Collection-Zeit können einige Closures in einem Rahmen aktiv, andere aber Müll sein. Obwohl ein Rahmen selbst gesichert werden muss auch wenn nur wenige seiner Closures aktiv sind, Garbage Closures müssen nicht rekursiv kopiert werden. Würde man dies dennoch machen, würde das zu einem Speichermangel führen, da Müll fälschlicherweise kopiert und somit Speicher permanent unzugänglich gemacht würde. Wenn ein Closure kopiert und entfernt wird, können die aktiven Stellen in dessen Umgebung durch seinen Code-Zeiger bestimmt werden.

Ein Cheney-Collector, der jeden Rahmen nur einmal überprüft, ist unpassend, da ein Rahmen zwischen mehreren Closures aufgeteilt sein kann und jedes von denen benutzt einen verschiedenen Satz an aktiven Objekten. Eine Möglichkeit wäre den Tospace regelmäßig abzutasten bis keine neuen

Rahmen mehr kopiert werden, der Nachteil daran wäre eine auf $O(n^2)$ gestiegene Komplexität. Stattdessen implementieren Thomas and Jones die Collection rekursiv, aber ziehen den Rekursionsstack durch die Fromspace Closures die bereits besucht wurden. Stellt sich nur mehr eine Frage: Wie kann eine Beschreibung eines Satzes von Umgebungsstellen (environment slots) in einem einzigen Closure-Slot gespeichert werden ohne einen interpretierenden Overhead auf den Collector zu setzen?

Thomas and Jones Collector wird spezifisch auf jedes Programm vom Compiler zugeschnitten, Closure Code Zeiger zeigen auf eine Informations-tabelle (nicht auf den Code selbst) Diese Tabelle beinhaltet den Code, der für die Auswertung der Closures nötig ist, und einen Zeiger auf den „Scavenger“, der für diese Code Sequenz verantwortlich ist. Der Code des Scavengers weiß genau welche Stellen in der Umgebung des Closures vom Auswertungscode gebraucht werden.

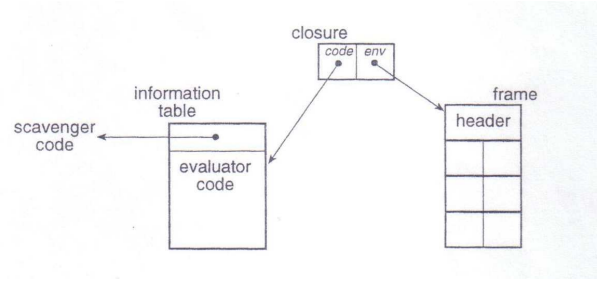

Abbildung 13: Code-Umgebungs-Closures für stackfreies rekursives Kopieren

C. *„Approximately depth-first copying“*

Moon modifizierte 1984 Cheneys Algorithmus derart, sodass er annähernd depth-first wurde. Anstatt von der Zelle auf die scan zeigt zu lesen, wird immer an der zuletzt teilweise gefüllten Seite (page(free)) im Tospace fortgesetzt, dabei werden die Tospace Seiten mehr wie ein Stack als wie eine Queue behandelt. Scan_partial untersucht die Seite am Ende von Tospace solange bis die letzte allokierte Seite vom Tospace komplett gelesen wurde (entweder ist es komplett gefüllt oder es wurden keine weiteren Fromspace Referenzen gefunden). Obwohl scan_partial Breitensuche benutzt, versichert es, dass Objekte so weit es möglich ist auf dieselbe Seite kommen wie die zugehörigen Referenzen. Wird ein Objekt auf eine neue Seite kopiert, startet der Scanner auf dieser Seite. Wenn ein neu kopiertes Objekt die Seitengrenzen berührt, startet der Lesevorgang von neuem an dem Teil des Objekts auf der neuesten Seite in der Absicht es zu füllen.

