Der Java Garbage Collector: Funktionsweise und Optimierung

Mathias Dolag, Prof. Dr. Peter Mandl, Christoph Pohl
iSYS Software GmbH, Hochschule München
München

Schlüsselworte
Garbage Collection, GC, Garbage Collector, Java, Mark-and-Copy, Concurrent Mark-and-Sweep, CMS, Garbage First, G1, Garbage Collection Tuning,

Einleitung
Ein oft wenig beachteter, jedoch essentieller Bestandteil der Java Virtual Machine (JVM) ist der *Garbage Collector*. Er ist für das Entfernen nicht mehr benötigter Objekte aus dem Heap-Speicher (Heap) zuständig. Dem Entwickler wird dadurch diese fehleranfällige Arbeit abgenommen, sodass sich dieser auf sein eigentliches Spezialgebiet konzentrieren kann. Doch wie arbeitet der Garbage Collector? Welche Möglichkeiten bieten sich dem Entwickler bzw. Administrator Einfluss auf ihn zu nehmen? Diese Fragen wurden im Rahmen einer Untersuchung an der Hochschule München in Kooperation mit der iSYS Software GmbH untersucht. In diesem Artikel werden die Ergebnisse der Untersuchung vorgestellt. Im ersten Teil wird auf die Funktionsweise ausgewählter Garbage Collector - Algorithmen der HotSpot™ JVM eingegangen. Der zweite Teil beschreibt die grundlegenden Aspekte des Garbage Collection (GC)-Tunings. Im abschließenden Teil werden Ergebnisse durchgeführter Leistungs-Messungen vorgestellt.
Alle dargestellten Informationen und Ergebnisse beziehen sich, sofern nicht explizit anders ausgewiesen, auf die HotSpot™ JVM Version 6.

Allgemeines
Alle Objekte werden bei Java vom Allokator im Heap erzeugt. Die Größe des Heaps kann beim Starten der JVM durch die Parameter -xms (initiale Heap-Größe) und -xmx (maximale Heap-Größe) gesteuert werden. Werden diese Parameter mit identischen Werten gesetzt, so wird bereits beim Start der JVM die gesamte Kapazität reserviert. Andernfalls wir der Heap bei Bedarf dynamisch vergrößert und verkleinert. Es ist festzuhalten, dass es bei Java nicht möglich ist, direkt Heap-Adressen anzusprechen oder Objekte an einer fest definierten Speicheradresse anzulegen. Daher kann ein Objekt im Heap nur erreicht werden, falls darauf eine Referenz existiert. Indirekt muss es dabei von einer *Root-Referenz* aus erreichbar sein. Root-Referenzen sind Referenzen, die beispielsweise von den einzelnen Applikations-Threads in deren Thread-Stacks gehalten werden. Jede Root-Referenz dient dabei als Einstiegspunkt in den Objektgraphen, welcher das Gerüst aller erreichbaren Objekte darstellt. Bei der Summe aller Root-Referenzen spricht man von einem *RootSet*. Durch Verfolgen aller Objektreferenzen (ausgehend von den Root-Referenzen) können vom Garbage Collector bzw. der Applikation alle lebendigen Objekte erreicht werden. Alle nicht erreichbaren Objekte stellen im Sinne der Applikation „Müll“ (engl. Garbage) dar und können vom Garbage Collector entfernt werden, sodass deren Speicherbereich zur Allokation neuer Objekte wieder zur Verfügung steht. Man spricht bei solchen Objekten auch von „toten“ Objekten. Würden die, nicht mehr von einer Root-Referenz aus erreichbaren Objekte, nicht aus dem Heap entfernt, könnte dies irgendwann zu einer *OutOfMemory-Exception* führen, da keine neuen Objekte im Heap allokiert werden können. Vorausgesetzt ist hierbei eine aktive Verwendung der Applikation.
In den folgenden Abschnitten werden die in der JVM zur Verfügung stehenden GC-Algorithmen näher vorgestellt.

Mark-and-Sweep
Nachfolgend wird die Vorgehensweise von *Mark-and-Sweep*, dem einfachsten GC-Algorithmus, beschrieben. Wie dem Namen zu entnehmen ist, läuft dieser in zwei Phasen ab. Phase 1 geht den Objektgraphen durch und markiert alle erreichten Objekte. Dies kann auch parallel durch mehrere Threads durchgeführt werden. Jedem Thread wird dabei eine Root-Referenz zugewiesen, deren Referenzen durch ihn verfolgt werden. Hierdurch kann die Markierungs-Zeit verkürzt werden.
In Phase 2 werden anschließend alle nicht markierten Objekte aus dem Heap entfernt und die freigegebenen Speicherbereiche einer Liste des Allokators (FreeList) angehängt. In dieser Liste verwaltet der Allokator alle freien Speicherbereiche sowie deren Größe. Für die Allokation eines neuen Objekts wird dann ein passender Speicherbereich in der FreeList gesucht.
Mark-and-Sweep fällt in die Kategorie der blockierenden Algorithmen, da während der Markierung der bzw. die GC-Thread(s) uneingeschränkten Zugriff auf den Heap benötigen. Die eigentlichen Applikations-Threads werden dabei unterbrochen. Man spricht bei der entstehenden Pausenzeit auch von einer *Stop-the-World-Pause*.
Durch das reine Entfernen von Objekten wird der Heap auf Dauer unterschiedlich stark fragmentiert. Man spricht folglich von *Heap-Fragmentierung*. Durch diese wird die Allokation neuer Objekte erschwert und verlangsamt, da für jedes neue Objekt eine passende Position im Heap gefunden werden muss. Daher stellt zunehmende Heap-Fragmentierung ein Problem dar. Um diese zu vermeiden bzw. möglichst gering zu halten, wurden komplexere Algorithmen und Konzepte entwickelt, welche nachfolgend besprochen werden.

Generational Garbage Collection
Alle heutigen GC-Algorithmen basieren auf Generationen bzw. Regionen. Man spricht daher von der *Generational Garbage Collection*. Dieser Ansatz resultiert aus der Tatsache, dass Objekte unterschiedliche Lebensdauern haben. Es existieren in einer Java-Applikation dabei meist sehr viele kurzlebige Objekte (z. B. Objekte, welche nur innerhalb einer Methode oder Expression zugreifbar sind, wie z. B. Iteratoren oder StringBuilder), weniger mittellang lebende (z. B. Session-spezifische Variablen) und noch weniger langlebige Objekte (z. B. Singletons oder Datenbank-Verbindungen). Dies wurde nach [2, S. 6] auch durch die *weak generational hypothesis* wissenschaftlich belegt. Diese Hypothese ist dabei unabhängig von einer bestimmten Programmiersprache zu sehen. Ebenfalls wurde festgestellt, dass ältere Objekte nur selten Referenzen auf jüngere Objekte halten.

