Günther Gsenger

(Mostly) Concurrent Garbage Collection

***Abstract*—Diese Seminararbeit befasst sich mit (größtenteils) paralleler Garbage Collection. Zuerst wird auf die Grundidee, sowie die Einsatzgebiete eingegangen und es wird ein kurzer geschichtlicher Überblick über die Implementierungen gegeben. Anschließend wird auf den Algorithmus allgemein eingegangen und es werden konkrete Implementierungen von Boehm et. Al [1] und Printezis und Detlefs [2] besprochen. Zum Abschluss werden die Ergebnisse der Benchmarks, die diese beiden Forschergruppen ausgearbeitet haben präsentiert und analysiert.**

***Index Terms*—Computers, Memory Management, Parallel algorithms, Computer languages, C language, Cache Memories, Computer performance, Programming, Computer science, Software**

I. Nomenklatur

In der Literatur zu dem Thema “Garbage Collection” gibt es unterschiedliche Begriffsdefinitionen. Während Boehm et al. [1] das Ausführen eines separaten Collector Threads als “mostly parallel” bezeichnet, so bezeichnen es Printezis und Detlefs [2] als „mostly concurrent“. In dieser Seminararbeit wird für diesen Begriff die Bezeichnung „größtenteils nebenläufig“ benutzt.

Als „Collector“ wird die Implementierung des Garbage Collectors bezeichnet. Der Collector sammelt Datenobjekte ein, die nicht mehr länger referenziert und somit auch nicht mehr länger gebraucht werden, ein.

Ein „Mutator“ ist der Teil eines Programms, der die Datenobjekte verändert, neue hinzufügt usw., also im Grunde alle Teile, die nicht direkt zum Collector zählen.

Als „Objekte“ werden allgemein Speicherobjekte bezeichnet. Hierbei handelt es sich nicht nur um Objekte im objektorientierten Sinn, sondern auch um Arrays, Strukturen etc.

II. Einleitung

Es gibt viele verschiedene Algorithmen, um Garbage Collection zu implementieren. Diese reichen von Reference Counting, Mark & Sweep usw. bis hin zur Generational Garbage Collection [3]. Diese Collectoren haben im Allgemeinen eines gemeinsam: Während sie nicht mehr benötigte Datenobjekte einsammeln, müssen sie die Programmausführung stoppen. Dieser Zeitraum, in dem der Collector den ganzen Ablauf anhält, wird als „Stop-the-world-Phase“ bezeichnet [1][2]. Während dieser Phase kann der Mutator nicht auf I/O Operationen reagieren oder diese Ausführen, er hat keine Prozessorzeit zur Verfügung.

Diese Stop-the-world-Phase bringt also Probleme mit sich: Je länger sie ist, desto länger muss der Benutzer eines Programms auf die Reaktion auf seine Eingaben warten. Dies wird zwar im Allgemeinen nicht besonders häufig auftreten, bzw. wird dies kaum merkbar sein, es kann aber dennoch dazu führen, dass der Benutzer frustriert wird.

Schlimmer ist diese Phase allerdings bei Echtzeitanwendungen: Da ein nicht speziell dafür angepasster Collector die Zeitschranken für das Echtzeitsystem nicht kennen kann, beziehungsweise während seines Einsammelzyklus nicht abbrechen kann, kann es passieren, dass während der Stop-the-world-Phase eine harte Zeitschranke überschritten und nicht mehr eingehalten wird. Bei den Einsatzgebieten von Echtzeitsystemen (in Raketenleitsystemen, Automobilen etc.) könnte dies verheerende Folgen haben. Echtzeitsysteme müssen deshalb Collectoren benutzen, die „preemptiv“, d.h. unterbrechbar sind. Dies wird beispielsweise auch in der Real Time Specification for Java [4] gefordert.

Ein paralleler Garbage Collector ist ein Collector, der arbeitet, während auch der Mutator läuft. Dies hat verschiedene Vorteile, so werden die Stop-the-world-Phasen minimiert und man kann den Collector in Multiprozessorarchitekturen auf einem eigenen Prozessor laufen lassen (vorausgesetzt die Prozessoren können auf den selben gemeinsamen Speicher zugreifen). Es ist offensichtlich, dass normale Garbage Collectoren nicht einfach nebenläufig zum Mutator ausgeführt werden können, so würden beispielsweise ohne Synchronisation die vom Allokator verwendeten Blocklisten überschrieben werden; aber es treten auch noch andere Probleme auf, die im nächsten Kapitel genauer erläutert werden.

1991 beschäftigte sich eine Forschergruppe rund um Hans-J. Boehm im Xerox PARC erstmals mit der Erstellung eines Garbage Collectors, dessen Stop-the-world-Phase möglichst kurz sei. Sie entwickelten einen Collector, der ohne Hilfe von Compiler oder Betriebssystem auf UNIX Systemen für in C geschriebene Programme eingesetzt werden konnte [1]. Basierend auf dieser Arbeit entwickelten Tony Printezis und David Detlefs im Jahr 2000 einen ähnlichen Garbage Collector für die Java Virtual Machine [2]. Dieser Garbage Collector ist heute ein fixer Bestandteil der Sun JVM 5.0 und kann über gewisse Kommandozeilenargumente aktiviert werden [5].

III. Die Grundidee des Algorithmus

Nebenläufiges Mark & Sweep

Um die Stop-the-world-Phase abzuschaffen, kann man zunächst versuchen, einen seperaten Collector-Thread zu

starten, der unabhängig davon, was der Mutator macht, eine Garbage Collection durchführt. Dass es dabei zu Race-Conditions, also in Folge zu ungewollten Zuständen kommen kann, ist intuitiv leicht nachvollziehbar. Betrachtet man beispielsweise nur die als Wurzelobjekte bezeichneten [1] Speicherobjekte, also die Register, statisch reservierten Speicherbereiche und den Stack, so kann es hier bei paralleler Ausführung des Collectors und des Mutators zu unvorhersehbaren Zuständen kommen, wenn beispielsweise der Mutator Register und den Stack verändert, während der Collector diese gerade analysiert. Boehm et al. [1] erkannten deshalb, dass sich Stop-the-world-Phasen nicht gänzlich vermeiden lassen. Der Collector muss das ganze Programm anhalten, um eine Analyse der Wurzelobjekte durchzuführen.

Ein Collection-Zyklus lässt sich also in 3 Phasen unterteilen:

- In der ersten Phase werden die Wurzel-Speicherobjekte bestimmt und analysiert. Der Collector markiert sämtliche Objekte, die von den Wurzelobjekten aus referenziert werden. Diese müssen in eine Liste aufgenommen werden. Wie bereits vorhin erwähnt, muss diese Phase eine Stop-the-world-Phase sein, damit nicht der Mutator Wurzelobjekte verändert, während der Collector diese analysiert.
- Von den in der ersten Phase markierten Objekten ausgehend werden rekursiv alle von diesen referenzierten Objekte gefunden und markiert. Anders als bei „traditionellen“ Collectoren wird diese Phase nebenläufig zum Mutator ausgeführt. Zu welchen Zuständen dies führen kann und wie Sonderfälle behandelt werden, wird später in diesem Kapitel beschrieben.
- In der letzten Phase werden alle nicht markierten Objekte freigegeben. Auch in dieser Phase muss das Programm nicht angehalten werden, denn es gibt keinen Grund, warum das Freigeben und Reservieren von Speicher nicht parallel ablaufen sollte [1].

