Copying Garbage Collection

Guido Tack

März 2002

Zusammenfassung

Copying Garbage Collection ist eine einfache, aber effiziente Methode zur automatischen Speicherverwaltung. In dieser Ausarbeitung im Rahmen des Seminars „Garbage Collection“ im Wintersemester 2001/2002 wird die grundlegende Idee hinter *Copying Garbage Collection* beschrieben, Cheneys Algorithmus (der wichtigste – weil effizienteste – *Copying*-Algorithmus) im Detail erklärt sowie auf Vor- und Nachteile und Einsatzgebiete von *Copying Garbage Collection* eingegangen. Abschließend wird das Verhältnis von *Copying Garbage Collection* zu anderen, moderneren Methoden erläutert.

1 Einleitung

In dieser Ausarbeitung wird *Copying Garbage Collection* vorgestellt, eine Methode zur automatischen Speicherbereinigung (*Garbage collection*, im Folgenden durch *GC* abgekürzt).

Copying GC ist eine schon sehr alte Strategie zur Speicherverwaltung, die erste Implementierung stammt von Marvin Minsky aus dem Jahr 1963; Minsky implementierte einen *Copying Garbage Collector* in Lisp 1.5 (siehe [9]). Trotzdem ist sie auch heute noch aktuell, entweder in ihrer „Reinform“ oder als Grundlage modernerer Algorithmen.

Diese Ausarbeitung ist wie folgt gegliedert: Im Abschnitt 2 wird die grundsätzliche Idee beschrieben, die hinter *Copying GC* steht. Abschnitt 3 stellt den wichtigsten und grundlegenden Algorithmus in diesem Bereich vor. Abschnitt 4 beschäftigt sich mit den Nachteilen, Abschnitt 5 mit Variationen von *Copying GC*. Im 6. Abschnitt wird diskutiert, wie die Effizienz des Algorithmus gesteigert werden kann, und Abschnitt 7 zeigt die möglichen Einsatzgebiete von *Copying GC* auf. Abschließend gibt Abschnitt 8 einen Ausblick und eine Zusammenfassung.

2 Die Idee hinter *Copying GC*

Copying GC teilt den logischen Speicher in zwei sogenannte *Semi-Spaces* auf: den *From-Space* und den *To-Space*. Zu jeder Zeit ist nur einer der beiden Speicherbereiche aktiv, nämlich der *From-Space*.

Aktiv bedeutet in diesem Zusammenhang: Alle Operationen des Mutators arbeiten nur in diesem Speicherbereich, alle lebendigen Zeiger zeigen nur in diesen Bereich, und neuer Speicher wird nur in diesem Bereich allokiert.

Wenn die *GC* angestoßen wird (üblicherweise, wenn kein Speicher mehr allokiert werden kann), werden alle lebendigen Objekte aus dem *From-Space* in den *To-Space* kopiert und dann die Rollen der beiden Bereiche vertauscht. Dabei müssen selbstverständlich alle Zeiger auf die kopierten Objekte umgeleitet werden. Abbildung 1 zeigt exemplarisch den Zustand vor und nach dem Kopieren der Objekte.

Durch das anschließende Vertauschen der Rollen wird der *From-Space* zum neuen *To-Space*; das heißt, dass sowohl die ehemals lebendigen Objekte als auch der „Müll“ darin beim nächsten Kopiervorgang überschrieben werden. Insofern ist *Copying GC* eigentlich kein *Garbage Collector*, sondern eher ein *“Live Nodes Collector”*[1].

Minsky's *Collector* benutzte die oben genannte Aufteilung in *Semi-Spaces* noch nicht, sondern kopierte die lebendigen Objekte in eine Datei auf einem Bandlaufwerk, um sie von dort wieder linear in den Speicher einzulesen.

Die Idee der *Semi-Spaces* im Speicher wurde von Robert Fenichel und Jerome Yochelson im Jahr 1969 in [6] beschrieben. Ihre Implementierung kopierte rekursiv die Objekte aus dem *From-Space* in den *To-Space*.