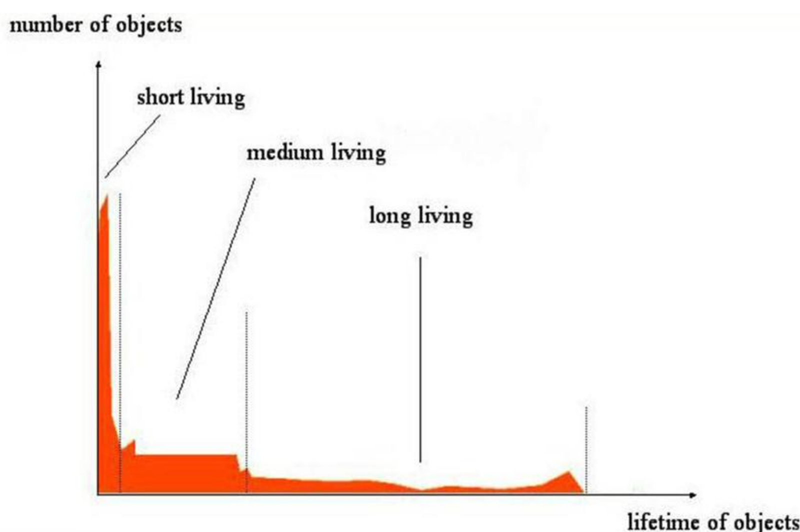

Abbildung 1: Typische Objektpopulation einer Java-Applikation (Quelle: [1, S. 151])

Abbildung 1 zeigt die typische Objektpopulation einer Java-Applikation mit einer Verteilung der Objekte nach deren Lebensdauer. Daraus lässt sich ableiten, dass es nicht vorteilhaft ist, bei einer GC jeweils den gesamten Heap nach nicht mehr referenzierten Objekten zu durchsuchen. Dies würde mit zunehmender Heap-Größe zu immer längeren GC-Pausenzeiten führen und die eigentliche Applikation dadurch massiv behindern.
Der Heap wird in der HotSpot™ JVM daher in Generationen eingeteilt. Junge bzw. neu allokierte Objekte sind in der *Young Generation* gespeichert, ältere Objekte in der *Old Generation*. Die Young Generation ist dabei noch feiner, in *Eden* und zwei gleich große *Survivor Spaces* namens *From-Space* bzw. *To-Space*, unterteilt. Diese spielen eine entscheidende Rolle zur Steigerung der GC-Performance (siehe hierzu Kapitel „Mark-and-Copy"). Daneben existiert noch die *Perm(anent) Generation*, welche u.a. die geladenen Class-Files beinhaltet. Sie spielt für die GC jedoch nur eine untergeordnete Rolle, auf die in diesem Artikel nicht weiter eingegangen wird. Die Aufteilung des Heaps in der HotSpot™ JVM ist in Abbildung 2 grafisch veranschaulicht.

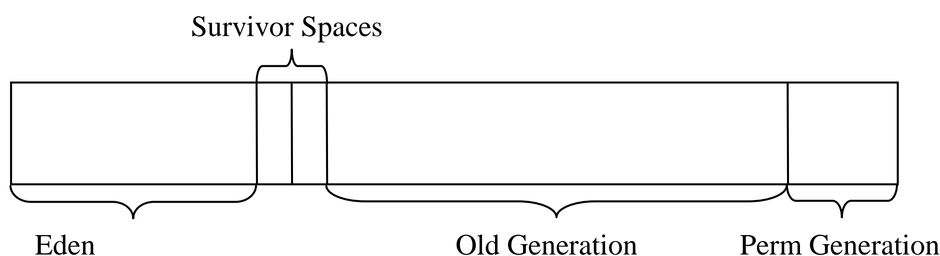

Abbildung 2: JVM-Heap-Aufteilung (Quelle: in Anlehnung an [1, S. 153])

Alle neuen Objekte werden in Eden allokiert, wohingegen die Survivor-Spaces nur Objekte beinhalten, welche bereits mindestens einen GC-Durchlauf überlebt haben. Eine GC wird ausgelöst, sobald Eden keine neuen Objekte mehr aufnehmen kann bzw. sobald die Old Generation einen festgelegten Füllgrad erreicht hat. Dieser Füllgrad kann entweder dynamisch durch die JVM bestimmt, oder manuell durch einen JVM-Parameter festgelegt werden.
Eine GC kann nun entweder nur die Young Generation (*Minor Collection*) oder die Young- und die Old Generation (*Major Collection*) bereinigen. Major Collections sind während der Laufzeit einer Applikation deutlich seltener und bereinigen meist auch nicht mehr Speicherplatz als eine reine Minor Collection. Dabei haben Major Collections jedoch meist eine deutlich längere Laufzeit, da der zu bereinigende Bereich wesentlich größer ist.
Im Folgenden werden unterschiedliche Collection-Algorithmen für die Young- bzw. Old Generation betrachtet. Begonnen wird mit einem Young Generation - Algorithmus.

Mark-and-Copy
Ein rein für die Young Generation verwendbarer Algorithmus ist *Mark-and-Copy*. Er ist aktuell auch der Standard-Algorithmus für die Young Generation. Mark-and-Copy läuft in zwei Phasen ab, einer Markierungs-Phase und einer Kopier-Phase. Die Markierung läuft wie bereits im Kapitel „Mark-and-Sweep" beschrieben ab. Es werden dabei jedoch nur Root-Referenzen, welche auf Objekte in der Young Generation verweisen, verfolgt. Beim Kopieren spielen nun die Survivor-Spaces eine entscheidende Rolle. Zu Beginn einer GC ist der To-Space leer, wohingegen der From-Space Objekte aus früheren GCs enthalten kann. Alle überlebenden Objekte aus Eden und dem From-Space werden nun in den To-Space kopiert. Für den Fall, dass der To-Space nicht alle Objekte aufnehmen kann, werden direkt alle überlebenden Objekte in die Old Generation verschoben. Eine Ausnahme existiert hierbei für Objekte, welche bereits eine festgelegte Anzahl an GCs überlebt haben. Diese werden in die Old Generation verschoben. Diesen Vorgang nennt man *Promotion* oder *Tenuring*. Der Schwellwert, nach wie vielen überlebten GC-Durchläufen ein Objekt verschoben wird, wird als

Tenuring Threshold bezeichnet. Dieser kann entweder über einen JVM-Parameter eingestellt, oder automatisch durch die JVM gesteuert werden. Nach Abschluss des Kopiervorgangs wird der nun leere From-Space zum To-Space für die nächste GC und der To-Space zum From-Space.
Der Kopiervorgang ist in Abbildung 3 nochmals verdeutlicht. Die grau hinterlegten Objekte sind nicht mehr über eine Root-Referenz erreichbar und werden während des GC-Durchlaufs entfernt. Objekt Nr. 6 hat den Tenuring Threshold erreicht und wird nicht in den To-Space, sondern in die Old Generation verschoben.