Wann immer scan_partial mit dem Lesen der letzten Tospace Seite fertig ist, kehrt der Algorithmus zu seiner üblichen Breitensuche zurück und liest die Tospace Objekte ein. Scan_all ist beinahe der Standard-Breitensuche-Scavanger, jedoch stoppt der Scan-Vorgang sobald ein Objekt aus dem Fromspace in den Tospace kopiert wird. Dieses Objekt wird als Kern für scan_partial verwendet. Copy wird beinahe unverändert von Cheneys Algorithmus übernommen, einziger Unterschied liegt darin nach Zeigern zu suchen die bereits verfolgt wurden. Flip wechselt zwischen Scan-Seiten am Ende des Tospace und der Standard-Breitensuche. Folgende Code passage beschreibt diesen Algorithmus:

```
flip() =
        Fromspace, Tospace = Tospace,
Fromspace
        scan, partial, free = Tospace

        for R in Roots
                R = copy (R)

        while scan < free
                scan_partial()
                scan_all

scan_partial () =
        while partial < free
                *partial = copy(*partial)
                partial = max(page(free), partial +
1)

scan_all() =
        oldfree = free
        while oldfree == free
        and scan < partial
                *scan = copy(*scan)
                scan = scan + 1
        if free > oldfree
                partial = max(page(free), partial)

copy(P) =
        if atomic(P)
                return P
        if tospace(P)
                return P
        if forwarded(P)
                return forwarding_address(P)
        else
                addr = free
                move(P,free)
                free = free + size(P)
                forwarding_address(P) = addr
                return addr
```

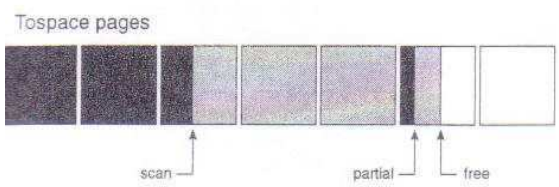

Abbildung 14: Schwarze und grau Seiten können sich abwechseln

Ein Problem dieses Schemas ist dass scan_all Adressen scannen kann die bereits von scan_partial eingelesen wurden. Das ist ein Spezialfall des Problems dem Thomas beikommen wollte. Moon argumentiert dass die Kosten des extra Lesens niedrig sind (etwa 30 Prozent aller Objekte können erneut gelesen werden) verglichen mit den Kosten zur Vermeidung des erneuten Lesens. Findet der Scavenger irgendwelche ungelesenen Referenzen zu Objekte im Fromspace, muss er eine Arbeit erledigen die in jedem Fall angefallen wäre. Wird keine Fromspace Referenz gefunden, so wird kein Verlagerungsmechanismus angestoßen und es tritt auch kein Seitenfehler auf.

Nach Moons Berichten benötigt sein Algorithmus für die Garbage Collection etwa um 6 Prozent länger, er zeigt jedoch nicht wie effektiv sein Algorithmus war bezüglich Reduzierung der Seitenfehler. Courts verzeichnete 1988 eine 15 prozentige Verbesserung durch die Benutzung eines rekursiven „depth-first“ Scavenger.

D. *„Hierarchical decomposition“*

Wilson et al. entfernen das erneute Lesen aus Moons Algorithmus durch eine Modifikation, die den Algorithmus zu einer Art 2-Level Version von Cheney macht. Jede Tospace-Seite greift dabei auf 4 Zeiger zurück, einen großen und einen kleinen free und scan Zeiger. Der Algorithmus liest in regelmäßigen Abständen die erste ungelesene Adresse in der ersten unfertig gelesenen Seite im Tospace. Dabei zeigt der große scan Zeiger auf die Seite, der kleine scan Zeiger weißt auf die aktuelle Stelle in der Seite. Wie im Algorithmus von Moon wird, wenn eine Referenz auf ein nicht kopiertes Fromspace Objekte gefunden wird, dieses Objekt ans Ende des Tospace kopiert und ein Breitensuch-scan auf diese Seite begrenzt durchgeführt. Der Durchlauf endet wenn entweder die Seite voll ist oder sie durchtraversiert wurde.

Sowohl Moons „Annährende Tiefensuche“ als auch die Version von Wilson et al. resultieren in einer hierarchischen Aufspaltung des Graphen. Es werden immer einige Knoten zusammengefasst auf eine Seite, zum Beispiel angefangen mit der Wurzel und seinen zwei Nachfolgern, auf diese Art werden auch alle Sub-Graphen rekursiv durchgelaufen.