Wie kann nun die zweite Phase einfach nebenläufig zum Mutator ausgeführt werden, ohne dass Objekte übersehen oder versehentlich freigegeben werden? Dazu bedarf es der Unterstützung der Hardware, des Betriebssystems, bzw. der Virtuellen Maschine, auf der das Programm läuft. Ohne dieser Unterstützung könnte folgendes Szenario eintreten: Angenommen, im Speicher liegt eine einfach verkettete Liste (Fig. 1). Der Collector beginnt rekursiv alle verwendeten Objekte zu markieren. Dabei beginnt er beim Kopf (a). Danach werden rekursiv alle referenzierten Objekte markiert. Wenn der Collector bei dem zweiten Knoten der Liste ankommt, verändert aber in diesem Beispiel der Mutator die Liste: Der erste Knoten zeigt nunmehr auf das Objekt C und der dritte Knoten zeigt auf kein Objekt mehr (b). Da der Collector aber den ersten Knoten der Liste bereits „besucht“ hat und nichts von der Veränderung mitbekommt, läuft er normal weiter über den dritten Knoten und zum Ende der Liste (und in unserem Beispiel dem Ende des Heaps) (c). In der nächsten Phase werden nun alle nicht markierten Objekte freigegeben. Objekt C wurde nicht markiert und wird dadurch gelöscht, Objekte A und B sowie die Knoten der Liste wurden markiert und werden somit nicht gelöscht. Das Ergebnis dieses Löschvorgangs ist ein unerwünschter Zustand: Objekt C wird noch referenziert, wurde aber bereits freigegeben, Objekt A wird nicht mehr länger referenziert, wurde aber nicht freigegeben (d).

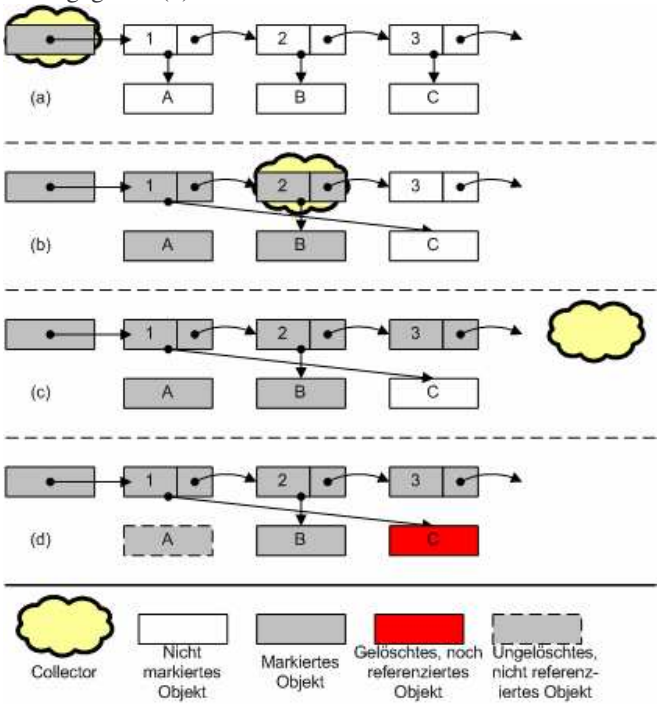

Fig.1. Mark & Sweep mit parallelem Verändern einer einfach verketteten Liste und die Konsequenzen.

Das A Objekt ist in dieser Situation kein größeres Problem. Es wird bei späteren Durchläufen des Garbage Collectors gefunden und freigegeben werden. Das Problem ist offensichtlich, dass das C Objekt freigegeben wurde, obwohl es noch vom Knoten 1 referenziert wird. Nun wird auch klar, welche Unterstützung der Plattform für den Collector benötigt wird, um ein fehlerfreies Einsammeln von nicht mehr länger referenzierten Objekten zu ermöglichen. Probleme treten immer dann auf, wenn während eines Collection-Zyklus, also der zweiten Phase bei der größtenteils nebenläufigen Garbage Collection, der Mutator Objekte verändert, die bereits vom Collector rekursiv durchlaufen wurden.

Nebenläufige Garbage Collection mit Unterstützung der Plattform (Write-Barriers)

Hätte in unserem Beispiel (Fig. 1) der Mutator den ersten Knoten nicht so verändert, dass er das C Objekt referenziert hätte, so wäre dieses am Ende des Collection Zyklus von keinem anderen Knoten (bzw. Objekt) mehr referenziert worden und hätte freigegeben werden dürfen. Allgemein bedeutet das, dass noch nicht markierte Objekte vom Mutator ohne Probleme verändert werden dürfen. Wird aber ein bereits markiertes Objekt verändert, so muss der Collector von diesem veränderten Objekt aus erneut ein rekursives Mark & Sweep durchführen. Zur Menge der veränderten Objekte

gehören natürlich auch jene Objekte, die neu angelegt wurden. Diese neu angelegten Objekte werden (oder wurden) nämlich von den Wurzelobjekten aus referenziert, nachdem bereits die zweite Phase der größtenteils nebenläufigen Garbage Collection begonnen hatte.

Durch das erneute Überprüfen der veränderten Objekte wird sichergestellt, dass keine Referenzen zu Objekten „verloren gehen“, d.h. dass jedes Objekt markiert wird, bevor die Sweep-Phase beginnt.

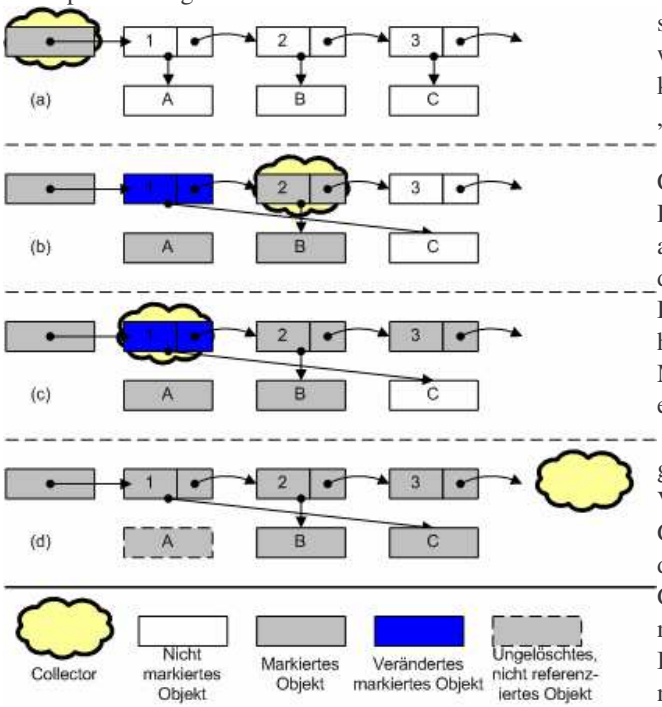

Fig. 2. Mark & Sweep mit parallelem Verändern einer einfach verketteten Liste mit Unterstützung der Plattform