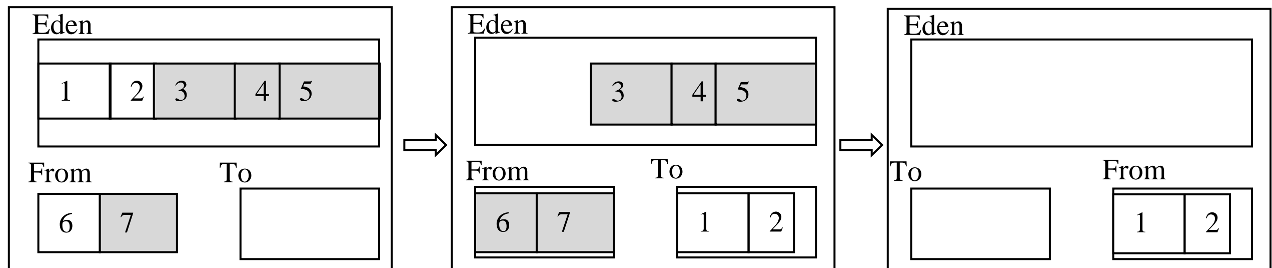

Abbildung 3: Mark-And-Copy (Quelle: In Anlehnung an [8])

Während der Markierung muss bei einer Minor Collection noch ein Sonderfall beachtet werden - *Intergenerational References*. Dies sind Referenzen von Old Generation - Objekten auf Young Generation - Objekte, welche dadurch ebenfalls lebendig gehalten werden können.
Allgemein kann man festhalten, dass Mark-and-Copy ein effizienter Algorithmus ist. Man spricht bei diesem Garbage Collector auch vom *Throughput Collector*, da er für das Erreichen eines hohen Durchsatzes verwendet wird. Das Ziel der Durchsatzoptimierung wird weiter unten erläutert. Da das Verschieben von Objekten nicht ohne Anhalten der Applikations-Threads durchgeführt werden kann, entstehen auch durch Mark-and-Copy Stop-the-World-Pausenzeiten. Diese halten sich auf Grund der geringen Größe der Young Generation jedoch in Grenzen.
Um die Effizienz des Algorithmus weiter zu steigern, kann auch eine parallelisierte Variante von Mark-and-Copy verwendet werden. Hierbei wird das Kopieren parallel durch mehrere GC-Threads durchgeführt. Dazu wird der To-Space in sogenannte *Parallel Local Allocation Buffers (PLAB)* aufgeteilt. Jeder Thread bekommt einen spezifischen PLAB zugewiesen, in den er (ohne Synchronisation mit den anderen Threads) die überlebenden Objekte verschieben darf. Ist ein PLAB voll, so bekommt der Thread einen neuen zugewiesen. Ein gewisses Maß an Fragmentierung ist hierbei nicht zu vermeiden. Dies wird allerdings durch die kürzer ausfallenden Stop-the-World-Pausen in Kauf genommen.

Lessons learned: Wie dargestellt ist Mark-and-Copy ein effizienter Algorithmus zur Erreichung eines hohen Durchsatz. Ebenfalls werden nur geringe Stop-the-World-Pausen verursacht. Er eignet sich daher besonders für interaktive Systeme wie z. B. Webshops. Für die Young Generation ist er aktuell der meist verwendete Collector.

Concurrent-Mark-and-Sweep
Wie bereits beschrieben können durch Major Collections lange Pausenzeiten entstehen. Grund hierfür ist die Größe der Old Generation. Es wurde daher für die Old Generation ein Algorithmus entwickelt, welcher nebenläufig (concurrent) zur eigentlichen Applikation ausgeführt werden kann - der *Concurrent-Mark-and-Sweep (CMS)*. Wie dem Namen zu entnehmen ist, handelt es sich dabei jedoch nur um einen Sweep-Algorithmus, bei dem sich das Problem der Heap-Fragmentierung nicht vermeiden lässt. Da Objekte, sobald sie einmal die Old Generation erreicht haben jedoch selten ihre

Verbindung zu einer Root-Referenz verlieren, hält sich dieses Problem in Grenzen und kann auf Grund des Performance-Vorteils toleriert werden.
Zu Beginn des CMS werden in einer kurzen Stop-the-World-Pause alle Old Generation-Objekte bestimmt, welche von den Root-Referenzen erreicht werden können. Ausgehend von diesen Objekten werden dann nebenläufig zur Applikation alle lebendigen Objekte im Heap bestimmt. Da die Applikation nebenbei jedoch weiter auf dem Heap arbeiten kann, ist nicht sichergestellt, dass alle als lebendig markierten Objekte nach Abschluss der nebenläufigen Markierung auch tatsächlich noch lebendig sind. Eine ehemals gesetzte Referenz kann nach der Objekt-Markierung entfernt worden sein. In diesem Fall spricht man von *Floating Garbage*. Dieser hält sich meist in Grenzen und stellt kein weiteres Problem dar, da er einfach bei der nächsten GC entfernt wird. Ein größeres Problem jedoch stellt die Allokation von Objekten in einem bereits gescannten Heap-Bereich dar. Diese Objekte müssen zwangsläufig als lebendig erkannt werden, sofern sie über eine Root-Referenz erreichbar sind. Daher wird nach der nebenläufigen Markierung eine nicht nebenläufige *Re-Markierung* durchgeführt. In dieser werden alle vom Allokator als geändert (dirty) markierten Bereiche erneut auf lebendige Objekte gescannt. Nach Abschluss der Re-Markierung sind dem Collector definitiv alle lebendigen Objekte bekannt. Diese können dann in der abschließenden, nebenläufig ausgeführten, Sweep-Phase entfernt werden. Um die Pause, welche durch die nicht nebenläufige Re-Markierung entsteht, zu verkürzen kann optional zwischen der nebenläufigen Markierung und der Re-Markierung eine Zusatzphase durchgeführt werden. Diese wird als *Preclean(ing)* bezeichnet. In dieser werden nebenläufig die als dirty markierten Heap-Bereiche auf lebendige Objekte gescannt. Diese Zusatzphase hat dabei eine konfigurierte maximale Laufzeit (Standardwert 5000 ms) und kann auch iterativ durchgeführt werden.
Wie alle bisherigen GC-Algorithmen kann auch der CMS parallel durch mehrere Threads ausgeführt werden. Die Besonderheit hierbei ist, dass nur nicht nebenläufige Phasen durch mehrere Threads abgearbeitet werden. Um die Applikation während den nebenläufigen Phasen möglichst wenig zu beeinträchtigen, werden diese nur von einem einzelnen Collection-Thread (auch als *Reaper Thread* bezeichnet) bearbeitet.
Auf Grund der Nebenläufigkeit wird der GC-Durchlauf des CMS bereits gestartet, sobald ein definierter Füllgrad der Old Generation vorliegt. Man bezeichnet diesen auch als *OccupancyFraction*. Dies soll sicherstellen, dass während der GC weiterhin genügend Platz zur Allokation neuer Objekte in der Old Generation zur Verfügung steht. Den größten Nachteil des CMS stellt die mögliche Heap-Fragmentierung dar. Durch diese kann auch die Minor Collection verlangsamt werden, da bei der Promotion erst eine passende Lücke in der Old Generation gefunden werden muss. Die freien Speicherbereiche in der Old Generation werden wiederum mit einer FreeList verwaltet. Diesem Nachteil gegenüber steht der große Performance-Vorteil durch die deutlich kürzeren Stop-the-World-Pausen.