Wilson et al behaupten, ihr Algorithmus sei sogar effizienter als die Traversierung mittels Tiefensuche, denn der Zugriff auf einen Knoten lädt sofort auch seine Nachfolger in den Hauptspeicher. Wird ein Knoten vom Mutator besucht, ist es sehr wahrscheinlich dass auch bald Knoten eines Levels darüber oder darunter benötigt werden. Hierarchische Aufspaltung versucht eher mehrere wichtige Knoten zusammenzufügen als strikt nur nach Tiefen- oder Breitensuche vorzugehen. Dieses Vorgehen wird gewählt, da es den typischen Zugriffsmustern der Benutzerprogramme entspricht.

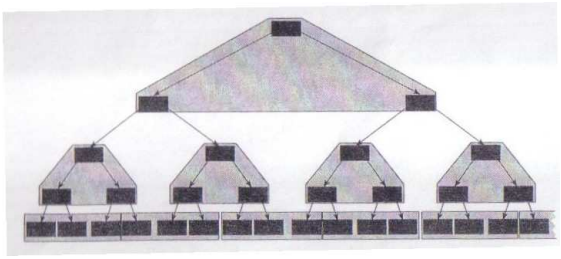

Abbildung 15: Hierarchische Aufteilung

E. *Hash Tabellen*

Wilson et al vermieden das traversieren der Graphen von System Hash Tabellen. Stattdessen modifizierten sie den Compiler damit er eine lineare Liste verwaltet, die aus den globalen Variablen der Hash Tabelle besteht und deren Objekte nach dem Entstehungsdatum geordnet sind. Diese Liste wird nur vom Garbage Collector verwendet und hat keinen Einfluss auf das normale Programm. Ebenso gruppierten sie globale Prozeduren, die von den Variablen angezeigt werden. Beim Traversieren erreicht der Collector nun die globalen Objekte in der Reihenfolge in der sie erzeugt wurden. Frühere Studien zeigten, dass das Kopieren nach der Definitionsanordnung nicht nur eine bessere Lokalität als Zufallsanordnung, wie es bei Hash Tabellen der Fall ist, aufweist, sondern auch Tiefen- und Breitensuche hinter sich lässt. Weiters zeigten die Ergebnisse einen signifikanten Rückgang der Seitenfehler-Häufigkeit, im Besonderen bei Programmen die relativ klein waren.

Ein Grossteil dieser Verbesserung geht auf die bessere Handhabung der Hash Tabellen zurück und durch das Umgruppieren wurde auch eine bessere Lokalität erzielt, wobei die meisten Zeiger auf Objekte innerhalb derselben Seite zeigten.

Eine spätere Studie zeigte dass optimales Gruppieren sehr von der Form und dem Typus der Datenstruktur, die kopiert wurde, abhängig war. Obwohl sich die hierarchische Aufspaltung sehr gut für Baumstrukturen eignete, konnte sie auf andere Strukturen angewandt nicht überzeugen. Deshalb wurde empfohlen, die Traversieranordnung auf die Art des Objektes anzupassen. Funktionen etwa sollten nach ihren Aufrufen, Assoziationslisten in Tiefenanordnung und andere Listen in hierarchischer Aufteilung geordnet werden.

Obwohl immer versucht wird, die Anzahl der Seitenfehler zu minimieren, wäre es doch am effektivsten wenn erst gar keine Fehler beim Kopieren auftreten. Diese Version kann allerdings nur realisiert werden, wenn beide Semi-Spaces im reellen Speicher gehalten werden können. Entweder es ist ein größerer Speicher vorhanden oder die beiden Semi-Spaces müssen kleiner gewählt werden. Bei letzterem verkürzt sich somit auch die Zeit zwischen den Garbage Collection Durchläufen. An diesem Punkt setzt auch die Generational

Garbage Collection an, die aber an dieser Stelle nicht genauer erläutert wird.

IX. Copying Collection – Ja oder Nein?

Copying ist wahrscheinlich die am weitesten verbreitete Garbage Collection Methode, entweder in seiner eigenen Form oder als Grundlage für andere generierende oder inkrementierende Collectoren.

