Inkrementelle Speicherbereinigung

Uwe Kern

27. März 2002

Zusammenfassung

Inkrementelle Speicherbereiniger erlauben es, Mutator und Collector verschränkt miteinander laufen zu lassen anstatt den Mutator während der Speicherbereinigung anzuhalten. Der folgende Artikel beschreibt anhand dreier Beispiele, wie bestehende Tracingverfahren inkrementalisiert werden können sowie die dabei auftretenden Probleme. Behandelt werden dabei Mark-Sweep, Copying und Generational Collection. Die Darstellung basiert auf dem 8. Kapitel nach [1].

1 Einführung

Klassische Verfahren zur Speicherbereinigung wie Mark-Sweep, Copying aber auch neuere wie Generational Collection haben alle eines gemeinsam: Solange der Collector seinen Sammelzyklus nicht vollendet hat, muß der Mutator warten.

Für einen Applikationsbereich wie Echtzeitsysteme ist dies nicht akzeptabel, da hier dem Mutator feste Zeitschranken auferlegt sind. Der Collector darf hier also nicht beliebig die Aktivitäten des Mutators anhalten.

Auch in interaktiven Systemen sind für den Benutzer mehrsekündige und aus seiner Sicht grundlose Wartepausen inakzeptabel. Diesem Problem läßt sich schon gut mit Hilfe der Generational Collection beikommen, da dieses Verfahren die durchschnittliche Dauer eines Sammelzyklus drastisch verkürzt. Aber auch auf diese Art kann man immer noch keine festen Zeitgrenzen für den schlechtesten Fall angeben.

Der hier vorgestellte Lösungsansatz läßt sich wie folgt beschreiben: Anstatt den Mutator anzuhalten läuft er verschränkt mit dem Collector. Wir werden uns im folgenden damit beschäftigen, wie man die bestehenden Tracingverfahren so abändert, daß sie mit dem Mutator verschränkt laufen können, kurz: Wir inkrementalisieren diese Collectoren.

Synchronisationsmechanismen

Zuerst wollen wir uns die naive Lösung betrachten: Wir lassen Mutator und Collector ohne weitere Maßnahmen parallel laufen. Daß dies nicht funktioniert, sei an folgendem Beispiel demonstriert:

Wir betrachten drei Zellen A, B und C. A und B sind zu Beginn über die Wurzelmenge (Rootset) erreichbar, in B existiert ein Verweis auf C – somit ist C ausschließlich über diese Referenz erreichbar.

Zuerst soll der Mutator zum Zuge kommen: Über die Wurzelmenge findet er A, besucht diese Zelle und stellt fest, daß von A aus keine weiteren Zellen referenziert werden. Die Behandlung von A durch den Collector ist damit abgeschlossen.

Nun kommt der Mutator an die Reihe: Er löscht im Rahmen der Programmausführung die Referenz auf C in B und setzt danach in A eine Referenz auf C.

Wenn nun der Collector in seiner Arbeit fortfährt, wird er als nächstes einen Verweis auf B in der Wurzelmenge finden. In B befindet sich mittlerweile keine weitere Referenz auf irgendeine andere Zelle.

Das Rootset wurde nun vollständig abgearbeitet, folglich ist der aktuelle Sammelzyklus beendet. Der Collector hat dabei A und B gefunden, alle anderen Zellen werden als Abfall behandelt.

Somit wird C wieder dem Freispeicher zugeschlagen, aber diese Zelle ist noch lebendig, da ja A auf C verweist ! Wir haben es mit einer klassischen Wettlaufbedingung zu tun. Mutator und Collector müssen synchronisiert werden.

1.1 Trikolorierung der Zellen

In Zukunft werden wir den Status einer Zelle bezüglich des Collectors mittels eines Farbschemas ausdrücken, das erstmals von Dijkstra [2] eingeführt wurde:

- **Weiße Zellen** wurden vom Collector im aktuellen Sammelzyklus noch nicht besucht. Eine Zelle, die am Ende des Zyklus immer noch weiß ist, ist Abfall.
- **Graue Zellen** wurden vom Collector zwar bereits gefunden, aber sie wurden noch nicht gescannt. Das bedeutet, daß der Collector noch nicht die in ihnen enthaltenen Referenzen untersucht hat.
- **Schwarze Zellen** hat der Collector vollständig abgearbeitet.