Lessons learned: Der CMS empfiehlt sich speziell für Systeme, deren Pausenzeiten kurz gehalten werden müssen. Hierzu zählen u. a. Webserver und interaktive Systeme. Eine gute und häufig verwendete Garbage Collector - Kombination ist Mark-and-Copy für die Young Generation und CMS für die Old Generation.

Garbage First (G1)
Der neueste GC-Algorithmus trägt die Bezeichnung *Garbage First (G1)*. Er wurde mit Java 6 Update 14 in einer experimentellen Version eingeführt. Seit Java 7 liegt er in einer finalen Version vor. Langfristiges Ziel ist die Ablösung des CMS. Der G1 nutzt ebenfalls die Vorteile der heutigen Hardware und versucht möglichst viele Arbeiten parallel durch mehrere Threads abzuarbeiten. Einige Aufgaben können auch nebenläufig zur eigentlichen Applikation durchgeführt werden. Allgemein lässt sich beim G1 die maximale Pausenzeit durch einen JVM-Parameter vorgeben. Die Einhaltung

dieser Zeit wird jedoch nicht garantiert. Daneben lässt sich auch das Intervall zwischen dem Start zweier GCs vorgeben.
Anders als bei den bisher beschriebenen Algorithmen wird der Heap beim G1 nicht in Generationen, sondern in *Regionen* eingeteilt. Eine Region ist dabei zwischen 1 MB und 32 MB groß. Die Größe kann konfiguriert werden, muss jedoch eine 2er-Potenz sein. Abbildung 4 zeigt die unterschiedlichen zur Verfügung stehenden Regions-Typen. Diese sind mit den bereits bekannten Generationen vergleichbar. Der Typ einer Region ist dabei nicht fixiert, sondern bestimmt sich dynamisch aus den aktuell vorliegenden Anforderungen.
Der GC-Durchlauf kann beim G1 kann in zwei Modi ablaufen, zwischen denen dynamisch gewechselt werden kann. Der *Fully Young Mode* bereinigt alle *Young Regions* (*Allocation-* und *Survivor-Regions*), wohingegen beim *Partially Young Mode* zusätzlich auch ausgewählte *Old Regions* bereinigt werden. Welche Regionen nun während einer GC bereinigt werden, bestimmt sich aus dem „Müllanteil“ der jeweiligen Region. Daher resultiert auch der Name dieses GC-Algorithmus, da Regionen mit hohem „Müllanteil“ (engl. garbage first) vorrangig bereinigt werden.

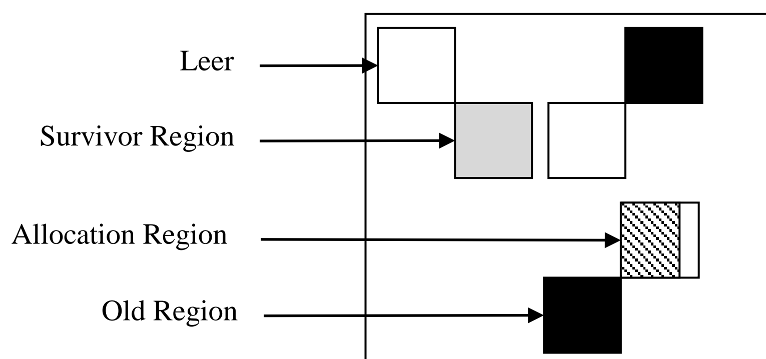

Abbildung 4: G1-Heap-Aufteilung (Quelle: In Anlehnung an [1, S. 232])

Jede Region verfügt über ein sogenanntes *Remembered Set*, welches alle Referenzen aus anderen Regionen in diese Region enthält. Für jede Änderung dieses Sets wird ein *Update Task* erstellt, der abgearbeitet wird, sobald genügend CPU-Zeit zur Verfügung steht. Nachfolgend wird ein GC-Durchlauf mit Markierung und *Evakuierung* beschrieben. Der Begriff Evakuierung bezeichnet das Verschieben von nicht entfernten Objekten in eine neue Region.
Wie beim CMS läuft auch beim G1 die Markierung nebenläufig zur Applikation ab (Concurrent Marking). Dabei wird ein *Snapshot-at-the-Beginning* erzeugt. Dieser enthält alle lebendigen Objekte zu Beginn der Markierung. Hieraus wird ermittelt, ob die Objekte, die in den Remembered Sets vermerkt sind, noch leben. Des Weiteren wird in einer *Count-Phase* die Größe der erreichten Objekte festgehalten. Anhand dieser Information werden die zu bereinigenden Regionen in einem Garbage-First-Ansatz ausgewählt. Objekte, die während der nebenläufigen Markierungs-Phase von der Applikation erstellt werden, sind automatisch als lebendig gekennzeichnet.
Die Evakuierung überlebender Objekte findet während einer Stop-the-World-Pause statt. Alle dabei anfallenden Aufgaben werden in relativ kleine Sub-Tasks zerlegt, die parallel von allen zur Verfügung stehenden CPUs abgearbeitet werden, um die Pause so kurz wie möglich zu halten bzw. um das angegebene Pausenziel zu erreichen. In dieser Pause werden drei Phasen durchlaufen. Zuerst werden die noch nicht durchgeführten Updates der Remembered Sets ausgeführt. Danach werden durch Referenzverfolgung alle lebendigen Objekte innerhalb des *Collection Sets* bestimmt. Die Referenzen eines Objekts werden dabei nur verfolgt wenn dieses Quell-Objekt lebendig ist. Zusätzlich werden alle Referenzen, die in das Collection Set verweisen, auf lebendige Objekte überprüft. Zu diesen Referenzen zählen z. B. RootSet-Referenzen sowie Referenzen aus den Remembered Sets anderer Regionen.

Die letzte Phase der GC-Pause ist das eigentliche Bereinigen der Regionen. Dabei werden alle überlebenden Objekte der Regionen in eine neue Region, den To-Space, kopiert und die bereinigten Regionen freigegeben. Die überlebenden Objekte aus unterschiedlichen Regionen können dabei in einer neuen Region zusammengefasst und ohne Fragmentierung in dieser angeordnet werden. Auch Regionen, welche überhaupt keine lebendigen Objekte mehr enthalten, werden freigegeben.