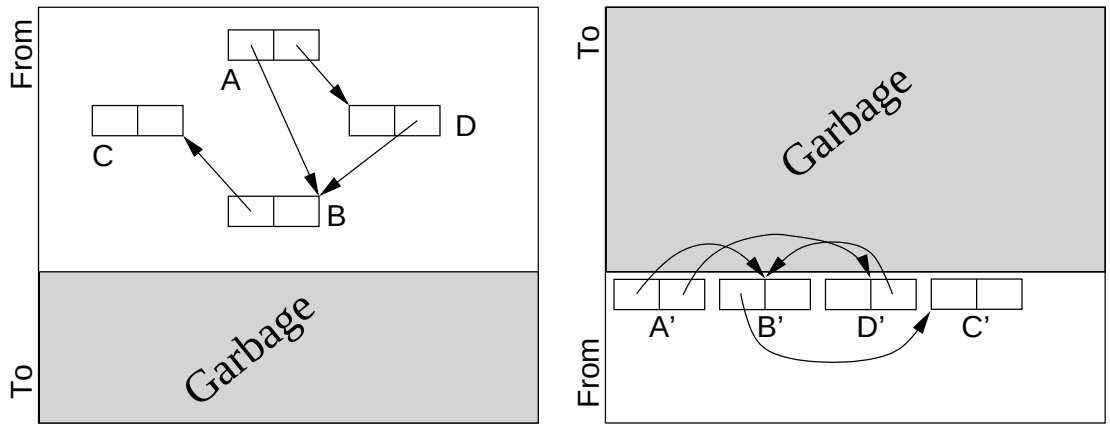


Abbildung 1: Vor und nach dem Kopieren

Abbildung 1 illustriert auch gleich einen großen Vorteil von *Copying GC*: Nach dem Kopieren liegen die Objekte linear und kompakt im *To-Space*, *Copying GC* vermeidet also eine Fragmentierung des freien Speichers.

Ein weiterer Vorteil ist die effiziente Allokation von neuem Speicher, sie erfolgt einfach linear, beginnend hinter dem letzten Objekt im *From-Space*.

[1] Ein solcher *Collector* wird auch oft *Scavenger* genannt.

3 Cheneys Algorithmus

3.1 Motivation

Die naive Implementierung des oben beschriebenen Algorithmus würde den Graph der lebendigen Objekte rekursiv durchsuchen und in den *To-Space* kopieren. Cheney beschreibt in [5] einen Algorithmus, der im Gegensatz dazu iterativ arbeitet und deshalb nur konstanten Laufzeit-Stack benötigt. Cheneys Algorithmus ist die effizienteste Implementierung von *Copying GC*.

3.2 Cheneys Ansatz

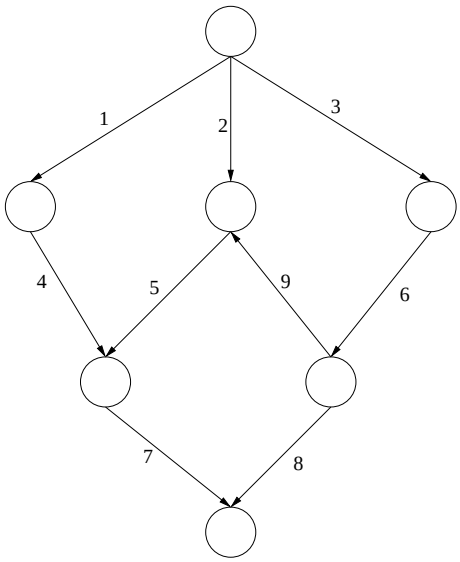
3.2.1 Breitensuche

Um Cheneys Algorithmus zu verstehen, betrachten wir zunächst einen einfachen Algorithmus für Breitensuche *(breadth first search)* in einem Graphen (Abbildung 2). Dieser durchmustert den gesamten Graphen, wobei er jeden Knoten nur einmal in die *Queue* einträgt. Die Laufzeit ist offensichtlich linear in der Anzahl der Knoten.

```
q = new queue
for x in root_set do
  q.pushBack(x)
  x.markAsSeen
end

while not q.isEmpty do
  d = q.popFront
  for x in d.children do
    if x.notYetSeen then
      q.pushBack(x)
      x.markAsSeen
    end
  end
end
```

a) Pseudocode



b) Illustration
(Nummern geben die Reihenfolge der Exploration an)

