Concurrent Garbage Collection

Seminar: Sprachen für Parallelprogrammierung

Emre Kasif Selengin

18. Juli 2010

Zusammenfassung

Die Programmiersprachen mit automatischen Speicherverwaltung werden immer beliebter. Programmiersprache Java oder Common Intermediate Language Sprachen (wie C#) erlauben dem Programmierer, Objekte in den Speicher zu allokieren und danach löschen, ohne dass er dafür kein explizites Code schreiben muss. Dies wird durch den Algorithmus *Garbage Collector(GC)* möglich.
Man hat dieses Konzept in den letzten 15 Jahren intensiv geforscht, da der Effizienz von GC auch entscheidend dafür ist, ob man diese Sprachen in der Praxis benutzen wird.
Eine der Ideen, die GCs effizienter gemacht haben, ist der Benutzung der Parallelismus. In diesem Dokument werden parallele und nebenläufige Varianten von GC zusammengefasst.

1 Einführung

Die *managed* oder nicht-nativen Sprachen, wie C# oder Java, bieten eine automatische Speicherverwaltung an. Verbesserung dieses Konzepts ist ein Forschungsthema von Firmen wie Microsoft und Sun. Ein wichtiges Teil dieser Speicherverwaltung ist Garbage Collection (sogar manchmal werden diese als Synonym verwendet), die dafür sorgt, mit der Automation allozierten Speicher zu bereinigen und auch defragmentieren.

Obwohl viele Programmiersprachen seit den 50er Jahren die Idee von Garbage Collection benutzt [1,2] haben, finden die Forscher neue Ideen, die Garbage Collection Algorithmen verbessern. Sie wurde zuerst nicht intensiv geforscht, weil die explizite Speicherverwaltung wegen Effizienzgründen in der Praxis bevorzugt wurde. Obwohl die Einfachheit von Garbage Collection bewusst und akzeptiert wurde, musste sie warten bis die Programmiersprache Java sich verbreitet, damit sie auch Mainstream und in großen Systemen benutzt wird [1].

Die naive Implementation der Garbage Collection bringt zwei große Probleme mit sich: Durchsatz (engl. throughput) und Wartezeit (engl. latency). Die Entwickler waren immer skeptisch wegen dieser Probleme. Sie bedeuten, dass der GC-Algorithmus die Kontrolle über ihre Programme nehmen und lange Pausen hinzufügen oder die Effizienz des Programms im allgemein beeinflussen oder beides. Diese Sorgen der Entwickler waren auch durch mit neuen Technologien (also schnelleren Computern), nicht weg, weil solche Technologien entsprechende Vergrößerung an Speicherbedarf mitgebracht haben[1].

Heute ermöglichen Forschungen neue Ansätze bei Garbage Collection gegen solche Probleme. Hier werden diese Ansätze zusammengefasst. Es gibt Dokumentationen (wie [3],[4],[14]), die versuchen, den Entwickler beim Wahl zwischen Algorithmen und bei der Anpassung von Garbage Collection zu Bedürfnissen von verschiedenen Arten von Programmen zu helfen.

Zur Verbesserung von GC wurden neue Ansätze entwickelt. Dazu gehören auch die Concurrent Garbage Collectors (GCGs). Es gibt keinen einzigen GCG. Die heutigen Collectors sind hybride Ansätze, wo dieses Konzept vorkommt. Beispielsweise die Java Virtual Machine HotSpot von Sun [3] bietet vier Arten von Kollektoren und einer davon; Concurrent Mark-Sweep (CMS) arbeitet nach dieses Prinzip. Außerdem gibt es weitere Garbage Collectors in den heutigen JVMs, die auch mit Hilfe der Parallelismus funktionieren. Obwohl CGCs und diese parallelen GCs unterschiedlich sind, möchte ich beide zusammenfassen.

Concurrent Garbage Collectors (oder CGCs) werden möglich durch heutigen Zustand der parallelen Verarbeitung. Parallelisierung ist ein wichtiges Forschungsthema und in der Praxis werden Programme parallelisiert. Dann wieso auch nicht Garbage Collector?