Lessons learned: Der G1 ist ein Allround-GC-Algorithmus, welcher im Vergleich zu den etablierten Collectoren einen neuen Ansatz der Heap-Bereinigung wählt. Der wichtigste Ansatz des G1 ist die Heap-Bereinigung über die Laufzeit der Applikation. Die Heap-Fragmentierung wird ebenfalls über die Laufzeit verringert. Durch die vorgebbaren Pausenzeiten werden nach und nach die einzelnen Regionen bereinigt. Ein Test des G1 empfiehlt sich daher auf allen Systemen.

Nach der Vorstellung der gängigsten Garbage Collectoren beschäftigt sich der kommende Abschnitt mit Ansätzen und Möglichkeiten des GC-Tunings.

Garbage Collection Tuning

Ist eine Applikation langsam, ist evtl. eine Optimierung der GC-Konfiguration anzustreben. Dabei ist festzuhalten, dass ein GC-Tuning nur bei einem Problem angestrebt werden soll, welches auf einen fehlerhaft konfigurierten Garbage Collector zurückzuführen ist. Andernfalls wird ein dementsprechendes Tuning nicht den erhofften Effekt bringen. Daher sind z. B. Implementierungsfehler in einer Applikation vor einem GC-Tuning auszuschließen. Ein möglicher Implementierungs-Schwachpunkt könnte eine statische Collection sein, zu welcher Objekte zwar hinzugefügt, jedoch nicht wieder entfernt werden, wenn sie nicht mehr benötigt werden. Über die statische Referenz der Collection können dann auch alle sich in der Collection befindlichen Objekte erreicht und als lebendig markiert werden. Allgemein ist der Einsatz von statischen Variablen zu überprüfen.

Die Feststellung, ob Applikations-Probleme (z. B. zu hoher Speicherverbrauch oder zu lange Antwortzeiten) durch GC verursacht werden, kann z. B. durch die Analyse von *GC-Traces* geschehen. Dies sind die Logfiles des Garbage Collectors, in denen sich dessen Aktivitäten inkl. der zeitlichen Dauer ablesen lassen. Die Detailtiefe der Logfiles kann mit JVM-Parametern eingestellt werden. Auf Grund der Größe und der zumeist sehr detaillierten Aussagen über freien Speicher, bereinigten Speicher, Zeiten für Speicherbereinigung, etc. ist eine manuelle Auswertung sehr langwierig. Hierzu kann alternativ das OpenSource-Tool *GCViewer* verwendet werden. Mit diesem kann man einen GC-Trace einlesen. Das Tool stellt die Informationen anschließend grafisch aufbereitet und in Tabellenform dar. Lange Pausenzeiten oder ein schlechter Durchsatz lassen sich so schnell erkennen. Korrespondieren die auftretenden Applikations-Probleme mit langen GC-Pausenzeiten, so kann man mit einer recht hohen Wahrscheinlichkeit annehmen, dass eine fehlerhafte GC-Konfiguration vorliegt und eine Optimierung anzustreben ist.

Einen wichtigen Aspekt, weshalb eine optimal konfigurierte GC für die Performance der Applikation unabdingbar ist, stellt die heutige leistungsfähige Hardware dar. Je mehr Rechenkerne gleichzeitig an der Allokation neuer Objekte beteiligt sind, desto schneller wird der Heap voll und eine GC ausgelöst. A. Langer und K. Kreft beschreiben dies in [1, S. 208] wie folgt: „Für das Erzeugen von Objekten ergibt sich also, was sonst im Allgemeinen nicht gilt: Die Anzahl neuer Objekte steigt (fast) linear mit der Anzahl der zur Verfügung stehenden Cores." Daraus lässt sich schließen, dass mit zunehmender Anzahl an CPUs die Anzahl der GC-Durchgänge steigt und der Durchsatz der Applikation sinkt. Daher sollte bei Erhöhung der CPU-Anzahl der Heap ebenfalls vergrößert werden, um einen zu schnellen Speichermangel zu verhindern.

Wie lässt sich die GC-Performance nun messen? Im Rahmen der Untersuchung konnten hierfür zwei wichtige Kennzahlen ermittelt werden. Diese sind auf der einen Seite der Durchsatz (*Throughput*) und auf der anderen Seite die verursachten Stop-the-World-Pausenzeiten. Daneben existieren weitere

Kennzahlen, wie z. B. die Häufigkeit einer GC (*Frequency of Collection*) oder die Zeitspanne, in welcher der Speicherplatz eines Objekts nach Entfernen der Verbindung zu einer Root-Referenz wieder verfügbar wird (*Promptness*). Bei älteren Systemen oder Systemen mit beschränktem Arbeitsspeicher kann auch der unnötige Verbrauch von Arbeitsspeicher eine wichtige Kennzahl sein. Die zuerst genannten Kennzahlen werden nachfolgend näher beleuchtet. Daneben werden Optimierungsmöglichkeiten diese Kennzahlen betreffend vorgestellt.

Durchsatz

Unter Durchsatz wird der Prozentsatz an Zeit verstanden, welchen eine Applikation nicht mit GC beschäftigt ist. Für ein aussagekräftiges Ergebnis sollte dieser über einen längeren Zeitraum ermittelt werden, am besten über die Gesamtlaufzeit der Applikation. Der Kehrwert, also die prozentual mit GC verbrachte Zeit, wird als *GC-Overhead* bezeichnet.

Die Ermittlung des Durchsatzes kann aus dem GC-Trace erfolgen. Hierzu sind die fett markierten Bereiche des nachfolgenden Beispiel-Traces aus [1, S.200] relevant: **176.532**: [GC 56794K -> 17083K (129792K), **0.1041413 secs**]. Der erste markierte Wert ist dabei der Zeitstempel (gemessen in Sekunden ab Start der Applikation), an welchem die GC begonnen wurde. Der zweite markierte Wert stellt die verursachte Stop-the-World-Pause dar. In die Durchsatz-Berechnung geht die Gesamtzeit des Trace (gz) und die Summe über alle Stop-the-World-Pausen (zmgc) ein. Der Durchsatz (in %) ist nach folgender Formel zu berechnen: _(100 * (1 - zmgc / gz))_. Ein Nachteil ist jedoch, dass die CPU-Zeiten der nebenläufigen GC-Aktivitäten nicht berücksichtigt werden. Dies liegt u. a. daran, dass diese nicht explizit im GC-Trace ausgewiesen sind.