Abbildung 2: Breitensuche

3.2.2 Eine *Queue* im *To-Space*

Cheneys Idee war jetzt, den *To-Space* als *Queue*-Datenstruktur zu nutzen. Dazu benötigt man zwei Zeiger, im Folgenden ►**free** und ►**scan** genannt, die beide in den *To-Space* zeigen.

►**free** zeigt hinter das letzte kopierte Object und repräsentiert das Ende der *Queue*. ►**scan** zeigt auf das nächste zu scannende Objekt und stellt den Anfang der *Queue* dar. (Siehe Abbildung 3; dort ist der Graph aus Abbildung 1 zu sehen, von dem schon die Objekte A,B und D in den *To-Space* kopiert wurden. A ist schon wieder aus der *Queue* entfernt und somit vollständig abgearbeitet.)

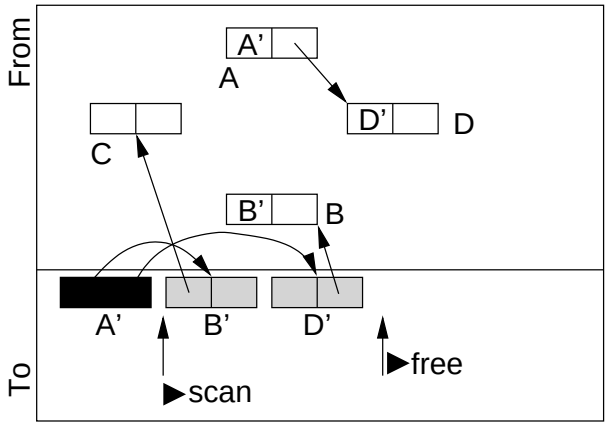


Abbildung 3: ►**scan** und ►**free**, *Forwarding Pointers*
(Der Graph entspricht dem aus Abbildung 1, einige Zeiger wurden durch *Forwarding Pointer* überschrieben: In Objekt A wurde z.B. der Zeiger auf B mit A' überschrieben.)

Die Beschreibung des Algorithmus ist jetzt einfach:

- `q.pushBack` entspricht dem Kopieren des Objekts in den *To-Space* mit anschließendem Verschieben von ►**free** hinter das Ende des Objekts.
- `q.popFront` entspricht dem Auslesen des Objekts an Position ►**scan** mit anschließendem Verschieben von ►**scan** auf das nächste Objekt.
- `q.isEmpty` ist äquivalent zu ►**free**=►**scan**.
  (Das bedeutet insbesondere, dass der Algorithmus terminiert, wenn sich ►**scan** und ►**free** treffen.)

3.2.3 *Forwarding Pointers*

Ein wichtiger Aspekt muss aber noch berücksichtigt werden: Die Zeiger innerhalb der Objekte müssen entsprechend auf die Kopien umgeleitet werden.

Der erste Fall ist dabei einfach: Wenn der Algorithmus ein Objekt erreicht und kopiert, schreibt er den Zeiger, über den das Objekt erreicht wurde, auf die neue Adresse um (dieser Fall trifft auch auf Zeiger aus dem *Root-Set* zu).

Etwas schwieriger wird es beim zweiten Fall: Was passiert, wenn ein schon kopiertes Objekt nochmal über einen Zeiger erreicht wird?