In diesem Dokument werden Concurrent (und Parallele) Garbage Collection mit anderen verglichen und praktische Implementierungen erwähnt.

1.1 Terminologie

Zuerst muss man ein Paar Begriffe klarstellen, wo es auch manchmal Konflikte gibt:

Garbage Collection

Wie schon oben erwähnt, benutzt man manchmal “Garbage Collection” als Synonym für automatische Speicherverwaltung. Es liegt auch daran, dass diese sehr stark voneinander abhängig sind aber Garbage Collection bedeutet eigentlich “Müllabfuhr”.

Gemeint ist hier, dass die Programmierungsumgebung die Aufgabe vom Programmierer nimmt und nicht-benutzte Objekte automatisch erkennt und löscht. Dabei können auch Fragmentierungen entstehen. GC soll dieses Problem auch lösen. Kurz gesagt: GC ist das Programm, das die automatische Bereinigung des Speicher macht.

Concurrent vs. Parallel

Meine zwei Hauptartikeln kollidieren bei der Benutzung dieser Begriffe. Deshalb heißt der Algorithmus in [5] "Mostly Parallel Garbage Collection" während die in [1] "Mostly-concurrent Garbage Collector", obwohl diese gleiche Algorithmen sind. Die Terminologie von neuerem Paper [1] wird heute mehr akzeptiert, deshalb benutze ich sie auch.

Parallel bedeutet (mindestens in Dokumente über GC), dass der Algorithmus parallel in mehreren Prozesseinheiten bearbeitet wird und die Aufgabe zwischen diesen Prozessen verteilt wird. In unserem Fall: Die gleiche Datenmenge des seriellen Garbage Collection werden von Threads der parallelen Garbage Collector bearbeitet und dadurch wird ein Speedup erzielt.

Concurrent (nebenläufig) heißt dagegen, der Algorithmus kann nebenläufig mit anderen Threads (im GC-Fall: das eigentliche Programm) unabhängig arbeiten. Concurrent GC kann bei Objektenallokationen mit dem eigentlichen Programm kommunizieren müssen. Dies geschieht in kleinen Schritten aber die große Arbeit wird immer noch in einem unabhängigen Thread gemacht. Damit das Programm darf unabhängig arbeiten.
Es ist möglich, einen Parallel-Nebenläufige GCs zu entwerfen.

Incremental (inkrementell) bedeutet, dass der Algorithmus nur schrittweise arbeitet. Solche verursachen nur kleine Pausen. Manchmal werden sie auch concurrent genannt, obwohl sie nicht auf einem anderen Thread existieren.

Write Barriers

sind die Befehle, die nach jedem Schreibzugang auf einem Objekt automatisch hinzugefügt wird. Diese Befehle erlauben die Kommunikation vom Programm zum Garbage Collector, indem er den GC die Änderungen im Speichergraph meldet[6].

Mutator

Das eigentliche Programm. Der Anwendung.

1.2 (eine Zusammenfassung der) Geschichte der (Concurrent) Garbage Collection

Die erste Garbage Collectors wurden für LISP Programmierung entwickelt. John McCarthy hat eine kurze Beschreibung der "automatischen Speicherreklamation" im Jahr 1960 veröffentlicht. Damals hieß es noch nicht "garbage collection". 1968 hat D.G. Bobrow das erste System beschrieben, das inkrementelles Garbage Collection mit *write barrier* (s.o.) erlaubte. Hier wurde zum ersten mal GC als ein unabhängiges Thread beschrieben.

Dijkstra et. al.[7] haben 1978 der Mark-and-Sweep Algorithmus mit der Tri-colour Markierung (s.u) entwickelt. Diese sind beliebte Ansatz und werden auch in neuen Algorithmen benutzt. Neue Ideen für ihre Verbesserung werden vorgeschlagen: 1989 versuchten Queinnec et. al. Mark- und Sweep-phasen zu trennen und parallel arbeiten zu lassen. 1993 haben Wallace und Runciman eine Variante vorgeschlagen, die in eingebetteten (embedded) Echtzeitsystemen besser funktionieren soll. Diese Idee wurde 1998 von Heuelsbergen und Winterbottom weiterentwickelt. Doligez und Gonthier[8] haben der Algorithmus von Dijkstra zu den Multiprozessor-Anwendungen angepasst.[6]

1991 haben Boehm [6] et. al die Idee von Concurrent Garbage Collection mit Generational GC verbunden und 2000 haben Tony Printezis und David Detlefs [1] diese Version für Java Virtual Machine realisiert. Diese Implementierung ist bis jetzt in der JVM geblieben.