Werte > 95% werden als sehr guter Durchsatz angesehen. Bei vielen Systemen ist eine Optimierung des Durchsatzes praktisch eher irrelevant (Ausnahme: Batchbetrieb), da sehr lange Pausenzeiten entstehen können. Der Aufwand, den Durchsatz z. B. um 10% zu steigern, ist häufig sehr hoch. Alternativ kann eine CPU mit mehr Leistung eingesetzt werden, wodurch der Durchsatz zwar vorerst nicht erhöht, die Performance der Applikation durch schnellere Verarbeitung der GC-Aktivitäten dennoch gesteigert wird. Der Grund hierfür ist in den kürzer ausfallenden Stop-the-World-Pausen zu sehen.

Stop-the-World-Pausen

Die Reduzierung von GC-Pausenzeiten ist ein häufig angestrebtes Ziel und oft der Anlass sich näher mit dem GC-Tuning zu beschäftigen. Wie bereits dargestellt, sind GC-Pausen unvermeidbar und stellen tendenziell auch kein Problem dar. Werden sie jedoch zu lang, sodass sie den Betrieb der Applikation beeinträchtigen (z. B. durch zu lange Reaktionszeiten bei einer interaktiven Applikation), rücken sie in den Fokus einer Optimierung. Auch hierbei muss vor einer GC-Optimierung sichergestellt sein, dass der Fehler durch einen fehlerhaft konfigurierten Garbage Collector ausgelöst wird. Häufig sind auch hier Design- und Implementierungsfehler eher der Grund für eine langsame Applikation. Können diese ausgeschlossen werden, so lassen sich z. B. die GC-Traces nach langen Pausenzeiten durchsuchen. Korrespondieren diese mit dem Auftreten des Problems, kann im nächsten Schritt ein GC-Tuning hinsichtlich der Reduzierung der Pausenzeiten begonnen werden.

Lessons learned: Es wurde dargestellt, dass der Durchsatz und die Stop-the-World-Pausenzeiten sehr gute Kennzahlen für die Ermittlung der GC-Performance sind. Diese lassen sich relativ leicht durch Hilfsmittel wie GCViewer aus den GC-Traces ermitteln. GC-Traces können dabei automatisch durch die JVM erstellt werden, sofern dies mittels Parametern eingeschalten ist.

Konkrete GC-Tuning-Maßnahmen

Es lässt sich festhalten, dass ein GC-Tuning ein iterativer und möglicherweise langwieriger Prozess ist. Nach jeder durchgeführten Änderung sollte überprüft werden, ob man dem gesetzten Ziel näher gekommen ist. Auch müssen Ziele während des Tunings evtl. angepasst oder aufgegeben werden. Für

die gewählten Ziele ist wichtig, dass diese realistisch und zu erreichen sind. So ist z. B. ein Ziel „Keine Pausenzeiten bei 100% Durchsatz“ weder realistisch, noch erreichbar. Es empfiehlt sich in jedem Fall vor einer GC-Konfiguration die Standardwerte der JVM zu testen. Oftmals lassen gänzlich ohne eigene Konfiguration bereits respektable Ergebnisse erzielen. Mit jedem Upgrade der JVM ist jedoch ein neuer Test notwendig, da sich die Standardeinstellungen ändern können.
Der erste Schritt bei einem GC-Tuning ist die Wahl des richtigen Garbage Collectors. Für hohen Durchsatz empfiehlt sich der Einsatz des parallelen Mark-and-Copy für die Young- sowie der parallele Mark-and-Compact für die Old Generation. Um eine Optimierung hinsichtlich kurzer Pausenzeiten anzustreben kann für die Old Generation alternativ der CMS gewählt werden.
Im zweiten Schritt sind die gewählten Collectoren an die Objektpopulation der zu tunenden Applikation anzupassen. Dabei ist zu beachten, dass sich die Objektpopulation über die Laufzeit der Applikation mit hoher Wahrscheinlichkeit verändert. Webshops haben z. B. Zeiten in denen sie stärker belastet werden und somit mehr Objekte erzeugen. Das Tuning ist in jedem Fall an die Lastspitzen des Systems anzupassen. Falls ein dementsprechendes Tuning für den Regelbetrieb zu viele Ressourcen benötigt, können auch individuelle Tunings auf unterschiedlichen JVMs für einzelne Betriebszeiten durchgeführt werden.
Die wichtigste Tuningmaßnahme ist die Anpassung der Größenverhältnisse zwischen Young- und Old Generation sowie zwischen Eden und den Survivor Spaces. Ziel bei der Anpassung sollte sein, Objekte möglichst in der Young Generation sterben zu lassen, um langwierige Major Collections zu vermeiden. Für das Verhältnis Young- zu Old Generation steht der JVM-Parameter
-XX:NewRatio=<value> zur Verfügung. Wird dieser z. B. auf „3“ gesetzt, hat dies ein Größenverhältnis von 1:3 zwischen Young- und Old Generation zur Folge. Die Anpassung zwischen Eden und den Survivor Spaces wird über den Parameter
-XX:SurvivorRatio=<value> durchgeführt. Wird für diesen der Wert „6“ gewählt, entspricht ein Survivor Space 1/8 der Young Generation-Größe.
Eine weitere Möglichkeit, die Promotion in die Old Generation zu verhindern, ist die Erhöhung des Tenuring Threshold, dem Schwellwert ab wie viel überlebten GCs ein Objekt in die Old Generation übertragen wird. Objekte mit mittellanger Lebenszeit können so länger in den Survivor Spaces gehalten werden, was wiederum Aufwände für die GC der Old Generation sparen kann.

Zur Verkürzung der Pausenzeiten empfiehlt sich ein Collector-Typus, welcher möglichst viele GC-Aufgaben nebenläufig ausführt. Hierzu kann z. B. der CMS gewählt werden. Allerdings besteht bei diesem das Problem der Heap-Fragmentierung, welches im schlechtesten Fall eine extrem lange Pausenzeit verursacht. Der Grund hierfür ist ein Rückfall in den seriellen Mark-and-Compact, falls in der Old Generation keine Objekte mehr allokiert werden können. Um dies zu verhindern kann man einerseits die Old Generation extrem vergrößern. Zu einem für die Applikation unkritischen Zeitpunkt kann dann durch den Aufruf von *System.gc()* manuell eine GC ausgelöst werden. Eine andere Anpassungsmöglichkeit beim CMS ist die Einstellung der OccupancyFraction. Ziel dabei ist, dass der CMS weder zu oft, noch zu selten läuft. Falls kein expliziter Wert für den Füllgrad gesetzt wird, wird dieser von der JVM während der Laufzeit dynamisch angepasst. Zu Grunde liegen dabei Informationen, welche von der JVM automatisch bei jeder durchgeführten GC gesammelt werden.