In diesem Fall bedient man sich sogenannter *Forwarding Pointer*. Sobald ein Objekt kopiert wird, wird zum Beispiel im ersten Feld des Original-Objektes die neue Adresse hinterlegt. Wenn dieses Objekt jetzt nochmal erreicht wird, merkt der Algorithmus, dass es sich um ein schon kopiertes

Objekt handelt und kann den Zeiger, über den das Objekt erreicht wurde, entsprechend umleiten (siehe ebenfalls Abbildung 3).

3.2.4 Allokation

Die Allokation von neuem Speicher zwischen zwei *GC*-Zyklen erfolgt linear, hierzu kann ►**free** benutzt werden (der ja, nach Vertauschen der Rollen, jetzt im *From-Space* liegt). Neuer Speicher wird dann allokiert, indem der aktuelle ►**free** als Anfang des Speicher-Blockes zurückgegeben und dann hinter das Ende des Blockes verschoben wird.

Der vollständige Algorithmus ist in Abbildung 4, eine Illustration seiner Vorgehensweise in Abbildung 5 zu sehen.[2] In der Darstellung werden die Objekte in drei Farben dargestellt: Weiße Objekte gelten nach der *Garbage Collection* als „Müll“, graue Objekte wurden schon kopiert, aber noch nicht vollständig bearbeitet, bei schwarzen Objekten ist die Bearbeitung abgeschlossen. Konkret heißt das, dass alle Objekte im *From-Space* weiß sind (und bleiben), in den *To-Space* kopierte Objekte sind zuerst grau, und wenn ►**scan** über sie hinweggelaufen ist werden sie schwarz. Somit sind am Ende der *GC* (wenn ►**scan**=►**free**) alle lebendigen Objekte schwarz.

```
//q = new queue
►new = new pointer
for x in root_set do
  //pushBack(x)
  copy(x, ►free)
  setForwardingPointer(x, ►free)
  x.markAsSeen
  ►free+=sizeof(x)
end
```

```
while not ►scan=►free do
  d = objectAt(►scan) //q.popFront
  for x in d.children do
    if x.notYetSeen then
      //q.pushBack(c)
      copy(x, ►free)
      setForwardingPointer(x, ►free)
      ►new = ►free
      x.markAsSeen
      ►free+=sizeof(x)
    else
      ►new = getForwardingPointer(x)
    end
    update(c.child(x), ►new)
  end
end
```

Abbildung 4: Cheneys Algorithmus, Pseudocode

3.3 Voraussetzungen für den Algorithmus

Cheneys Algorithmus macht folgende Voraussetzungen über Aufbau und Eigenschaften der behandelten Objekte:

- Die Größe eines Objekts muss bekannt sein, um ►**scan** und ►**free** entsprechend inkrementieren zu können.
- Zu jedem Objekt muss bekannt sein, wie man dessen Kinder erreicht, also wo im Objekt Zeiger auf andere Objekte stehen.

[2] Eine animierte Darstellung der Vorgehensweise des Algorithmus findet man in [11].