Wie nun ein Collector funktioniert, der veränderte markierte Objekte erneut überprüft, wird in Fig. 2 verdeutlicht. Die Ausgangssituation ist dieselbe wie bei Fig. 1. Im zweiten Schritt (b) teilt aber die Plattform dem Collector mit, dass der bereits markierte erste Knoten der Liste vom Mutator verändert wurde. Nachdem der Collector die Objekte zu Ende durchlaufen hat, startet er erneut bei allen veränderten markierten Objekten, in unserem Fall beim Knoten 1. Der Collector markiert das C Objekt, beim 2. Knoten bricht er ab, da dieser bereits markiert ist (c). Zum Schluss gibt der Collector alle nicht markierten Objekte frei (in unserem Fall gar keines). Wie auch beim vorigen Beispiel, wird das A Objekt, obwohl es nicht mehr länger referenziert wird, vom Collector nicht freigegeben (d). Es bleibt aber durchaus nicht permanent im Speicher, da es beim nächsten Collection-Zyklus mit Sicherheit freigegeben wird (es sei denn, es besteht darauf eine schwache Referenz, die vor dem nächsten Collection Zyklus zu einer starken Referenz wird; solche schwachen Referenzen werden beispielsweise von der Java Plattform unterstützt [6] und stellen einen – in dieser Seminararbeit nicht näher erläuterten – Sonderfall dar).

Generationelle größtenteils nebenläufige Garbage Collection

Garbage Collection, die den gesamten Heap analysiert, wurde von der generationellen Garbage Collection größtenteils abgelöst. Hierbei werden Objekte nach ihrer Lebensdauer unterschieden. Statistisch gesehen haben jüngere Objekte eine kürzere Lebensdauer. Dies lässt sich leicht dadurch erklären, dass Objekte beispielsweise nur während einer Methode existieren.

Bei der generationellen Garbage Collection werden nun Objekte je nach ihrer bisherigen Lebenszeit unterschieden: Jüngere Objekte zählen zur „Jungen Generation“, Objekte, die schon länger existieren und noch immer referenziert werden, werden zur „Älteren Generation“ gezählt. Junge Objekte können nach einer gewissen Zeit auch zur Älteren Generation „befördert“ werden.

Diese Unterscheidung hat den Vorteil, dass man pro Collection-Zyklus nicht den gesamten Heap bereinigen muss. Da jüngere Objekte eine weitaus höhere Wahrscheinlichkeit aufweisen, nicht mehr benötigt zu werden, wird meistens bei den Collection Zyklen nur die Junge Generation überprüft. Die ältere Generation muss nicht so oft überprüft werden, es handelt sich hierbei aber erfahrungsgemäß um eine größere Menge und wenn diese Menge untersucht wird, so gleicht dies einer Bereinigung des gesamten Heaps.

Printezis und Detlefs [2] entwickelten deshalb einen größtenteils nebenläufigen Garbage Collector für die Sun Java Virtual Machine, der eingesetzt wird, wenn die Ältere Generation von Objekten untersucht wird. Da sich die Menge dieser Objekte auch nicht so schnell verändert, wie die Junge Generation, scheint dies ein idealles Einsatzgebiet für den nebenläufigen Garbage Collector zu sein. Wie diese Implementierung eines Collectors funktioniert, wird in den nächsten Kapiteln genauer erörtert.

IV. Algorithmus von Boehm et al.

Implementierung

Hans-J. Boehm, Alan J. Demers und Scott Shenker entwickelten 1991 im Xerox PARC einen generationellen, größtenteils nebenläufigen Garbage Collector für – wie sie es nannten – primitive Sprachen wie C, die keine Zeiger Informationen bieten. Dieser Collector wurde erfolgreich auf einer SparcStation II eingesetzt und getestet.

Da dieser Garbage Collector für Programme, die direkt in Maschinencode übersetzt wurden und auf keiner Virtuellen Maschine liefen, eingesetzt wurde, brauchte man entsprechende Unterstützung von der Hardware. Dazu wurden die Speicherschutz-Funktionen, die von den meisten modernen Prozessoren unterstützt werden, eingesetzt, um festzustellen, ob auf bereits überprüfte Seiten im Speicher schreibend zugegriffen wurde. War dies der Fall, so mussten alle Objekte, die innerhalb dieser Seite im Speicher lagen, neu überprüft werden.

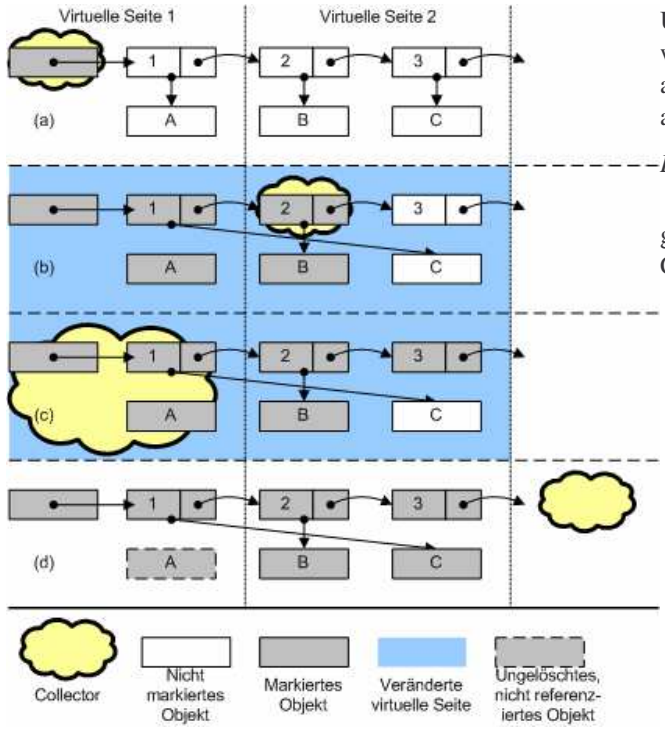

Fig. 3. Beispiel für nebenläufiges Mark & Sweep implementiert mit Seitenschutz

Wie dies in einem Programm auch ohne direkte Unterstützung des Betriebssystem Kernels umgesetzt werden konnte, sollte hier kurz anhand des Beispiels der Intel 386er Architektur veranschaulicht werden. Die Intel i386- (und kompatible) Prozessorfamilie unterstützt Funktionen zur Verwaltung von virtuellem Speicher [7]. Dabei kann der Betriebssystem Kernel (bzw. eine Anwendung über Systemcalls) virtuelle Seiten auf „read-only“ setzen. D.h., dass, wenn auf diese virtuelle Seite schreibend zugegriffen wird, der Prozessor einen Page Fault feststellt und die entsprechende Handler-Routine des Betriebssystems aufruft. Das Betriebssystem leitet diesen Fehler and die Anwendung über ein Signal weiter. In der Implementierung von Boehm et al. wurde jenes Signal abgefangen und festgehalten, das anzeigt, dass eine Seite verändert wurde. Sämtliche Objekte innerhalb einer solchen Seite wurden erneut überprüft.

Fig. 3 sollte dies veranschaulichen. Es handelt sich hierbei wieder um dasselbe Beispiel, wie vorhin benutzt. In diesem Beispiel wird angenommen, dass sich das Kopf-Objekt der Liste, der erste Knoten und das A Objekt in der ersten virtuellen Seite im Speicher befinden; die Knoten 2 und 3, sowie die Objekte B und C befinden sich in der zweiten virtuellen Seite.