Lessons learned: GC-Tuning bedarf eines iterativen Tunings. Der Schwerpunkt einer GC-Optimierung liegt meist auf Anpassung von Größenverhältnissen zwischen den einzelnen Heap-Bereichen. Hauptziel ist ein Verhindern der Promotion von Objekten in die Old Generation. Die Größenverhältnisse sind dementsprechend so anzupassen, dass Objekte mit kurzer und mittellanger Lebenszeit möglichst in der Young Generation (inkl. den Survivor Spaces) verbleiben, bis sie entfernt werden.
Das abschließende Kapitel beschreibt die Ergebnisse durchgeführten Leistungs-Messungen mit unterschiedlichen GC-Konfigurationen.

Leistungs-Messungen
Im Rahmen der Untersuchung wurden Leistungs-Messungen mit unterschiedlichen GC-Konfigurationen durchgeführt. Als Testsystem stand dabei ein Clusterknoten (in Kopie) eines bereits produktiv eingesetzten B2B-Webshop-Systems zur Verfügung. Um realistische Testbedingungen zu schaffen und vergleichbare Testresultate zu erhalten, wurden am produktiven System *Requests* aufgezeichnet, welche am Testsystem beliebig häufig wiedergegeben werden konnten. Es wurden dabei Lasttests mit zwei unterschiedlichen GC-Konfigurationen durchgeführt. Jeder Test wurde sechsmal wiederholt. Das Testsystem wurde pro Testdurchlauf ca. sieben Stunden mit Requests belastet. Die dabei erstellen GC-Traces wurden mittels GCViewer analysiert und ausgewertet. Anschließend wurden aus allen durchgeführten Testdurchläufen Mittelwerte gebildet, um einmalige Schwankungen auszugleichen. Diese Mittelwerte sind in den Abbildungen dieses Abschnitts dargestellt.

Im ersten Testfall wurde die GC-Kombination Parallel Mark-and-Copy (Young Generation) und CMS (Old Generation) untersucht. Die Heap-Größe betrug 16 GB, das Verhältnis zwischen Young- und Old Generation „1:5“. Insgesamt wurden 30 parallele GC-Threads verwendet. Als maximale Pausenzeit für den CMS wurden einmal 500 ms und einmal 1000 ms vorgegeben. Ziel dieses Tests war die Überprüfung, ob es sich bei der gewählten Konfiguration um eine realistisch einsetzbare Konfiguration handelt.
Abbildung 5 (links) zeigt, dass der Durchsatz bei diesem Testfall mit ~95% sehr hoch war. Korrespondierend dazu war auch die Gesamtsumme der Pausenzeiten (Abbildung 5, rechts) moderat. In Abbildung 6 (links) ist hingegen zu sehen, dass die vorgegebenen maximalen Pausenzeiten im Durchschnitt nicht erreicht wurden. Speziell die Major Collections konnten diese ziemlich deutlich nicht einhalten. Auch die maximalen Pausenzeiten (Abbildung 6, rechts) waren extrem hoch. Dies führte in Summe zu einer hohen Anzahl an Request-Timeouts während des Tests. Als Fazit dieses Testfalls zeigte sich, dass die Kombination Mark-and-Copy und CMS eine gute Wahl für einen hohen Durchsatz darstellt. Die Verringerung der Pausenzeiten wären der nächste Ansatzpunkt für weitere Optimierung. Auf Grund verschiedener Einschränkung konnte im Rahmen der Untersuchung hier jedoch keine weitere Optimierung vorgenommen werden.

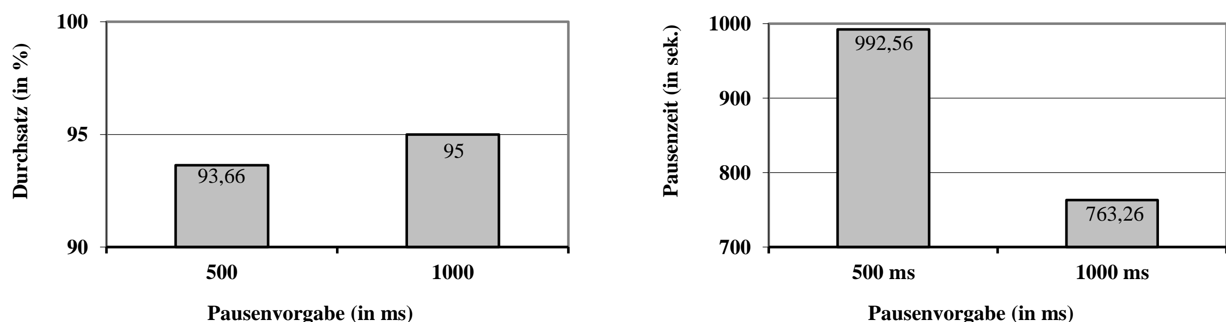

Abbildung 5: Lasttest CMS I - Durchsatz (links); ∑ Pausenzeiten (rechts)

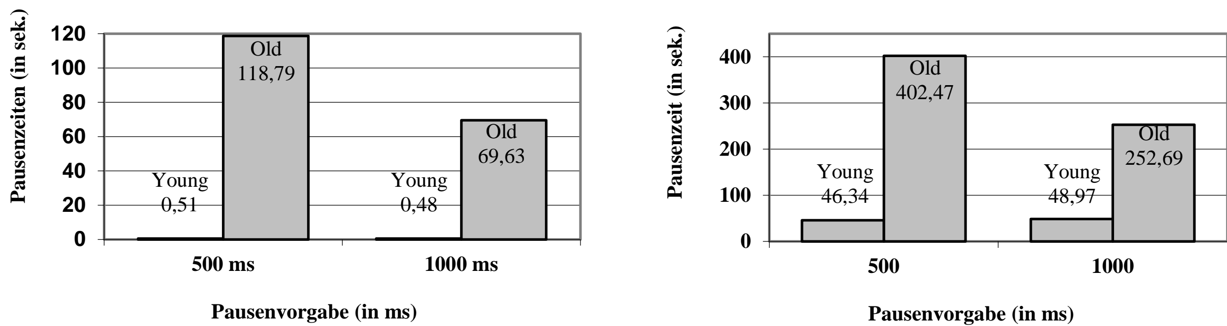

Abbildung 6: Lasttest CMS II - Ø Pausenzeit (links); maximale Pausenzeiten (rechts)

Beim zweiten Lasttest wurde der G1 eingesetzt. Um eine finale Version dieses Collectors zu verwenden, wurde für diesen Testfall Java 7 eingesetzt. Die Heap-Größe betrug erneut 16 GB und es kamen 30 parallele GC-Threads zum Einsatz. Als maximale Pausenzeiten wurden wie beim vorherigen Test 500 ms bzw. 1000 ms gewählt.
In Abbildung 7 (links) ist zu sehen, dass auch mit dem G1 ein sehr hoher Durchsatz-Wert erzielt wird. Doch, wie die durchschnittlichen Pausenzeiten in Abbildung 7 (rechts) zeigen, wurden auch mit dem G1 die vorgegebenen Pausenziele deutlich nicht eingehalten. Ein weiteres Ergebnis dieses Testfalls war, dass das vorliegende Testsystem bei Verwendung des G1 sehr viele Request-Timeouts hatte. Ein genauer Grund hierfür konnte nicht ermittelt werden. Daher konnte keine Empfehlung zum produktiven Einsatz des G1 gegeben werden. Die Leistungsfähigkeit dieses Collectors müsste mit weiteren Tests und Konfigurationsanpassungen überprüft werden. Auf Grund unterschiedlicher Einschränkungen konnte dies im Rahmen der Untersuchung jedoch ebenfalls nicht bewerkstelligt werden.