Printezis et al.[11][12][13] haben dieses Thema weitergeforscht und 2008 den "Garbage First Collector" entwickelt. Dieser wird noch getestet und mit Java 1.7 vollständig veröffentlicht. Der Algorithmus von G1 ist wieder ein Hybrid von CGC und PGC. Der Concurrent Mark-Sweep collector wird auch durch G1 ersetzt[12].

2 Funktionsweise

2.1 naive Garbage Collection

Hier werden die Funktionsweisen von naiven GCs und ihre zwei großen Problemen erklärt.

Alle GCs erlauben dem Programmierer, jederzeit Speicherplatz für ein Objekt zu reservieren. Wenn dieses Objekt nicht mehr verwendet wird, sollte sein Platz wieder verfügbar gemacht werden. Das passiert durch den GC automatisch, ohne dass dies explizit im Programm kodiert werden muss.

Der Zeitpunkt der Bereinigung ist nicht vorhersehbar, weil der GC selber entscheidet, wann er seine Aufgabe macht. Dieser kann man in heutigen Programmierumgebungen parametrisieren, trotzdem bleiben GCs nicht-deterministisch.

Man muss noch sagen, dass GCs üblicherweise in Zyklen arbeiten. Nachdem sie merken, dass die Speicherbenutzung an einen bestimmten Grenze nähert, fäng sie mit der Arbeit an. Danach wiederholt sich dieses Vorgang automatisch. Viele GCs (z.B von JVM oder .NET-Framework) haben Konstrukte, damit man ihn außerhalb des Zyklus zur Arbeit zwingt.

```
Object o = new SomeComplexObject();
// ... work with o ...
o = null;
```

Hinter einem einfachen Code, wie diesem, steckt die Allokation des Objekts *SomeComplexObject* in den Speicher. Nach der Arbeit wird die Referenz (o) auf dem Objekt wieder auf "null" gesetzt. Wenn GC versucht während der Arbeit (in der zweiten Zeile) den Speicher zu bereinigen, erkennt er, dass o noch benutzt wird und löscht es nicht aber wenn er nach der Arbeit ins Spiel kommt, erkennt das unbenutzte Objekt (durch die Null-Referenz) und löscht es.

Ein Objekt wird als *tot* erklärt, wenn es nicht mehr referenziert wird. Im obigen Beispiel wird das SomeComplexObject nur durch Referenz o zugegriffen. Natürlich können mehrere Referenzen auf das gleiche Objekt zeigen. In dem Fall muss das Objekt lebendig bleiben. Deshalb muss ein GC immer sicher stellen, dass ein Objekt *wirklich tot* sind, bevor er es löscht.

Das großes Problem in GC ist, dass ihr Algorithmus das gesamte System anhalten muss, damit er die Objekte im Speicher besucht. Er muss Objekte kontrollieren, ob diese von der Anwendung wirklich referenziert wird. Deshalb kann er nicht erlauben, dass man diese Referenzen ändert. Dieses Phänomen nennt man"*Stop-the-world*". Dadurch werden Wartezeiten erhöht.

Außerdem der allgemeine Effizienz des Programms ist durch den GC verschlechtert, besonders wenn Overhead zu groß wird. Durch die inkrementelle Ausführung werden die Wartezeiten kurz gehalten aber die eigentliche Arbeit bleibt gleich. Also es lohnt sich nicht immer GC inkrementell auszuführen, damit die Pausen kürzer sind.