Am Ende eine Sammelzyklus gibt es nur noch schwarze und weiße Zellen, wobei die schwarzen die lebendigen und die weißen den Abfall darstellen. Im Gegensatz zu nichtinkrementellen Algorithmen kann eine schwarze Zelle im Verlauf eines Sammelzyklus bei inkrementellen Verfahren wieder grau oder weiß gefärbt werden.

2 Barrieren

Das ganze Problem im obigen Beispiel entstand dadurch, daß der Mutator sozusagen hinter dem Rücken des Collectors den Erreichbarkeitsgraphen abgeändert hat. Wäre der Collector darüber informiert worden, hätte er entsprechende Maßnahmen ergreifen können.

Im folgenden wird mit Hilfe der Trikolorierung beschrieben, welche Änderungen des Erreichbarkeitsgraphen kritisch für den Collector sind. Anhand eines Beispiels wird geschildert, wie man derartige Zustände vermeidet.

Die Problematik von schwarz-zu-weiß Referenzen

Nicht jede Änderung des Erreichbarkeitsraphen hat automatisch katastrophale Konsequenzen. Das oben geschilderte Problem entsteht, wenn der Mutator eine Referenz auf eine vom Collector noch nicht besuchte (weiße) Zelle in eine schwarze, also vom Collector bereits vollständig abgearbeitete, Zelle schreibt.

Dies allein bleibt unproblematisch, solange es einen Pfad von einer grauen Zelle zu der referenzierten weißen Zelle noch gibt, da der Collector diese dann doch noch finden wird. Alternativ zu einer grauen Zelle kann man auch einen Pfad vom Rootset aus nehmen, sofern der entsprechende Startverweis noch nicht vom Collector untersucht worden ist.

Wurde jedoch eine schwarz-zu-weiß Referenz erzeugt und existiert kein derartiger Pfad mehr, wird die weiße Zelle fälschlicherweise zu Abfall erklärt. Verhindert man das Eintreten eine der beiden obigen Voraussetzungen, kann es auch nicht mehr zu dem geschilderten Problem kommen.

Eine Beispielbarriere

Eine Möglichkeit dafür ist, daß man die Erzeugung von schwarz-zu-weiß Referenzen verhindert, indem man die referenzierte weiße Zelle sofort grau färbt. Anhand einer statischen Programmanalyse kann man nicht feststellen, ob ein Schreibvorgang eine derartige Referenz erzeugt oder nicht, da der Collector die Farbe jeder Zelle jederzeit abändern kann.

Es bleibt nichts weiter übrig als jeden einzelnen Schreibzugriff abzufangen, um die nötigen Tests und Änderungen durchzuführen. Man spricht in diesem Zusammenhang von einer Barriere, genauer einer Schreibbarriere.

Lesebarrieren fangen hingegen Lesezugriffe ab und können jede referenzierte weiße Zelle in eine graue oder gar schwarze verwandeln – der Mutator sieht dann nur schwarze oder graue Zellen, wodurch unser Problem auch verhindert wird.

Im obigen Beispiel kann man eine Schreibbarriere einsetzen, die bei jedem Setzen einer Referenz in eine schwarze Zelle die referenzierte Zelle grau färbt sofern sie noch weiß ist.

Dadurch wird das Problem gelöst: Sobald der Mutator in A eine Referenz auf C schreibt, färbt die Barriere C grau. Der Collector wird damit gezwungen, C vor Ende des Zyklus zu besuchen, wodurch C irgendwann schwarz gefärbt wird. Damit wird die Zelle nicht fälschlicherweise zu Abfall erklärt.

Konservative Approximation des Erreichbarkeitsgraphen

Die Beispielbarriere liefert dem Collector kein exaktes Abbild des veränderten Erreichbarkeitsgraphen. Wird in A die Referenz auf C noch vor Ende des Sammelzyklus vom Mutator wieder gelöscht, greift die Barriere nicht ein und C bleibt grau, obwohl die Zelle jetzt wirklich Abfall ist.

Führt dies zu ähnlich gravierenden Problemen wie das Freigeben von lebendigen Zellen ? Nein, denn auch wenn eine Zelle auf diese Art als lebendig markiert wird, obwohl sie bereits tot ist, wird sie im nächsten Sammelzyklus nicht mehr entdeckt werden können und folglich als Abfall dem Freispeicher zugewiesen werden.