Ein Nachteil von nicht-bewegenden Speicher-Managern ist die Anfälligkeit für Fragmentierung. Obwohl es viele Methoden gibt um das Problem zu verbessern, liegt die einzige Möglichkeit zur Beseitigung des Problems bei der Benutzung eines komprimierenden Collectors wie etwa Mark & Compact oder Copying. Weiters verbessert die Verdichtung der Objekte auch die Lokalität.

Im direkten Vergleich von Marc & Compact und Copying sticht die erste Methode durch zwei Vorteile heraus. Erstens arbeitet sie nur auf einem Adressraum, ein zweiter Semi-Space wird nicht benötigt. Übersteigt beim Copying die Größe der beiden Semi-Space den reellen Speicher, kann es sehr leicht zum „Paging“ und dem damit verbundenen Leistungseinbruch kommen. Wegen der LRU-Strategie (Least Recently Used) der linearen Allokation und der Virtuellen Speichersysteme ist die nächste Seite die allokiert wird diejenige Seite, die am wahrscheinlichsten entfernt werden würde. Jedes Mal wenn eine neue Seite allokiert wird muss sie hereingeladen werden.

Zweitens behält Mark & Compact die Allokationsordnung der Objekte im Heap bei, da einige Anwendungen davon profitieren. Prolog Compiler etwa brauchen diese Anordnung um unbegrenzte Mengen an Speicher in konstanter Zeit anzufordern, dabei wird der Heap als Stack behandelt. Der Nachteil von Marc & Compact Collectoren sind die Kosten von der Verdichtungs-Phase, denn dabei muss der Heap bis zu dreimal durchlaufen werden.

Da das Allokieren in einem komprimierten Heap sehr günstig ist, liefert Copying Garbage Collection sehr gute Ergebnisse bei Systemen mit erhöhtem Allokationsanteil. Aus diesem Grund werden Systeme mit einer hohen Allokationsrate normalerweise von Copying Collectors verwaltet. Für einige Heap-Konfigurationen wiederum eignet sich das Verfahren weniger. Da das Kopieren von Objekten von deren Größe abhängig ist, wäre das einfache Markieren bei allen Objekten (außer den kleinsten) am billigsten. Besteht der Heap zusätzlich noch aus vielen großen und langlebigen Objekten, ist Copying nicht die attraktivste Lösung.

Warum aber sollten alle Objekte im Heap auch nur von einer einzigen Collection-Methode verwaltet werden? Stattdessen kann eine Art Misch-Collector die verschiedenen Objekttypen mittels mehreren Collection-Methoden behandeln. Viele Collectoren verwenden diese Misch-Form indem sie den Heap in mehrere Bereiche unterteilen. „Permanente“ Objekte, von denen man weiß dass sie sich sehr lange im System aufhalten werden, können in einem statischen Bereich gespeichert werden. Obwohl sie auf Zeiger untersucht werden müssen, brauchen sie niemals markiert, ausgelagert oder kopiert werden. Objekte mit einem atomaren Wertebereich, das heißt sie enthalten keine Zeiger, müssen in diesem statischen Bereich nicht gescannt werden. Wenn die Verzögerung, die durch große Objekte verursacht wird, zu kostspielig ist, können diese Objekte einfach in einen separaten Bereich im Heap gespeichert werden (large-object area), wobei der Header im normalen Heapbereich gespeichert werden muss damit das Objekt zugänglich ist. Dieser Bereich für große Objekte sollte von einem Collector wie Marc & Sweep bearbeitet und zwischendurch Komprimierphasen eingeleitet werden.

Ein Punkt der bis dato noch nicht angesprochen wurde ist die Tatsache, dass es sich bei Copying Collector um stop/start-Collector handelt. Darunter versteht man das Anhalten aller laufenden Prozesse bis der Heap wieder vollständig gesäubert ist. Abhängig davon welche Datenmengen eine Collection überstehen, kann der Garbage Collection Vorgang durch interaktive oder Echtzeitprogramme unterbrochen werden. Eine Möglichkeit wäre den Heap schrittweise zu kopieren, das heißt den Benutzer und den Garbage Collector abwechselnd agieren zu lassen. Eine andere Variante wäre sich auf die Bereiche des Heaps zu konzentrieren, die am ehesten Müll enthalten und nur diese Segmente reinigen zu lassen. Dieser Lösungsweg ist vor allem dann angebracht, wenn der Heap aus kurz- und langlebigen Objekten besteht.