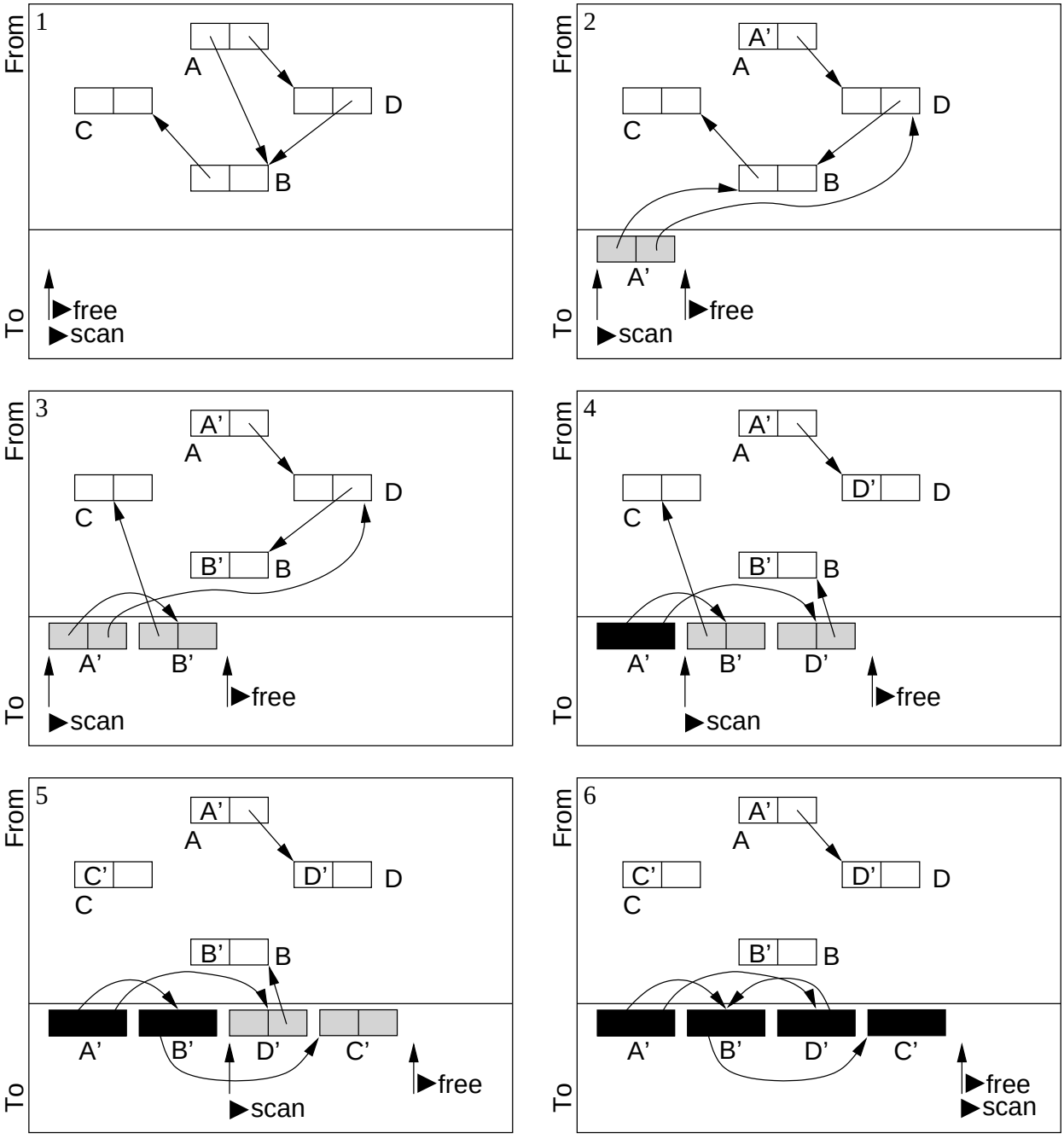


Abbildung 5: Cheneys Algorithmus, illustriertes Beispiel
(Nach dem 6. Schritt folgt noch die Vertauschung der Rollen von *From-Space* und *To-Space*.)

- Objekte müssen als „schon gesehen“ markiert werden können.
- Man benötigt zwei logisch zusammenhängende Speicher-Blöcke.

Die letzte Annahme kann man insofern aufweichen, als dass man den Algorithmus relativ leicht so anpassen kann, dass er eine Kette von *From-* und *To-Spaces* benutzt (siehe Abbildung 6).