Wollte man hingegen dem Collector stets ein vollkommen korrektes Abbild des Erreichbarkeitsgraphen liefern, müsste die Barriere auch beim Löschen einer Referenz aktiv werden, um festzustellen, ob die ehemals referenzierte Zelle grau

oder schwarz war. Wenn ja müßte geprüft werden, ob noch andere Referenzen auf diese Zelle existieren. Gibt es keine solchen Verweise, wäre ein Weißfärben der Zelle und eine rekursive Wiederholung der ganzen Prozedur für alle von ihr referenzierten Zellen notwendig.

Dieser Aufwand brächte sehr hohe Kosten mit sich und scheidet in der Praxis aus. Eine Näherung des Erreichbarkeitsgraphen reicht also vollkommen aus, nur muß diese konservativ sein, das heißt: Sie muß dafür Sorge tragen, daß lebendige Zellen stets vom Collector entdeckt werden, auch wenn dabei einige tote Zellen nicht gleich zu Abfall erklärt werden.

2.1 Qualitätskriterien

Die Qualität einzelner inkrementeller Speicherbereiniger kann man hauptsächlich an den folgenden Kriterien festmachen:

- Zusatzkosten für den Mutator
- Konservativität der Approximation
- Grad an Inkrementalität

Mutatorzusatzkosten Der erste Punkt umfaßt die durch den Bereiniger entstehenden zusätzlichen Speicher- und Laufzeitkosten. Insbesondere die Barriere sowie bei nebenläufigen Algorithmen die zusätzlichen Synchronisationskosten schlagen hier zu Buche. Daneben spielen oftmals noch die Kosten zur Anforderung neuer Speicherzellen eine Rolle.

Konservativität Je höher der Konservatismus eines Algorithmus, desto weniger freier Speicher kann in einem Zyklus geschaffen werden. Zudem muß der Collector unnötigerweise größere Bereiche des Erreichbarkeitsgraphen absuchen als eigentlich nötig ist, was wiederum die Laufzeit in die Höhe treibt. Deswegen sollte die Konservativität möglichst niedrig gehalten werden.

Inkrementalitätsgrad Wie sich bald zeigen wird, erfordert fast jedes Verfahren, daß bestimmte Aktionen des Collectors nicht unterbrochen werden dürfen. Je kürzer deren Zeitverbrauch ist, desto besser. Länger andauernde atomare Aktionen sollten sehr selten sein, am besten gibt es gar keine.

3 Nebenläufiges Mark-Sweep

Als erstes Verfahren betrachten wir ein von Steele [3] veröffentlichtes Verfahren, bei dem Mutator und Collector parallel ausgeführt werden. Die zur Synchronisation verwendete Schreibbarriere weist große Ähnlichkeiten mit der oben geschilderten Beispielbarriere auf.

3.1 Das Verfahren

Steeles Mark-Sweep Collector benutzt neben dem obligatorischen Markierungsbit in jeder Zelle einen eigenen Markierungsstapel. Im Sinne der Trikolorierung kann man die Zellen folgendermaßen einteilen:

- Eine Zelle ist weiß, wenn ihr Markierungsbit nicht gesetzt ist und sie sich nicht auf dem Stapel befindet.
- Eine Zelle ist schwarz, wenn das Bit gesetzt ist.
- Ist sie unmarkiert und befindet sich ihre Adresse auf dem Stapel, so ist sie grau.

Die Markierungsphase

Der Collector arbeitet nach dem üblichen Markierungsverfahren: Zuerst wird die Adresse eines bisher noch nicht besuchten Elementes aus dem Rootset auf den Stapel gelegt. Derartige Elemente besitzen wie jede andere Zelle auch ein Markierungsbit.

Solange der Stapel nicht leer ist, wird das oberste Element genommen und die referenzierte Zelle untersucht. Ist das Markierungsbit noch nicht gesetzt, so wird dies nun getan und alle Referenzen in dieser Zelle werden verfolgt; die Adressen von Zellen, deren Markierungsbit noch nicht gesetzt ist, werden auf den Stapel gelegt.

Ist das Bit dagegen gesetzt, so wird der Eintrag stillschweigend vom Stapel genommen. Zu einem solchen Fall kann es kommen, wenn dank den Aktivitäten der Schreibbarriere die Adresse einer Zelle mehrfach auf den Stapel kommt. In diesem Fall wird nur der oberste Eintrag behandelt und alle anderen im Nachhinein entfernt.