In Phase (b) verändert der Mutator wieder die Knoten 1 und 3. Dadurch werden die Seiten 1 und 2 auf „dirty“ gesetzt, d.h., dass der Collector „weiß“, dass Objekte innerhalb dieser Seiten verändert wurden. Welche Objekte dies sind, kann der Collector nicht feststellen und deshalb ist es nötig, sämtliche Objekte innerhalb dieser Speicherseiten neu zu überprüfen. Das bedeutet in unserem Fall, dass sämtliche Objekte neu überprüft werden müssen, sowohl jene in der virtuellen Seite 1 (c), als auch jene in der zweiten virtuellen Seite. Nach dem Überprüfen kann der Collector das „Dirty Bit“ der Seiten wieder löschen und die read-only Attribute der Seiten aufheben (d) (die Dirty-Bits werden dabei im Collector, also auf Softwareebene, verwaltet).

Diskussion

Diese Implementierung hat einige Einschränkungen, die größtenteils darauf zurückzuführen sind, dass die Garbage Collection von keiner virtuellen Maschine unterstützt wird.

- **Großer Overhead**. Wird auf Seiten schreibend zugegriffen und vom Prozessor eine Exception geworfen, so muss ein sogenannter „Context Switch“, also ein Wechsel von der Anwendung zum Betriebssystem erfolgen. Dies kostet viele Taktzyklen und verursacht eine hohe Latenz der Anwendung.
- **Ungenaue Informationen**. Wird an die Anwendung ein Signal geleitet, das anzeigt, dass auf eine Seite schreibend zugegriffen wurde, so kann die Anwendung (und auch nicht das Betriebssystem) nicht feststellen, auf welches Objekt innerhalb der Seite zugegriffen wurde. Es müssen also wieder alle Objekte innerhalb dieser Seite überprüft werden. Eine virtuelle Seite im Anwendungsbereich hat beispielsweise auf einer 32 Bit Intel-Architektur normalerweise eine Größe von 4096 Byte [7]. In einer solchen Seite könnten also im schlimmsten Fall bis zu 1024 32 Bit Zeiger liegen, die überprüft werden müssten.
- **Ungenaue Typinformationen**. Wird auf ein Objekt schreibend zugegriffen, so kann der Collector nicht feststellen, ob ein skalarer Wert oder eine Referenz zu einem anderen Objekt geändert wurde und so müssen oftmals Überprüfungen von Objekten durchgeführt werden, obwohl keine Referenz verändert wurde.
- **Anwendungsspezifisches Tuning**. Der Garbage Collector wird oft dann aufgerufen, wenn der freie Speicher knapp wird. Wenn nun aber der Mutator schneller Speicher reserviert, als der Collector diesen freigeben kann, so passiert es, dass mehr virtueller Speicher alloziert wird als physischer Speicher zur Verfügung steht und so müssen Speicherseiten auf langsamere Medien wie die Festplatte ausgelagert werden. Dies führt zu hohen Verzögerungen der Anwendung. In einigen Tests, die Boehm et al. durchführten, bemerkten sie dieses Phänomen und mussten den Collector für Anwendungen, die schnell viel Speicher reservieren, besonders tunen, beispielsweise indem sie dem Collector-Thread eine höhere Priorität einräumten.

V. Algorithmus von Printezis und Detlefs

Implementierung

Im Jahr 2000 implementierten Tony Printezis und David Detlefs einen generationellen, größtenteils nebenläufigen

Garbage Collector für die „Sun Microsystems Laboratories Virtual Machine for Research“. Diese Virtual Machine bietet eine Schnittstelle an, um (beliebige) Garbage Collectoren einzubinden [8]. Zusätzlich bietet sie das generational Framework, das es ermöglicht, einen generationellen Garbage Collector einzubinden.

Bei dieser Implementierung eines nebenläufigen Garbage Collectors konnte also auf eine gewisse Unterstützung der virtuellen Maschine zurückgegriffen werden. Zufälligerweise konnte außerdem eine bereits implementierte Komponente der generationellen Garbage Collection wiederverwendet werden, nämlich der sogenannte Card Table.

Der Card Table und der – für den größtenteils nebenläufigen Garbage Collector verwendete – Mod Union Table werden im Folgenden anhand eines Beispiels und Figur 4 genauer erklärt. Um bei der generationellen Garbage Collection Referenzen zwischen Objekten der älteren Generationen und Objekten der jüngeren Generationen zu erkennen, muss man Veränderungen zwischen zwei Collection Zyklen festhalten. Dabei werden Updates von Referenzfeldern in einer älteren Generation so verändert, dass dies in einer Tabelle (nämlich der Card Table) festgehalten wird. Ein Update eines Zeigers besteht also nicht mehr aus einer Instruktion, nämlich dem Zuweisen eines Wertes zu einem Speicherbereich, sondern es werden zusätzliche Instruktionen benötigt, um einen Eintrag in der Card Table vorzunehmen. Ein Eintrag im Card Table gibt dabei an, dass eine Referenz innerhalb eines Objekts innerhalb eines gewissen Speicherbereichs (einer „Card“) seit der letzten Collection verändert wurde, diese Card also „dirty“ („schmutzig“) ist. In Fig. 4 wird dies durch die rote Farbe dargestellt: Das erste und das sechste Objekt der älteren Generation wurden verändert und enthalten Zeiger auf Objekte in einer jüngeren Generation. Der Einfachkeit halber wurde in der Abbildung angenommen, dass ein Objekt genau eine Speicherkarte einnimmt, was in der Realität natürlich selten der Fall ist.

Wie bereits vorhin erwähnt, muss der Mutator Code so verändert werden, dass bei einem Schreibzugriff auf einen Zeiger auch der entsprechende Eintrag im Card Table gesetzt wird. Aus Performanzgründen [2] besteht der Card Table aus Bit-Einträgen und nicht aus einem Bit-Array. Zum Setzen dieses Wertes wird ein sogenannter Barrier verwendet. In dieser Implementierung wird ein 2-Befehls Barrier, wie dieser von Urs Hölzle vorgeschlagen wurde [9] verwendet. Eine Zuweisung einer Referenz sieht also im Maschinencode aus wie in Fig. 5 dargestellt. Der Code, der in Fig. 5 gezeigt wird, setzt voraus, dass eine Card die Größe 2^k hat, wie dies bei der Java Virtual Machine der Fall ist (dort ist eine Card 512 Byte groß).

```
; ptr im Feld des Objekts speichern
st    [%obj + offset], %ptr
; den „ungefähren“ Byte-Index berechnen
sll   %obj, k, %temp
; das Byte im Card Table löschen
st    %g0, [%byte_map + %temp]
```

Fig. 5 Maschinencode eines 2-Instruktions Barriers laut Hölzle [10] auf einer SPARC Architektur (Big Endian)

Der Barrier führt also einen bitweisen Leftshift um *k* Bits durch und setzt dann den entsprechenden Wert im Card Table.