Wird Copying Collection entweder als stop/start-Collector oder als Basis für einen generierenden Collector benutzt, bietet sich meist der Algorithmus nach Cheney eher an als die Version von Fenichel-Yochelson.

A. Performanz

Wie bereits erwähnt gibt es Möglichkeiten um das Kopieren einiger Objekte zu vermeiden, und dort wo es unverzichtbar ist kann es effizienter gemacht werden. Wenn große Objekte ihren eigenen virtuellen Speicherseiten zugewiesen werden können sie ohne Kopieren in den Tospace abgebildet werden, ermöglicht wird das durch ein „Re-Mappen“ der Seitentabelle des Betriebssystems. Ebenso kann Paging reduziert werden. Am Ende einer Collection-Phase enthalten alle Fromspace Seiten und alle ungelesenen Tospace Seiten Müll. Jeglicher Aufwand, sei es für das Speichern der Fromspace Inhalte (versetzt mit Forwarding-Adressen) oder für das Laden der Tospace Seiten bevor sie allokiert wurden, ist Verschwendung. Kooperiert der Collector mit dem Virtuellen Speichersystem, kann unnötiger Datenverkehr vermieden werden.

Schließlich ist noch zu sagen, dass das Kopieren mittels Breitensuche die Anordnung der Objekte im Heap durcheinander bringt. Es wäre deshalb sinnvoller bessere Traversiermethoden zu benutzen um eng miteinander verbundene Objekte auf denselben virtuellen Seiten zu speichern. Besonders Datenstrukturen wie Hash-Tabellen profitieren von einer gesonderten Behandlung.

X. Zusammenfassung und Ausblick

Copying Garbage Collection ist auf alle Fälle sinnvoll, wenn die Speicherverwaltung von Allokation dominiert wird und vor allem viele kleine kurzlebige Objekte vorhanden sind, denn in diesem Fall ist das Kopieren sehr günstig. Des Weiteren darf die Verzögerung durch Garbage Collection keine Rolle spielen. Echtzeit-Systeme sind daher ungeeignet für diese Collection-Methode, da für den Collection-Vorgang das System angehalten wird.

Eine Alternative zu Copying Garbage Collection sind hybride Systeme, das heißt ein Teil des Heaps wird mittels Copying behandelt, der Rest von einer anderen Variante wie etwa Mark-Compact oder Mark-Sweep.

Copying Garbage Collection wird sicherlich oft in seiner „Reinform“ benutzt, doch bildet er auch die Grundlage für andere Verfahren wie generationale oder inkrementelle Garbage Collection.

XI. Literatur

[1] Richard E. Jones and Rafael D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, New York 1996, 28-33/117-142

[2] C. J. Cheney. A non recursive list compacting algorithm. Communications of the ACM, 13(11): 677-8, November 1970

[3] Robert R. Fenichel and Jerome C. Yochelson. A List garbage collector for virtual memory computer systems. Communications of the ACM, 12(11): 611-612, November 1969

[4] Benjamin Zorn. Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection. ACM conference on LISP and functional programming, 1990

[5] David A. Moon. Garbage collection in a large LISP system. pages 235-245

[6] Guido Tack, Copying Garbage Collection, http://www.ps.uni-sb.de/courses/gc-ws01/slides/copying_gc.pdf

[7] Stephen P. Thomas and Richard E. Jones. Garbage collection for shared environment closure reducers. Technical Report 31-94, University of Kent and University of Nottingham, December 1994

[8] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. 23(11): 1-17, 1988

[9] Marvin L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963

[10] Andrew W. Appel. Compilers and runtime systems for languages with garbage collection.

[11] Andrew W. Appel. Garbage collection can be faster than stack allocation. Information Processing Letters, 25(4): 275-279, 1987