Der Mutator läuft wie erwähnt parallel nebenher. Die Schreibbarriere untersucht in der Markierungsphase, ob in der zu beschreibenden Zelle das Markierungsbit schon gesetzt wurde, sie also schwarz ist, und das Bit in der referenzierten Zelle gelöscht ist. Ein gelöschtes Bit zeigt eine weiße oder graue Zelle an – unterscheiden kann die Barriere das nicht, dazu müsste sie den kompletten Markierungsstapel durchsuchen.

Also legt sie die Adresse der referenzierenden Zelle auf jeden Fall auf den Stapel, womit die Zelle garantiert grau wird. Auf diese Weise kann es natürlich für eine einzige Zelle zu beliebig vielen Einträgen im Stapel kommen.

Die Barriere selbst läuft innerhalb des Mutatorthreads. Deshalb müssen sowohl die Zugriffe auf den Markierungsstapel als auch die Zugriffe auf die Markierungsbits synchronisiert werden. Der Erwerb von Sperren bei nahezu jeder Aktion des Collectors und bei Schreiboperationen des Mutators ist daher die Regel.

Speicherallokation Was passiert beim Allozieren einer neuen Speicherzelle ? Hier gibt es keine speziellen Vorkehrungen. Irgendwo wird kurz nach der Allokation eine Referenz auf die neue Zelle gesetzt werden, wodurch die Schreibbarriere aktiviert wird. Diese behandelt die neue und die referenzierende Zelle nach dem oben vorgestellten Verfahren. Neue Zellen werden somit weiß allokiert.

Die Auskehrphase

Der Collector leitet die Kehrphase ein, wenn das Rootset vollständig abgearbeitet wurde und der Markierungsstapel leer ist. Zu beachten ist dabei, daß die Barriere sich in ihrem Verhalten zwischen Markier- und Kehrphase unterscheidet. Folglich muß irgendwo gespeichert sein, in welcher Phase sich der Collector

befindet. Das Abändern des Zustandes erfordert wiederum eine Synchronisation – sonst könnte die Barriere wieder ein Element auf den Markierungsstapel legen, nachdem als neuer Zustand schon Fegen gesetzt wurde.

Alle Zellen, deren Markierungsbit nicht gesetzt ist, werden beim Ausfegen dem Freispeicher zugeschlagen. Dies ist insbesondere beim Anfordern neuer Speicherzellen wichtig: Liegt die neue Zelle im Speicher vor dem Kehrzeiger, so muß ihr Markierungsbit gesetzt werden, oder sie wird vom Collector wieder in den Freispeicher eingetragen werden.

Liegt sie dagegen hinter dieser Front, so muß das Markierungsbit gelöscht bleiben – sonst könnte es bei der nächsten Markierungsphase zu seltsamen Effekten kommen: Der Collector würde die neue Zelle als schwarz ansehen. Ihre Referenzen würden nicht untersucht werden und damit eine Zelle, auf die ausschließlich einer dieser Verweise zeigt, nicht entdeckt werden und damit als Abfall enden.

Bei Speicheranforderungen muß also stets getestet werden, in welcher Phase man sich befindet, damit der neue Speicher in der Kehrphase wie oben beschrieben modifiziert werden kann.

3.2 Qualität und Performanz

Konservatismus Die Taktik der benutzten Barriere, bei drohenden schwarz-zu-weiß Referenzen die schwarze Zelle grau zu färben und damit die eventuell noch weiße referenzierte Zelle weiß zu lassen, führt zusammen mit der Allozierung von ebenfalls weißen Speicherzellen zur Produktion von relativ wenig Abfall, der am Ende eines Zyklus nicht sofort dem Freispeicher wieder zugewiesen werden kann. Der Konservatismus des Verfahrens ist also niedrig.

Inkrementalität Der Grad der Inkrementalität ist hoch: Die Barriere selbst besteht nur aus wenigen Anweisungen; grössere atomare Aktionen, in denen der Mutator vom Collector blockiert wird, gibt es nicht.