Wozu wird aber nun der Mod Union Table benötigt? Der Mod Union Table ist ein Bitvektor, der die Vereinigung der veränderten Karten des Heaps beinhaltet (Union of Modifications). In dem System kann es vorkommen, dass zwei Garbage Collectoren gleichzeitig laufen, nämlich der nebenläufige Garbage Collector, der die ältere Generation bereinigt und ein anderer Garbage Collector, der die junge Generation bereinigt. Der Collector der jungen Generation scant den Card Table der alten Generation nach Veränderungen (also nach Fällen, wo das *dirty* Byte gesetzt ist). Findet er keine neue Referenz zu einem Objekt in der jungen Generation, so ist es nicht mehr nötig, diese Karte in weiteren Durchläufen zu scannen und deshalb setzt er die Karte auf *clean*. Der nebenläufige Collector benötigt allerdings die Information noch, dass die entsprechende Karte verändert wurde und hier kommt der Mod Union Table ins Spiel. Bevor der Collector der jungen Generation den Card Table durchsucht, setzt er die entsprechenden Bits im Mod Union Table. Dadurch kann eine Invarianz zwischen Card Table und Mod Union Table entstehen: Bits, die im Mod Union Table gesetzt sind, müssen nicht (mehr) zwangsläufig im Card Table gesetzt sein (und umgekehrt). Diese Invarianz erfordert, dass der Collector der alten Generation sowohl den Card Table als auch den Mod Union Table auf der Suche nach veränderten Karten traversiert.

Die bei dieser Suche gefundenen veränderten Karten entsprechen nun in etwa den veränderten Seiten bei Boehms

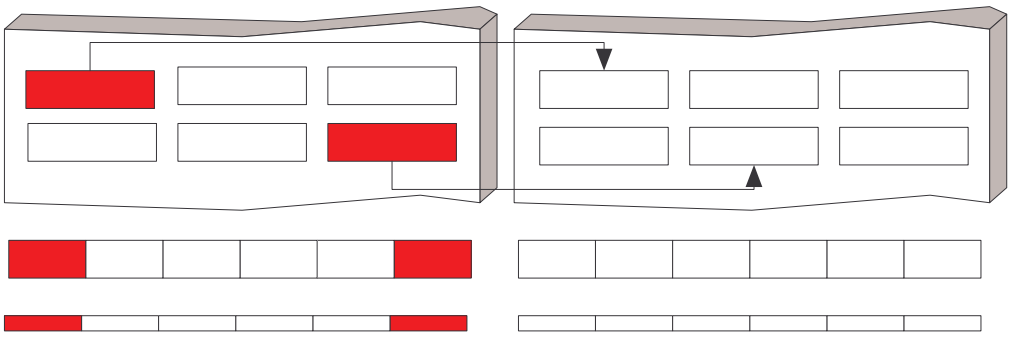

Fig. 4 Der Card Table und der Mod Union Table. Rot markierte Objekte geben an, dass eine Referenz innerhalb einer Speicherkarte verändert wurde.

Algorithmus.

Die **Markierungs-Phase** des Collectors ist so implementiert, dass möglichst wenig Speicher verbraucht wird. Zum Markieren wird ein externer Bitvektor verwendet (d.h. die Objekte werden nicht direkt im Objektheader markiert), wobei ein Bit einem 4-Byte Wort am Heap entspricht. Fig. 5 sollte dies veranschaulichen.

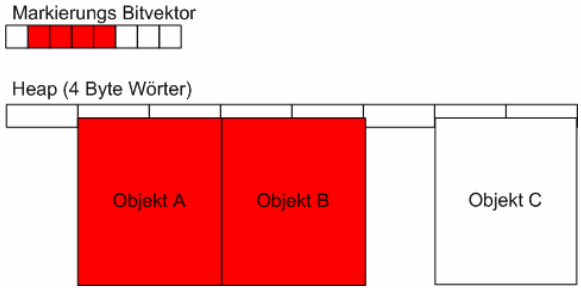

Fig. 5 Ein 32 Byte Ausschnitt des Heaps und der entsprechende Teil des Markierungs Bitvektors

Hier sieht man, wie der Heap in 4 Byte Wörter unterteilt wurde. Es liegen 3 Objekte am Heap, von denen Objekt A und Objekt B bereits markiert wurden. Ihre entsprechenden Einträge im Bitvektor wurden gesetzt. Objekt C wurde nicht markiert, die entsprechenden Bits im Bitvektor sind nicht gesetzt.

Der Scanvorgang nutzt diesen Bitvektor, um den benötigten Speicher zu minimieren. Beim Rootscan werden alle Objekte markiert, die direkt von den Wurzelobjekten aus erreichbar sind. Anschließend erfolgt ein linearer Scan des Heaps (bzw. des Bitvektors). Wird eine Markierung gefunden, so bedeutet das, dass ein markiertes Objekt gefunden wurde, dessen Referenzen auf andere Objekte überprüft werden müssen. Da der Collector linear den ganzen Heap scannt, werden jene Objekte, die sich „nach“ der aktuellen Position im Heap befinden, markiert aber nicht weiter überprüft. Diese Objekte werden sowieso später im Laufe des Heapscans erfasst. Jene Objekte, die sich weiter hinten im Heap befinden, werden einem To-Be-Scanned Stack hinzugefügt. Diese Objekte müssen rekursiv weiter überprüft werden, wobei wieder gilt: Wird ein Objekt gefunden, das sich nach der aktuellen Position am Heap befindet, so wird es markiert, ansonsten muss es gleich überprüft werden.

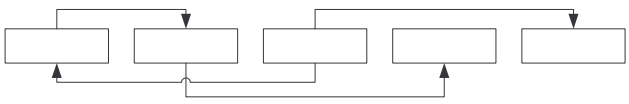

Fig. 6 Beispiel eines Markierungsverlaufs

Fig. 6 zeigt ein Beispiel, wie eine solche Markierung verlaufen könnte. Der Collector ist bei seinem Heapscan beim markierten Objekt c angekommen (*cur*). Dieses Objekt hat zwei Referenzen, nämlich zu Objekt e und Objekt a. Objekt e befindet sich nach cur (Adresse(e) > cur), es wird deshalb im Bitvektor markiert aber nicht gescannt. Objekt a befindet sich vor cur (Adresse(a) < cur), es wird deshalb markiert und auf einen To-Be-Scanned Stack gelegt, der, nachdem alle Referenzen von c verarbeitet wurden, abgearbeitet wird. Der Collector erkennt beim Scan von a, dass dies eine Referenz zu b hat, das wiederum vor cur am Heap liegt. b wird also zum To-Be-Scanned Stack hinzugefügt und anschließend gescannt. Der Collector erkennt wiederum eine Referenz zum Objekt d, das aber nach cur im Speicher liegt und somit markiert aber nicht sofort gescannt wird. Der To-Be-Scanned Stack ist somit abgearbeitet und der Collector fährt mit seinem Scan des Heaps fort (cur wird erhöht). Dort findet er die markierten Objekte d und e, die jeweils keine Referenzen zu anderen Objekten aufweisen. Welche Vor- und Nachteile diese Methode hat, wird im nächsten Kapitel genauer diskutiert.