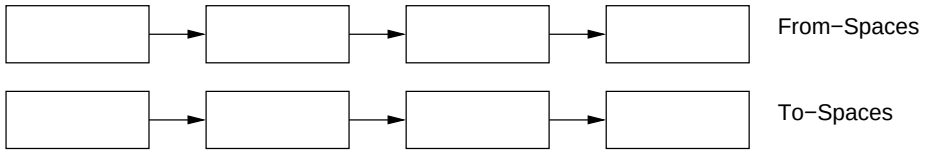


Abbildung 6: Kette von Speicherblöcken

3.4 Eigenschaften des Algorithmus

Cheneys Algorithmus hat einige wichtige Eigenschaften, die ihn interessant machen. Sie wurden teilweise schon oben erwähnt und sollen hier noch einmal zusammengefasst werden:

- Die wichtigste Eigenschaft von Cheneys Algorithmus ist der **Speicherverbrauch**: Es wird nur konstanter Speicher auf dem Stack benötigt. Diese Eigenschaft ist es, die den Algorithmus so effizient macht und die auch zu seiner Verbreitung beigetragen hat.
- Die **Laufzeit** beträgt $O(l)$, wenn l die Anzahl der „lebendigen“ Objekte ist, da nur diese vom Algorithmus betrachtet werden[3].
- **Zyklische Datenstrukturen** sowie ***Sharing***[4] werden korrekt behandelt (mittels *Forwarding Pointers*). Das heißt, dass bei der *GC* die gesamte *Struktur* des Graphen in den *To-Space* kopiert wird, Zykel und *Sharing* also erhalten bleiben.
- Der freie Speicher ist nach der *GC* **kompakt**, es tritt also keine Fragmentierung auf.
- Cheneys Collector ist ein **Stop/Start-Collector**, der Mutator muss also für die *GC* angehalten werden.
- Zwischen zwei *GC*-Zyklen ist dafür aber keine weitere Arbeit zu leisten, insbesondere sind keine Lese- oder Schreibbarrieren nötig.

4 Nachteile der *Copying GC*

4.1 Auf hoher Ebene

Die Laufzeit von *Copying GC* hängt nicht nur von der Anzahl der lebendigen Objekte, sondern auch von deren Größe ab. Während es bei kleinen Objekten keinen nennenswerten Unterschied macht, ob man sie nur markiert oder kopiert, kann das bei großen Objekten stark ins Gewicht

[3] In [1] wird beschrieben, dass in SML/NJ während der *GC* nur etwa 2% der Objekte überleben; diese Eigenschaft trägt also stark zur Effizienz bei.

[4] *Sharing* bedeutet, dass mehrere Zeiger auf dasselbe Objekt verweisen

fallen. Insbesondere für große, langlebige Objekte (die folglich oft kopiert werden müssen) ist *Copying GC* also nicht optimal.

Ein weiterer Nachteil kann sein, dass die doppelte Menge an logischem Speicher nötig ist. Das stellt auf modernen Rechnern dank virtuellem Speicher kein größeres Problem dar, kann aber evtl. bei kleinen Architekturen wie zum Beispiel Handhelds gegen *Copying GC* sprechen (siehe zum Beispiel [3]).

4.2 Auf niedriger Ebene

Die Breitensuche, die zum Bespiel in Cheneys Algorithmus benutzt wird, führt dazu, dass Objekte, die zuvor nahe beieinander im Speicher lagen, nach der *GC* weit voneinander entfernt liegen können. Die sogenannte **Lokalität** wird also zerstört.

Man kann annehmen, dass logisch, also vom Programmfluss her zusammenhängende Objekte kurz hintereinander erzeugt werden und deshalb (wegen der linearen Allokation) nahe zusammen im Speicher liegen (siehe [7]). Wenn jetzt also nach der *GC* wieder auf diese Objekte zugegriffen wird, liegen sie weit auseinander, so dass *Cache misses* oder sogar *Page faults* auftreten können, die zu Performanz-Einbußen führen.

Diese Aspekte sind aber so stark von der Implementierung und der Architektur, auf der diese laufen soll, abhängig, dass hier nicht weiter darauf eingegangen wird.