Mutatorkosten Alle geschilderten Vorteile werden durch die sehr großen Synchronisationskosten wieder aufgehoben. Steele selbst empfiehlt Hardwaresupport, damit diese nicht überhand nehmen. Ein praktischer Einsatz ist daher ausgeschlossen, falls eine Modifikation diese Kosten nicht deutlich herabsetzen kann. Es verwundert daher wenig, daß das Verfahren in der Praxis niemals angewendet wurde.

4 Inkrementelles Copying

Dieses von Baker [4] veröffentlichte Verfahren basiert auf Cheneys klassischem Algorithmus [5]. Der Collector läuft dabei als Coroutine des Mutators, wodurch die zusätzlichen Synchronisationskosten eines echt nebenläufigen Modells vermieden werden.

4.1 Das Verfahren

Wie bei Cheneys Collector startet ein neuer Sammelzyklus dann, wenn neuer Speicher angefordert wird, aber nicht mehr genügend vorhanden ist. Daraufhin

werden Fromspace und Tospace getauscht sowie alle Zellen, auf die das Rootset verweist, in den Tospace kopiert. Es existieren im Tospace zwei Zeiger namens Scan und Bottom: Scan steht direkt links von der Zelle, die als nächstes vom Collector untersucht werden muß; links von Bottom kann die nächste Zelle aus dem Fromspace kopiert werden. Sobald Scan und Bottom aufeinandertreffen, sind alle lebendigen Zellen in den Tospace evakuiert worden.

Nach dem Kopieren der aus dem Rootset referenzierten Zellen beginnt der Collector nun mit dem üblichen Verfahren: Die Zelle hinter Scan wird untersucht und dabei neu gefundene Zellen in den Tospace kopiert. In die alten Zellen im Fromspace wird die Adresse der korrespondierenden neuen Zelle im Tospace eingetragen, die sogenannte Forwardadresse. Die Verweise in der untersuchten Zelle werden auf die neuen Tospaceadressen der kopierten Zellen gesetzt und der Scanzeiger anschließend vorgeschoben.

Dies entspricht genau dem Vorgehen wie man es bei Cheney auch hat. Im Gegensatz dazu fährt der Collector nun aber nicht bis zur Beendigung des Zyklus fort, sondern gibt irgendwann die Kontrolle an den Mutator zurück.

Im Sinne unserer Trikolorierung sind Zellen, die vom Collector noch nicht entdeckt und damit auch noch nicht kopiert wurden weiß, Zellen, die zwar kopiert, aber noch nicht untersucht wurden, grau und die Zellen hinter dem Scanzeiger schwarz.

Eine Lesebarriere

Wie kann man nun den Mutator laufen lassen ohne daß es zu Konflikten mit dem Collector kommt ? Die Grundidee lautet, daß der Mutator immer im Tospace operiert. Die Zellen aus dem Rootset wurden bereits am Anfang kopiert, auf diese greift der Mutator also immer im Tospace zu.

Um zu verhindern, daß er durch Auslesen einer Referenz auf den Fromspace zugreift, wird eine Lesebarriere errichtet. Diese prüft bei jedem lesenden Zugriff, ob die Referenz in den Fromspace führt. Wenn ja, erhält der Mutator anstatt der Zellenadresse im Fromspace die dort gespeicherte Forwardadresse in den Tospace zurück. Sollte die Zelle noch nicht kopiert worden sein, so wird dies zuerst durchgeführt.

Durch dieses Verfahren werden Zellen sehr schnell grau gefärbt; werden während des Sammelzyklus alle Referenzen zu einer Zelle gelöscht, so besteht dennoch eine hohe Chance, daß sie vorher in den Tospace kopiert wird und dort als Abfall verbleibt. Zudem sind lesende Zugriffe wesentlich häufiger als schreibende, so daß dies die Kosten der Barriere zusätzlich in die Höhe treibt.

Speicherallokation

Da der Mutator stets im Tospace operiert, werden neue Speicherzellen auch dort allokiert. Prinzipiell könnte man wie bei Cheneys Collector einfach den Bottomzeiger verschieben, um das Anfordern neuen Speichers zu realisieren. Da der Scanprozeß jedoch noch nicht abgeschlossen ist, würde die neue Zelle vom Collector irgendwann untersucht werden. Alle Referenzen einer neuen Zelle zeigen stets in den Tospace; dies wird durch die Barriere wie oben geschildert garantiert. Deswegen wäre ein solches Scannen vollkommene Zeitverschwendung.