Für die **Sweep-Phase** dieses nebenläufigen Collectors müssen wieder einige Szenarien in Betracht gezogen werden. Da diese Phase auch ohnen eine globale Stop-the-world-Phase durchgeführt wird, muss sichergestellt werden, dass der Mutator und der Collector die free-lists, also die Listen, die die freien Blöcke verwalten, nicht gleichzeitig verändern. Konkret könnten folgende zwei Szenarien eintreten:

- Ein Mutator könnte gerade einen freien Block aus der free-list löschen, während der Collector einen Block in diese Liste hinzufügen möchte. Dadurch könnten Race-Conditions und in weiterer Folge Inkonsistenzen in den free-lists entstehen.
- Ein Collector könnte gerade mehrere freie Blöcke zu einem größeren zusammenfügen, während ein Mutator-Thread einen dieser freien Blöcke reserviert. Dadurch könnte Speicher doppelt belegt werden.

Deshalb ist es nötig, Locks in den Allokator einzuführen. Ein Mutator wird einen Lock auf eine free-list erhalten müssen, bevor er diese verändert und der Collector wird zusätzlich einen Lock anfordern, wenn er mehrere freie Blöcke zu einem großen zusammenfügen will.

Ein weiterers Feature der Implementierung dieses größtenteils nebenläufigen Garbage Collectors sind die Kontroll-Heuristiken. Ein Problem Boehms Implementierung war, wie bereits in Kapitel IV erwähnt, dass der Mutator manchmal schneller Speicher allokiert hat, als der Collector freigeben konnte. Die Implementierung für die Java VM setzt hier Heuristiken ein, damit es möglichst vermieden wird, dass der Speicher überfüllt wird und Teile ausgelagert werden müssen. Dabei stellt man sich das Allokieren des Mutators und das Freigeben des Collectors wie ein Wettrennen vor: Wenn der Mutator zu weit vorne liegt, muss der Collector mehr „Gas geben“, ist der Mutator gar schon knapp vor der Ziellinie, so muss der Collector das System ganz anhalten und sequentiell weiterarbeiten. Dies konnte nun folgendermaßen implementiert werden: Auf Uniprozessorsystemen werden die Mutator- und Collector-Threads so ausgeführt, dass jeder Thread eine gewisse Zeit lang Rechenzeit erhält. Immer wenn der Collector Rechenzeit erhält, überprüft er, wie viel Speicher allokiert wurde. Je mehr Speicher besetzt ist, desto kürzer werden seine Pausen, er arbeitet also schneller bzw. länger als der Mutator, wodurch er „aufholen“ kann. Ab einem gewissen Grenzwert übernimmt der Collector die gesamte Rechenzeit und gibt seine Rechenzeit erst wieder frei, wenn der Collection-Zyklus komplett abgeschlossen wurde.

Auf Multiprozessorsystemen kann diese Technik nicht

eingesetzt werden. Dort läuft der Collector-Thread typischerweise auf einem eigenen Prozessor und arbeitet dort so schnell er kann. Hier kann es durchaus vorkommen, dass der Collector das Wettrennen gegen den Mutator verliert.

```
initFrac = (1 - heapOccupancyFrac) *
    allocBeforeCycleFrac;
while (TRUE) {
  sleep(SLEEP_INTERVAL);
  if (generationOccupancy() > initFrac) {
    /* 1st stop-the-world phase */
    initialMarkingPause();
    concurrentMarkingPhase();
    concurrentPreCleaningPhase();
    if (markedPercentage() < 98%) {
      /* 2nd stop-the-world phase */
      finalMarkingPause();
      if (markedPercentage() < 98%)
        concurrentSweepingPhase();
    }
  }
}
```

Fig. 7 Pseudocode des Garbage Collectors

Fig. 7 zeigt noch einmal zusammendfassend den Collector-Thread in Pseudocode. Anfangs wird der Wert `initFrac` initialisiert, der bestimmt, ab welcher Heapbelegung eine Garbage Collection durchgeführt werden soll. Dieser Wert kann vom Benutzer mittels eines Kommandozeilenarguments bestimmt werden. Danach startet die Endlosschleife des Collector-Threads. In regelmäßigen Abständen erwacht der Thread und überprüft, ob die Belegung der Generation den geforderten Grenzwert erreicht hat. Ist dies der Fall, so startet die erste Stop-the-world-Phase, in der die Wurzelobjekte markiert werden. Anschließend findet die nebenläufige Markierungs-Phase und danach die PreCleaning-Phase statt. Sind weniger als 98% der Objekte am Heap markiert, so startet die finale Markierungsphase, bei der wiederum alle Threads außer dem Collector-Thread angehalten werden. Nun wird erneut überprüft, ob noch weniger als 98% der Objekte markiert sind. Eine Sweep-Phase würde sich nicht rentieren, wenn dieser Prozentsatz an markierten Objekten überschritten wird. Abschließend findet noch die nebenläufige Sweep-Phase statt, bei der die Objekte freigegeben werden.

Diskussion

Diese Implementierung hat einige Vor- und Nachteile, die zum Teil bereits erwähnt wurden. Zuerst zu den Vorteilen:

- **Weniger Overhead**. Da in das System kaum Eingriffe gemacht wurden, um den Garbage Collector der alten Generation größtenteils nebenläufig zu machen, gibt es keine großen Laufzeit-Overheads im Vergleich zum sequentiellen Collector.
- **Feinere Unterteilung der Art der Speicherzugriffe**. Bei dieser Implementierung wird nur dann der Card Table verändert, wenn eine Referenz geschrieben wird. Dadurch muss nicht auch bei skalaren Zugriffen der normale Programmfluss unterbrochen werden.
- **Feinere Granularität**. Die Kartengröße kann je nach Bedarf eingestellt werden. Dadurch müssen nicht so große Datenbereiche wie eine gesamte Speicherseite nach einem Schreibzugriff neu untersucht werden.
- **Nebenläufigkeit zu anderen Garbage Collectoren**. Tatsächlich laufen verschiedene Zyklen verschiedener Garbage Collectoren gleichzeitig ab. Während einer Sammlung der jungen Generation wird allerdings der nebenläufige Garbage Collector auch komplett angehalten, um Synchronisierungsoverhead klein zu halten und damit diese Einsammlungen der jungen Generation möglichst schnell ablaufen können.
- **Kontrollheuristiken**. Anders als die Implementierung von Boehm et al. kann sich dieser Algorithmus selbst regeln und es bedarf keiner Anpassung an die jeweilige Anwendung. Zusätzlich kann man durch bestimmte Argumente an die Java VM auch applikationsspezifische Optimierungen vornehmen, so kann man beispielsweise den Grenzwert der Speicherbelegung einstellen, bei dem der Collector-Thread anfangen sollte zu arbeiten.