5 Variationen

Ausgehend von den im vorigen Abschnitt vorgestellten Nachteilen könnte man folgende Variationen in Betracht ziehen:

1. **Spezielle Behandlung großer Objekte** (*Large Object Areas*).
   Große Objekte werden aus dem Zuständigkeitsbereich des *Copying Garbage Collectors* herausgenommen und separat, mit einer anderen *GC*-Strategie, behandelt (siehe [4], [13]).
2. **Bereiche langlebiger Objekte**
   Objekte, von denen festgestellt werden kann, dass sie eine lange Lebenszeit haben, werden in speziellen Bereichen gespeichert. Ihre Zeiger werden zwar gescannt (um andere lebendige Objekte zu finden), sie werden aber nie kopiert.
3. **Beibehaltung der Lokalität durch andere Explorations-Strategien**
   Tiefensuche *(Depth first search)* hat die Eigenschaft, Lokalität zu bewahren. Es gibt einige Ansätze, andere Explorationsstrategien, die der Tiefensuche ähnlich sind, zu benutzen, aber dabei ist es immer sehr schwierig, trotzdem mit konstantem Laufzeit-Stack auszukommen. Ohne die Eigenschaft kann man den Algorithmus nicht mehr als effizient betrachten. Einige Beispiele kann man in [8] finden, genauere Informationen zum Beispiel in [10] oder [12].

Besonders der Punkt 2 ist dabei interessant. Die Forschung auf diesem Bereich führte zur Entwicklung der generationalen *GC*, bei dem unterschiedlich alte Objekte (Objekte verschiedener

Generationen) unterschiedlich behandelt werden.

6 Verbesserung der Effizienz

Hier soll eine relativ einfache Möglichkeit vorgestellt werden, die Effizienz eines *Copying Garbage Collectors* zu verbessern.

Eine erste Beobachtung (die eigentlich trivial ist) besagt, dass die Häufigkeit, mit der der *Garbage Collector* aufgerufen wird, abnimmt, wenn die *Semi-Spaces* größer sind. Dabei spielt aber noch eine weitere Tatsache eine Rolle: Üblicherweise ist der Speicherplatzbedarf der lebendigen Objekte über die Laufzeit des Programmes relativ konstant. Das heißt aber, dass der *Garbage Collector*, obwohl er seltener aufgerufen wird, die gleiche Menge an lebendigen Objekten kopieren muss (siehe Abbildung 7).

Mehr noch: Je seltener die *Garbage Collection* einsetzt, desto älter werden die Objekte. Es ist also wahrscheinlich, dass bei größeren *Semi-Spaces* zum Zeitpunkt der *GC* sogar **weniger** Objekte lebendig sind.

Dies alles ist natürlich nur relevant, weil die Laufzeit von *Copying GC* nur von der Anzahl der lebendigen Objekte abhängt.

Theoretisch lässt sich die Effizienz auf diesem Weg beliebig steigern, wie in [2] beschrieben wird. Die Vergrößerung der *Semi-Spaces* ist aber selbstverständlich nur dann sinnvoll, wenn sie innerhalb des physikalisch verfügbaren Speichers passiert. Ein Auslagern eines *Semi-Spaces* auf Festplatte zum Beispiel hätte katastrophale Auswirkungen auf die Laufzeit.

7 Einsatzgebiete von *Copying GC*

Wann ist es also sinnvoll, einen *Copying Garbage Collector* einzusetzen? Hier lassen sich drei Kriterien identifizieren:

1. **Die Speicherverwaltung wird von Allokation dominiert.**
   Bei Systemen mit dieser Eigenschaft kann *Copying GC* große Vorteile gegenüber anderen Verfahren bieten, da Allokation extrem billig ist.

2. **Es gibt viele kleine, kurzlebige Objekte.**
   Hier kann *Copying GC* punkten, da nur lebendige Objekte behandelt werden. Außerdem ist das Kopieren kleiner Objekte fast genauso teuer wie das Markieren (zum Beispiel also im Vergleich zu *mark-sweep*).