Deshalb erfolgt die Speicherallokation nicht am Bottomzeiger, sondern beginnt am anderen Ende des Tospace. Der Beginn des freien Speichers wird durch

einen Zeiger namens Top angezeigt. Treffen Top und Bottom aufeinander, so ist der Freispeicher erschöpft. Geschieht dies noch während eines Sammelzyklus, so ist der Speicher endgültig erschöpft. Andernfalls wird allokiert, d.h. der nächste Zyklus gestartet.

Neue Zellen werden damit effektiv schwarz erzeugt; während eines Sammelzyklus können sie nicht eingesammelt werden, sollten sie sich in Abfall verwandeln.

Damit bliebe nur noch die Frage, wie der Collector eigentlich aufgerufen wird und auf welche Weise er die Kontrolle wieder an den Mutator zurück gibt. Zum einen werden Zellen natürlich durch die Lesebarriere kopiert. Ansonsten wird der Collector innerhalb der Speicherallokation aufgerufen: Jedesmal, wenn eine neue Zelle angefordert wird, scannt der Collector ein Stück weiter. Danach kehrt die Routine zur Speicheranforderung zurück und der Mutator läuft weiter.

4.2 Qualität und Performanz

Der beschriebene Algorithmus arbeitet einfach und elegant. Im Gegensatz zum obigen Mark-Sweep Verfahren kommt er ohne Synchronisation aus. Auch hier ist die Verschränktheit zwischen Mutator und Collector sehr feinkörnig, atomare Operationen des Collectors gibt es nur zu Beginn – die vom Rootset referenzierten Zellen müssen auf einmal kopiert werden.

Die Barriere selbst ist eine sehr teure Angelegenheit, da Lesezugriffe mit Abstand häufiger als Schreibzugriffe stattfinden, insbesondere bei funktionalen Sprachen, für die die meisten Speicherbereiniger konzipiert wurden. Zudem tendiert das Verfahren durch die schwarze Allokation neuer Zellen und dem schnellen Graufärben bestehender Zellen dazu, während des Sammelns entstehenden Abfall mit in den nächsten Zyklus zu verschleppen, was die Häufigkeit von nötigen Bereinigungen erhöht. Die Konservativität ist also sehr hoch.

So nimmt es denn auch nicht Wunder, daß mehrere Untersuchungen wie [6] der Methode bescheinigen, nahezu ein Drittel oder noch mehr Laufzeit zu verbrauchen.

5 Replizierendes Copying

Als letztes Verfahren wird nun noch der sog. replizierende Copying Collector von Nettles und O'Toole [8] vorgestellt. Dieser basiert auf Appels Generational Collector [9] für SML/NJ. In erster Linie war er zum Einsatz für die major collections gedacht, um die dabei auftretenden Pausenzeiten des Mutators zu reduzieren.

Genau wie Bakers Collector basiert er auf der Technik der Copying Collection. Im Gegensatz zu diesem wird jedoch eine Schreibbarriere verwendet. Die Methode ist zudem echt nebenläufig, ohne derartige Synchronisationskosten zu verursachen wie Steeles Mark-Sweep Collector.

5.1 Das Verfahren

Collector und Mutator laufen beim replizierenden Copying jeweils in einem eigenen Thread. Im Gegensatz zu Bakers Collector läuft der Mutator während des

gesamten Sammelzyklus stets im Fromspace. Erst am Ende, wenn der Collector seine Arbeit getan hat, wird der Mutator auf den Tospace umgesetzt.

Rootset und Shadowrootset

Ein Sammelzyklus beginnt damit, daß der Collector in einer einzigen atomaren Operation eine Kopie des Rootsets erzeugt, das sogenannte Shadowrootset. Hierzu muß der Mutator angehalten werden. Viele Laufzeitsysteme von funktionalen Sprachen wie ML arbeiten nicht mit Stapeln, so daß das Rootset sehr klein ist.

Der Collector arbeitet nun das Shadowrootset ab und beginnt damit, Zellen in den Tospace zu kopieren. Der Mutator arbeitet währenddessen ungestört im Fromspace weiter. Eine Sperre für den Zugriff auf das Rootset ist dank des Shadowsets nicht nötig.