Natürlich hat aber auch diese Implementierung einige Nachteile:

- **Speicheroverhead**. Der Mod Union Table ist ein Bitvektor, der die Speicherkarten vereinigt. Da Speicherkarten überlicherweise eine Größe von 512 Byte haben und eine solche Karte auf ein Bit abgebildet wird, beträgt dieser Overhead nur ca. 0.01%. Da aber ein externer Bitvektor zum Markieren der Objekte verwendet wird (die Objektheader können aufgrund der Nebenläufigkeit der Mutator-Threads nicht zum Markieren verwendet werden), muss beim Markieren ein Speicheroverhead von 3,125% für die alte Generation toleriert werden. Ein zusätzliches Problem stellt hier der To-be-scanned-Stack dar. Da keine speicherschonende Technik wie die Zeigerumkehr [3] verwendet werden kann, um die Referenzen bei der Mark-Phase zu halten, da der Mutator die Zeiger benötigt und man sonst einen teuren Read-Barrier einsetzen müsste, benötigt man diese Datenstruktur, die unter gewissen Umständen relativ groß werden kann. Dies ist besonders schlecht, da ein Collection-Zyklus im Normalfall nur dann starten, wenn der Speicher schon erschöpft ist.
- **Lineare Heapsuche**. Um den To-Be-Scanned Stack möglichst klein zu halten, wird eine lineare Suche über den gesamten Heap (zumindest der alten Generation) durchgeführt. Dies hat eine relativ schlechte Laufzeitkomplexität, vor allem, wenn der Heap nicht dicht belegt ist.

VI. Benchmarks

Die Benchmarks, die von Boehm et al. [1] in ihrer Arbeit durchgeführt wurden haben widersinnige Ergebnisse, die auf heutiger Hardware nicht mehr nachvollzogen werden können. So benötigte beispielsweise ein von ihnen zum Vergleich eingesetzter Stop-the-world Garbage Collector für eine

Bereinigung eines 10MB Heaps durchschnittlich 46 Sekunden, während ihr generationeller, nebenläufiger Collector nur 102ms benötigte. Unerklärlich ist auch, warum die gesamte Ausführungszeit der Programme so stark schwankte: So benötigte bei einem Benchmark der schlechteste Collector mit drei Sammlungen á 46 Sekunden für das ganze Programm 332,2 Sekunden, rechnete also abgesehen von den Collection-Zyklen 194 Sekunden lang. Der Benchmark, der mit ihrem Collector ausgestattet war, rechnete insgesamt aber nur 32 Sekunden. Somit ist davon auszugehen, dass es sich hier nicht um objektive Vergleiche verschiedener Collectoren handelte.

Aus diesem Grund wird sich diese Seminararbeit auf die Benchmarks, die von Printezis und Detlefs [2] durchgeführt wurden, beschränken. Diese sind auch aus heutiger Sicht interessanter, da sie auf aktuellerer Hardware durchgeführt wurden.

Der größtenteils nebenläufige Garbage Collector der Java Virtual Machine wurde auf einem Sun E3500 Server mit 8 336Mhz Prozessoren mit gemeinsamem Speicher getestet. Der Collector konnte natürlich von der Multiprozessorarchitektur profitieren, da der Collecor Thread auf einem eigenen Prozessor laufen konnte.

Zum Testen wurde eine eigene Anwendung namens „gcold“ implementiert, die verschiedene Lasten für den Garbage Collector simulieren konnte. Diese Anwendung allokiert ein Array von Zeigern auf binäre Suchbäume und erstellt dann mehrere solcher Bäume. Während des Tests werden kleinere Objekte erstellt, die bald zu Garbage werden und es werden Teile der binären Bäume ausgetauscht, verändert usw. Wie viele junge Objekte erstellt werden und wie oft die alten Binärbäume ersetzt werden, lässt sich durch Kommandozeilenargumente festlegen und es wurden verschiedene Konfigurationen getestet. Besonders wichtig zum Testen dieses Collectors ist natürlich die Rate der Veränderungen der Zeiger in den Objekten der alten Generation, da bei diesen Veränderungen die Cards beschrieben werden und der Collector diese somit neu auswerten muss. Deshalb wurden während eines Arbeitsdurchlaufs dieses Programms auch solche Zeiger derart verändert, dass die Anzahl der Objekte (und die Objekte selbst) die gleichen blieben, die Zeiger aber verändert wurden (z.B. indem sie untereinander ausgetauscht wurden).

Tabelle 1 zeigt die Ergebnisse der Tests, die mit dem größtenteils nebenläufigen Garbage Collector ausgeführt wurden im Vergleich zu den Ergebnissen mit dem standardmäßigen, generationellen Collector.

Diese Tabelle zeigt einige sehr interessante Ergebnisse. So ist auffallend, dass der nebenläufige Collector stets eine größere Heapgröße benötigt. Dies kann damit erklärt werden, dass der Collector zum einen einen gewissen Speicheroverhead verursacht, zum anderen das Rennen um

old-gen collector	live data *(Mbyte)*	elapsed time *(sec)*	max heap *(Mbyte)*	young-gen pauses avg *(ms)*	young-gen pauses max *(ms)*	young-gen pauses total *(sec)*	old-gen pauses avg *(ms)*	old-gen pauses max *(ms)*	old-gen pauses total *(sec)*
default	50	370	69	18	39	51.1	1298	1959	42.8
mostly-concurrent	50	334	93	24	70	69.8	14	39	1.2
default	100	351	189	19	36	54.1	2593	3491	20.7
mostly-concurrent	100	342	189	26	178	76.5	23	57	0.9
default	150	364	252	19	58	56.1	3985	6274	31.9
mostly-concurrent	150	347	286	28	361	81.7	27	67	0.7
default	200	363	315	20	42	57.5	4981	6763	29.9
mostly-concurrent	200	349	369	29	146	84.6	32	69	0.6
default	250	370	415	21	38	59.6	6944	10368	34.7
mostly-concurrent	250	356	498	31	145	91.1	41	105	0.6
default	300	362	500	21	39	61.8	7900	9938	23.7
mostly-concurrent	300	382	566	35	136	102.5	44	112	0.6

Tab. 1 gcold Benchmark des standardmäßigen generationellen Collectors gegen den größtenteils nebenläufigen.

das Allokieren und Freigeben von Daten gegen den Mutator manchmal verliert, wodurch mehr maximaler Heap angefordert werden muss.

Des weiteren zeigt sich, dass der standardmäßige Collector wie erwartet kürzere Pausen bei den Scans der jungen Generation aufweist. Dies ist leicht damit zu erklären, dass bei dem nebenläufigen Collector bei jeder Sammlung der jungen Objekte der Mod Union Table beschrieben werden muss, was laut diesen Ergebnissen einen großen Overhead verursacht.

Zuletzt zeigen sich die Pausen, die bei einer Sammlung der Objekte der alten Generation entstehen. Wie erwartet sind die Stop-the-world-Phasen hier beim größtenteils nebenläufigen Collector um ein vielfaches geringer. Bei einer Datenmenge von 300 MB benötigt ein Scan der Wurzelelemente etwa durchschnittlich nur 44ms, während ein kompletter Scan der alten Generation 7,9 Sekunden benötigt.

Es wurde zusätzlich zu diesem synthetischen Benchmark auch ein Benchmark mit realen Anwendungen durchgeführt. Zu diesen zählte unter anderem auch ein Test, bei dem 2740 Quelldateien mit dem Java Compiler übersetzt wurden. Es zeigte sich, dass beim nebenläufigen Collector zwar die Pausen bei den Einsammlungen bei der alten Objekt-Generation deutlich geringer waren, durch den überwiegenden Anteil an junge-Generation-Einsammlungen aber die

Gesamtlaufzeit mit dem herkömmlichen generationellen Collector niedriger blieb.