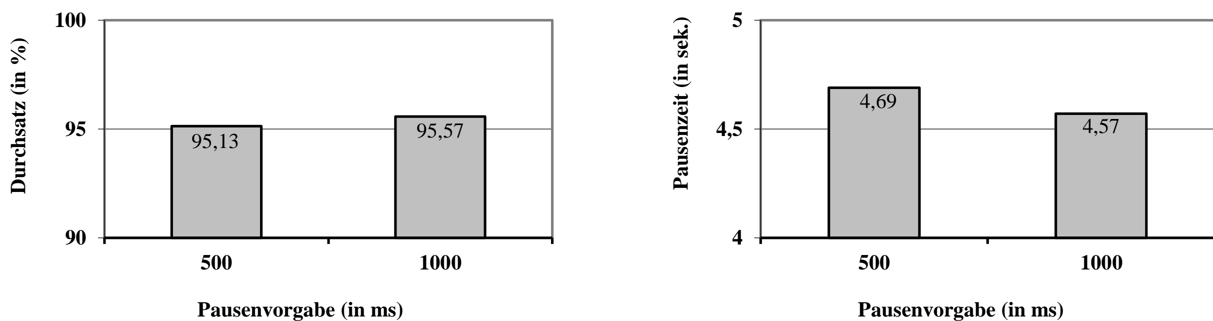

Abbildung 7: Lasttest G1 - Durchsatz (links) ; Ø Pausenzeiten(rechts)

Fazit

Bei der Untersuchung der in der HotSpot ™ JVM zur Verfügung stehenden Garbage Collectoren wurde festgestellt, dass es sich um ein breites, komplexes Themengebiet handelt. Jeder zur Verfügung stehende Garbage Collector bietet eine Summe an Einstellmöglichkeiten. Für die GC-Konfiguration an sich konnten keine allgemeingültigen Empfehlungen gegeben werden. Es wurde sich daher darauf beschränkt, Richtlinien zu erarbeiten, welche für die meisten Applikationen gelten sollten:

- Der JVM sollte so viel wie möglich Arbeitsspeicher zugewiesen werden (Heap-Größe). In einem offiziellen Oracle-Dokument ([6, Abschnitt 4]) wird hierzu folgende Richtlinie gegeben: „Unless you have problems with pauses, try granting as much memory as possible to the virtual machine.“

Bei der Zuweisung von Arbeitsspeicher an die JVM ist jedoch darauf zu achten, dass es sich um echten physikalischen Speicher handelt, um *Paging* zu vermeiden. Um mehr Speicher allokieren zu können ist evtl. auch ein Umstieg von einer 32-Bit-JVM auf eine 64-Bit-JVM in Erwägung zu ziehen.

- Das Verhältnis zwischen Young und Old Generation ist so zu wählen, dass nur wirklich langlebige Objekte in die Old Generation promoviert werden. Hierzu können neben der Young Generation-Größe auch die Größe der Survivor-Spaces sowie der Tenuring Threshold angepasst werden.

- Eine GC der Old Generation kann durch eine Vergrößerung des ihr zustehenden Heaps hinausgezögert werden. Sobald die Applikation weniger belastet ist, kann eine GC manuell ausgelöst werden.

- Die Wahl eines passenden Garbage Collectors muss an die angestrebten GC-Ziele angepasst werden. Je nach Collector und dessen Konfiguration kann entweder der Durchsatz erhöht und/oder die Pausenzeit minimiert werden.

- Eine physikalische Erweiterung der Ressourcen kann unter Umständen eine stärkere Wirkung haben als (langwierige) Parameter-Optimierungen. Hierzu zählt z. B. die CPU-Leistung, der verfügbare Arbeitsspeicher sowie die benutze Variante der JVM (32-Bit/64-Bit).

- Bei Erhöhung der CPU-Anzahl sollte auch die Heap-Größe erweitert werden. Hintergrund ist hier, dass die Allokation neuer Objekte sehr gut parallel ablaufen kann. Um den Durchsatz konstant zu halten muss folglich auch die GC parallel durchgeführt werden, da die Anzahl der GCs vermutlich ansteigen wird.

Quellen

[1] Langer, Angelika; Kreft, Klaus (2011): Java-Core-Programmierung. Memory Model und Garbage Collection. Frankfurt: entwickler.press.

[2] Memory Management in the Java HotSpot™ Virtual Machine; http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf (abgerufen am 11.11.2011, 08:31 Uhr)

[3] Java HotSpot VM Options; http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html (abgerufen am 06.12.2011, 14:41 Uhr)

[4] Java HotSpot Garbage Collection; http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html (abgerufen am 08.12.2011, 08:48 Uhr)

[5] Bacon, David F.; Diwan, Amer; Detlefs, David; Flood, Christine; Heller, Steve; Printezis, Tony (2004): Garbage-first garbage collection. In: Proceedings of the 4th international symposium on Memory management - ISMM '04: ACM Press, S. 37 - 48.

[6] Java SE 6 HotSpot™ Virtual Machine Garbage Collection Tuning; http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html (abgerufen am 13.12.2011, 08:25 Uhr)

[7] Java Memory Leaks und andere Übeltäter (2.Akt); http://blog.codecentric.de/2011/03/java-memory-leaks-und-andere-ubeltater-2-akt/ (abgerufen am 15.12.2011, 09:06 Uhr)

[8] Wie funktioniert der Java Garbage Collector; http://it-republik.de/jaxenter/artikel/2452 (abgerufen am 13.12.2011, 12:51 Uhr)

Kontaktadressen

Mathias Dolag
iSYS Software GmbH
Grillparzer Str. 10
D-81675 München

Telefon: +49 (0) 89-462328 44
Fax: +49 (0) 89-462328 14
E-Mail m.dolag@isys-software.de
Internet: www.isys-software.de.de

Prof. Dr. Peter Mandl
Hochschule München
Loth Str. 64
D-80335 München

Telefon: +49 (0) 89-1265 3704
Fax: +49 (0) 89-1265 3780
E-Mail mandl@cs.hm.edu
Internet: cs.hm.edu

Christoph Pohl
Hochschule München
Loth Str. 64
D-80335 München

Telefon: +49 (0) 89-1265 3764
Fax: +49 (0) 89-1265 3780
E-Mail christoph.pohl0@hm.edu
Internet: cs.hm.edu