Besonderheiten des Forwardzeigers

Modifiziert der Mutator eine bereits kopierte Zelle, so muß die Änderung irgendwann in seiner Replik im Tospace nachgetragen werden. Um dies zu ermöglichen, trägt der Collector in jede Zelle einen Forwardzeiger ein, bevor er anfängt, sie zu kopieren. Das Eintragen dieses Zeigers kann von modernen Prozessoren in einer einzigen Operation durchgeführt werden, eine Sperre für die Zelle ist somit nicht erforderlich.

Im Gegensatz zu Cheneys Collector darf der Forwardzeiger keine Daten in einer Zelle überschreiben, schließlich arbeitet der Mutator noch mit dem Inhalt der Fromspacezellen. Entweder reserviert man in jeder Zelle einen eigenen freien Bereich, um den Forwardzeiger dort eintragen zu können, oder man überschreibt Zellenteile, die ausschließlich vom Laufzeitsystem benutzt werden. Letzteres ist die Lösung, die in Nettles und O'Tooles Implementierung benutzt wird.

Schreibbarriere und Mutationslog

Versucht der Mutator, eine Zelle zu verändern, so prüft die Schreibbarriere anhand der Existenz eines Forwardzeigers, ob der Collector schon eine Replik angelegt hat bzw. dabei ist, eine anzulegen. Ist dies der Fall, so trägt sie die Änderung in ein Mutationslog ein.

Da der Mutatorthread als einziger Zugriff auf das Mutationslog hat und der Collectorthread auf die Repliken, ist auch hier keine Synchronisation erforderlich.

Wir haben nun alle Elemente beisammen, um eine Einteilung der Zellen im Sinne unserer Trikolorierung vornehmen zu können: Eine Zelle ist weiß solange der Collector sie noch nicht kopiert hat, schwarz wenn die Replik im Tospace schon gescannt wurde und kein Eintrag des Originals im Mutationslog vorhanden ist. Ist die Zelle kopiert, aber noch nicht untersucht worden, so ist sie grau. Eine schwarze Zelle kann wieder grau werden, wenn der Mutator mittels der Schreibbarriere einen Eintrag ins Log vornimmt.

Speicherallokation

Neue Speicherzellen werden stets im Fromspace allokiert, da der Mutator ausschließlich hier arbeitet. Eine neue Zelle ist daher immer weiß. Dies setzt natürlich

voraus, daß noch genügend freier Speicher im Fromspace vorhanden ist. Deswegen muß ein neuer Sammelzyklus gestartet werden noch bevor eine Speicherknappheit auftritt. Sollte dennoch der Speicher ausgehen, wird der Mutator solange blockiert bis der Sammelzyklus abgeschlossen wurde.

Beenden eines Sammelzyklus

Hat der Collector das Shadowrootset abgearbeitet, sowie alle erreichbaren Zellen kopiert und ihre Repliken nach weiteren Zellen gescannt, so wird der Mutator angehalten, damit der Collector das Mutationslog inspizieren kann. Bei jeder eingetragenen Zelle führt der Collector einen Abgleich mit der Replik durch.

Anschließend wird das Shadowrootset mit dem Rootset abgeglichen. Änderungen werden nachgezogen und dabei eventuell noch unbekannte Zellen nach dem geschilderten Verfahren repliziert.

Prinzipiell ist es möglich, während des Replizierens den Mutator wieder laufen zu lassen, jedoch kann dies wieder zu neuen Einträgen im Mutationslog und Änderungen im Rootset führen. In der Implementation von Nettles und O'Toole wurde darauf verzichtet.

Zum Schluß setzt der Collector die Verweise im Rootset auf die entsprechenden Repliken im Tospace um. Damit ist der Zyklus abgeschlossen.

5.2 Qualität und Performanz

Mutatorzusatzkosten Wie aus der Beschreibung des Verfahrens schon ersichtlich wurde, ist der Grad an benötigter Synchronisation sehr niedrig. Damit verblieben als Kostenfaktoren zusätzlich benötigter Speicherplatz, sowie das Mutationslog.

In Nettles und O'Tooles Implementierung verbraucht der Forwardzeiger wie in Cheneys oder Bakers Collector keinen zusätzlichen Speicher. Dies ist nur deshalb möglich, weil in jeder Zelle Laufzeitsystemdaten von zumindest Zeigergröße vorhanden sind, die überschrieben werden können. Derartige Daten fallen bei sehr vielen Laufzeitsystemen von funktionalen und objektorientierten Sprachen an.