Ein weiterer Benchmark, der durchgeführt wurde, maß die Auswirkungen der Promotions-Rate eines Programms, also wie oft Objekte aus der jungen Generation in die neue „befördert“ wurden. Der Test wurde mit einer konstanten Menge an allokierten Objekten im Heap durchgeführt (200 MB).

promotion rate *(Mbytes/sec)*	maximum heap size *(Mbytes)*	old-gen pauses avg *(ms)*	old-gen pauses max *(ms)*
0.76	369	13	29
1.38	369	17	38
2.29	369	35	78
3.40	369	82	204
3.77	602	4199	10637
3.69	602	9602	24147

Tab. 2 Auswirkung der Promotions-Rate auf Collector Pausen

Tabelle 2 zeigt die Ergebnisse dieses Benchmarks. Es ist auffällig, dass ab einer gewissen Promotions-Rate ein großer Sprung der Pausen bei den Einsammlungen in der alten Generation festzustellen ist. Bei einer Promotions-Rate ab etwa 3,77 MB/sec wird auch ein höherer Speicherverbrauch gemessen. Der Grund hierfür scheint einleuchtend: Da die Tests auf einer Multiprozessorarchitektur durchgeführt wurden, konnte der Collectorthread den Mutatorthread nicht steuern. Beide arbeiteten einfach so schnell sie konnten. Scheinbar reichte dies ab einer gewissen Rate der Beförderungen von Objekten nicht mehr aus, um das „Wettrennen“ gegen den Mutator zu gewinnen und es konnten Objekte nicht mehr schneller freigegeben werden als allokiert. Dadurch war es nötig, dass zusätzlicher Arbeitsspeicher angefordert wurde, was zu diesen hohen Pausen führte.

Der größtenteils nebenläufige Collector kam auch bei einer Anwendung einer Telekommunikationsfirma zum Einsatz. Diese Anwendung hatte bei normalem Workload in der älteren Generation mehrere Hundert Megabytes an Daten im Arbeitsspeicher. Da der Kunde aber keine längeren Wartezeiten als 1 Sekunde akzeptierte, konnte kein Collector eingesetzt werden, der Pausen länger als 1 Sekunde für eine Einsammlung in der alten Generation verursachte. Diese Anforderung konnte mit dem größtenteils nebenläufigen Garbage Collector erfüllt werden und er fand seinen ersten Einsatz in dieser Anwendung.

VII. Zusammenfassung und Aussicht

Unter der größtenteils nebenläufigen Garbage Collection versteht man Methoden der Garbage Collection, die es ermöglichen, dass ein Collector Thread nebenläufig zu den Mutator Threads ausgeführt wird. Dies führt dazu, dass sogenannte Stop-the-world-Phasen, die das ganze System anhalten, minimiert werden. Das bewirkt in weiterer Folge, dass Latenzen von Anwendungen, die Garbage Collectoren einsetzen, verringert werden können.

Einsatzgebiete für solche Collectoren sind also Anwendungen, die große Latenzzeiten nicht akzeptieren können. Zu diesen zählen:

- Echtzeitsysteme, die harte oder weiche Zeitschranken einhalten müssen,.
- Serveranwendungen, die mit sehr großen Datenmengen umgehen müssen und deshalb bei einer vollständigen Einsammlung sehr hohe Pausen haben würden.
- Desktopanwendungen, bei denen auch kleine Wartezeiten den Benutzer frustrieren können.

Ein größtenteils paralleler Garbage Collector benötigt, um Veränderungen von Objekten, die bereits gescanned wurden erfassen zu können, Unterstützung von der Plattform, auf der er rennt. Diese Unterstützung muss in Form einer Write-Barrier gegeben sein, es muss also kontrolleirt werden, ob auf bereits markierte Objekte schreibend zugegriffen wird.

Dieser Barrier kann entweder (in unkooperativen Systemen) mit Hilfe der Hardware in Form von Speicherschutzfunktionen des virtuellen Speichers zur Verfügung gestellt werden, oder von Seiten der Software, wenn das Programm auf einer virtuellen Maschine wie der Java Runtime Environment läuft. Dabei hat die Softwarevariante in mehreren Bereichen Vorteile, wie etwa bei der Performanz, der Portabilität auf verschiedene Plattformen und der Granularität der Informationen, was sich wieder auf die Performanz auswirkt. Außerdem kann eine Softwarevariante, die als Standard einen generationellen Collector verwendet leicht so angepasst werden, dass der Collector der alten Generation nebenläufig zum Mutator abläuft, da ein generationeller Collector per se Datenstrukturen und Barrier verlangt, die für den größtenteils nebenläufigen Collector meist ohne große Veränderungen benutzt werden können.

Benchmarks zeigten, dass der größte Vorteil von größtenteils nebenläufiger Garbage Collection vor allem auf Multiprozessorsystemen besteht und für Anwendungen, die eine geringe Promotions-Rate von Objekten von einer jungen Generation zu einer alten Generation aufweisen.

Seit Version 1.4 der Java Runtime Environment ist der größtenteils nebenläufige Garbage Collector ein fixer Bestandteil der Sun Virtual Machine. Er kann über das Kommandozeilenargument `-XX:+UseConcMarkSweepGC` aktiviert werden [5].

Es gibt noch keinen ähnlichen Algorithmus für das Microsoft .NET Framework 2.0, das andere Methoden für die Garbage Collection Synchronisation, wie beispielsweise Method Hijacking einsetzt [10], auf das hier nicht näher eingegangen wird. Es wäre also durchaus denkbar, dass ein größtenteils nebenläufiger Garbage Collector auch ins .NET Framework bald Einzug finden wird.

Für zukünftige Forschungsarbeiten gibt es auf diesem Gebiet auch noch genug Bereiche:

- Eine Parallelisierung der individuellen Markierungs- und Sweep-Phasen
- Eine Überlappung der Sweeping-Phase eines vorhergehenden Collection Zyklus mit der Markierungs-Phase des nächsten.

VIII. Literatur

[1] H. Boehm, A. J. Demers, S. Shenker. Mostly Parallel Garbage Collection. ACM PLDI 91, SIGPLAN Notices 26, 6 (June 1991), pp. 157-164.

[2] Printezis T., Detlefs D.: A Generational Mostly-Concurrent Garbage Collector. Sun Labs TR-2000-88

[3] R. Jones, Rafael Lins. Garbage Collection. August 1996. Wiley.

[4] The Real-Time for Java Expert Group. The Real-Time Specification for Java. Java Community Process. June 2000. Addison-Wesley.

[5] Tuning Garbage Collection with the 5.0 Java Virtual Machine. Chapter 5.4 “The Concurrent Low Pause Collector”
http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html#0.0.0.0.%20Concurrent%20Phases%7Coutline (07.01.2005)

[6] Java Package java.lang.ref.
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ref/package-summary.html (07.01.2005)

[7] Intel Corporation. IA-32 Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide. 2004.
http://www.intel.com/design/pentium4/manuals/index_new.htm (07.01.2005)

[8] D. White, A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc., 1999

[9] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. ACM OOPSLA ’93 Workshop on Memory Management and Garbage Collection, Washington, DC, October 1993

[10] Jeffrey Richter. Garbage Collection—Part 2: Automatic Memory Management in the Microsoft .NET Framework. MSDN Library. Microsoft Corporation. 2003.
http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/default.aspx (08.01.2005)