3. **Die Verzögerung durch *GC* spielt keine Rolle.**
   Echtzeit-Systeme sind also kein typisches Anwendungsbeispiel für einen klassischen *Copying Garbage Collector*.

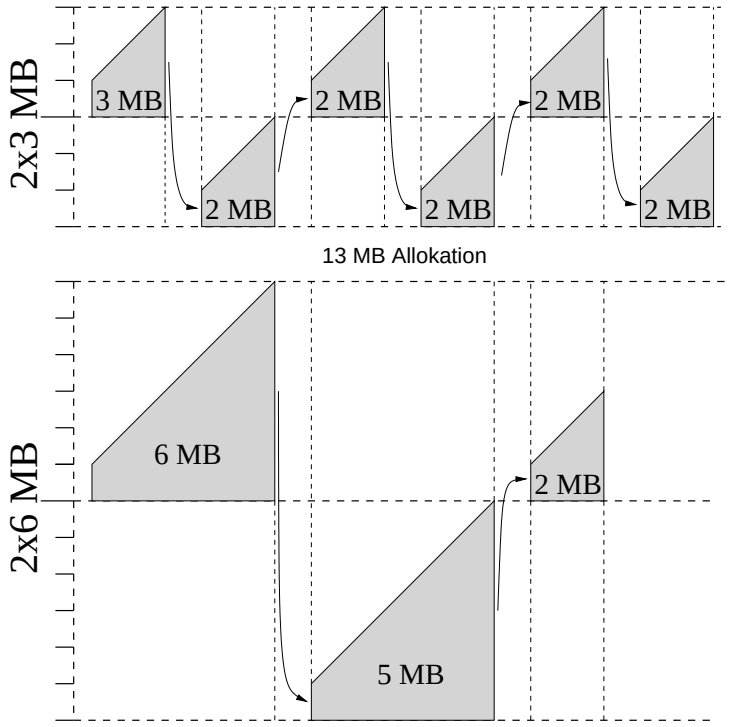


Abbildung 7: Effizienzsteigerung durch größere *Semi-Spaces*

8 Zusammenfassung und Ausblick

Copying GC ist eine effiziente Methode, um eine automatische Speicherverwaltung zu implementieren. Cheneys Algorithmus ist die effizienteste Implementierung von *Copying GC*; effizient wird sie dadurch, dass nur konstanter Stack zur Laufzeit benötigt wird und die Laufzeit der *GC* nur von der Anzahl der lebendigen Objekte abhängt.

Es wurde gezeigt, wie die Laufzeit von *Copying GC* von dem Parameter der Größe der *Semi-Spaces* abhängt. Außerdem wurden Probleme behandelt, die durch *Copying GC* auftreten, insbesondere was große Objekte und Lokalität angeht.

Abschließend wurde eine Bewertung gegeben, in welchen Zusammenhang die Benutzung eines *Garbage Collectors* mit *Copying* sinnvoll sein kann und wo eher nicht.

Die Idee, große Objekte getrennt zu behandeln, führt zur Entwicklung von hybriden Systemen, in denen ein Teil der *GC* von einem *Copying Collector* gemacht wird, ein anderer Teil aber zum Beispiel von *Mark-Sweep* oder *Mark-Compact*.

Als Ausblick sollte noch erwähnt werden, dass *Copying GC* die Basis für viele andere *GC*-Verfahren bildet: sowohl generationale als auch inkrementelle Verfahren bauen oft auf *Copying* auf.

Literatur

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

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

[3] Fred Bayer, LispMe (Scheme for the Palm Pilot), `http://www.lispme.de/doc/lm_hood.htm#gc`.

[4] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. pages 119–130.

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

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

[7] Barry Hayes. Using key object opportunism to collect old objects. pages 33–46.

[8] Richard E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, New York, 1996.

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

[11] Guido Tack, Copying GC, Seminar-Folien, `http://www.ps.uni-sb.de/~tack/gc/copying_gc.pdf`.

[12] 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.

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