Die Kosten eines Mutationslogs sind erträglich, wenn es nicht allzu viele Schreiboperationen gibt – eine Vorraussetzung, die funktionale Sprachen wie ML üblicherweise erfüllen. Das Verfahren setzt zudem auf einem Generational Collector auf, in dem zumindest Schreiboperationen von der alten auf die junge Generation mitgeloggt werden. Ein Mutationslog ist daher schon vorhanden und treibt die Laufzeitkosten gegenüber dem Originalverfahren nicht noch zusätzlich in bedeutender Weise in die Höhe.

Die relative Seltenheit von Schreiboperationen senkt auch die Einsatzhäufigkeit der Barriere und damit ihren Laufzeitverbrauch. Dieser beschränkt sich auf einen Bedingungstest nebst eventuellen Logeintrag.

Inkrementalität Dank der Nebenläufigkeit ist der Grad an Inkrementalität sehr hoch, sieht man von den atomaren Operationen zu Beginn und Ende eines Zyklus ab. Die Dauer dieser Operationen hängt von der Größe des Rootsets und des Mutationslogs ab.

Wie bereits erwähnt, sind Schreiboperationen in ML selten, das Mutationslog somit kurz. In funktionalen Sprachen werden sogar Funktionsaufrufe über die

Halde abgewickelt, mithin gibt es keine Laufzeitstapel, wodurch eine minimale Größe des Rootsets garantiert wird.

Konservatismus Die Allokation neuer Zellen erfolgt weiß, die Schreibbarriere wandelt durch das Mutationslog schwarze in graue Zellen um. Damit ist die Wahrscheinlichkeit recht hoch, daß Zellen, die während des Zyklus zu Abfall werden, auch noch dem Freispeicher zugeordnet werden. Die Konservativität kann somit als niedrig eingestuft werden.

Nettles und O'Toole führten einige Untersuchungen zur Performanz ihres Verfahrens durch. Dabei konnte der Collector ohne Probleme mit der unmodifizierten Originalversion konkurrieren und erreichte die erwünschte Reduktion der Pausenzeiten für den Mutator.

Somit kann man diesen Collector als einzigen der hier vorgestellten für die Praxis tauglich erklären. Das Verfahren nutzt einige Eigenheiten funktionaler Sprachen wie wenige Schreiboperationen und eine kleine Wurzelmenge sehr gut aus. Inwieweit die gute Performanz bei Einsatz in objektorientierten Sprachen mit ihren häufigeren Schreiboperationen und ihrer Verwendung von Laufzeitstapeln erhalten bliebe, wäre zu prüfen.

6 Fazit

Inkrementelle Speicherbereiniger arbeiten im Gegensatz zu den klassischen tracenden Collectoren verschränkt mit dem Mutator. Dies führt zu drastisch reduzierten Pausierungszeiten des Mutators, was insbesondere für interaktive oder Echtzeitanwendungen von Bedeutung ist. Prinzipiell läßt sich jedes der klassichen Verfahren inkrementalisieren; vollkommen neuartige Methoden werden dagegen nicht verwendet.

Um die Synchronisation zwischen Collector und Mutator zu gewährleisten werden sogenannte Barrieren eingesetzt. Diese fangen entweder Lese- oder Schreibzugriffe ab und modifizieren den Status der referenzierenden oder der referenzierten Zelle derart, daß der Speicherbereiniger keine lebendige Zelle übersieht und sie dem Freispeicher zuweist.

Für jedes der drei klassischen Verfahren Mark-Sweep, Copying und Generational Collection wurde eine entsprechend inkrementalisierte Variante vorgestellt. Insbesondere die letzte davon erwies sich für den realen Einsatz als durchaus tauglich.

Literatur

[1] Richard Jones, Rafael Lins: Garbage Collection, Algorithms for Automatic Dynamic Memory Management. *John Wiley & Sons*, 1996

[2] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten und E. F. M. Steffens: On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965-75, November 1978

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

[4] Henry G. Baker: List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280-94, April 1978

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

[6] Skef Wholey und Scott E. Fahlman: The design of an instruction set for Common Lisp. In Steele [7], 150-8, 1984

[7] Guy L. Steele, editor: *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[8] Scott Nettles und James O'Toole: Real-Time replication Garbage Collection. *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, 217-26

[9] Andrew W. Appel: Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171-83, 1989