2.1.1 Tracing

Tracing ist die Phase, wo (alle) Garbage Collector(s) die Objekte im Speicher besuchen und markieren. Hier werden die Objekten nach ihrem Lebenszyklus untersucht. Eine naive Methode dafür wäre Reference Counting. Diese Methode in modernen (parallelen) GCs nicht vorkommen. Folgende Methoden dagegen benutzt man in neuen Algorithmen.

naive mark-and-sweep ist eine Methode zum Tracing. Der beste Anfangspunkt ist das "Root" [1] fürs Tracing. Jedes benutzte Objekt ist aus dem Root entweder direkt oder transitiv erreichbar. In der Markierungsphase *(Mark)* werden die Objekte besucht, die transitiv erreichbar sind und dieser Vorgang wiederholt sich, bis kein Objekt mehr erreicht wird. In der *Sweep*-phase werden alle nicht markierte Objekte gelöscht.

Tri-colour Diese Variante von mark-and-sweep wurde von Dijkstra et. al.[7] entwickelt. Hier wird die Markierung nicht mit zwei Zuständen (mit einem Bit) gemacht, sondern mit drei Zuständen (oder Farben): Weiß, schwarz, grau. Der Algorithmus funktioniert folgendermaßen.

[1] Üblicherweise mit Root sind globale Datenbereiche, Registers und Stacks gemeint. Yeates und Champlain[9] erweiteren diese mit anderen Heaps (in Multi-Heap-Systemen) und Datenbereiche durch die Netzwerke (in verteilten Systemen)

Alle Objekte, die von "Roots" erreicht wird, werden zuerst grau markiert. Andere sind normalerweise nur weiß. Alle weiße objekte die durch grauen Objekten erreichbar sind, werden danach auch grau markiert. Wenn ein alle Unterobjekte eines grauen Objekt markiert sind, wird dieses Objekt schwarz markiert. Wenn alle grauen Objekten durchgegangen sind, der Graph besteht nur aus schwarzen und weißen Objekten. Es gibt niemals einen Referenz von einen schwarzen zu einem weißen Objekt. Also wenn man erkennt, dass der Graph stabil ist (kein graues Objekt), weiße Objekte können gelöscht werden.
Mit diesem Algorithmus ist man in der Lage, GC parallel zur Anwendung laufen zu lassen. "A Generational Mostly-concurrent Garbage Collector[1]", der auch in die JVM geschafft hat, funktioniert nach dieses Prinzip.

2.2 weitere GCs

2.2.1 Generational GC

Dieses Verfahren wird in der Praxis sehr oft benutzt. .NET Framework und JVM benutzen "Generationen" um Objekte zu klassifizieren. In .NET hat 3 und das standard-JVM hat 2 Generationen.
Es wurde empirisch beobachtet, dass die meisten neuen Objekten (*Young*) schneller nicht-erreichbar werden [2]. Deshalb ist es logisch die Neuen und Alten getrennt zu betrachten. Zum Beispiel: Alte Objekte können seltener besucht werden oder man benutzt verschiedene Parameter für unterschiedliche Generationen. Wenn ein Young-Object genug GC-Zyklen überlebt, dann kann man sie zu einem Alten befördern.
Damit Objekte von verschiedenen Generationen zusammen bleiben, werden sie in bestimmten Regionen gespeichert. Solche Regionen können mit verschiedenen Strategien bereinigt werden. Es lohnt sich, sich auf neuere Objekte zu konzentrieren. Damit wird der Durchsatz verbessert aber es ist nicht genug, weil die Regionen von Alten Objekten werden auch irgendwann besucht werden. Hier hat der GC ein Durchsatz und Latenz wie "full-Heap GC".

2.3 GCs, die Parallelismus ausnutzen

2.3.1 Parallel GC

Wie parallele Algorithmen im allgemeinen Fall arbeiten, ist oben erklärt. Diese GCs sind parallele Varianten des GC. Diese arbeiten auf der gleichen Datenmenge, die ein naiven GC haben würde (Speicher, Heap). Heap wird zwischen Threads verteilt. Was der Algorithmus mit diesen Teilen macht, ist eigentlich abhängig von anderen Architekturentscheidungen: Mark-and-sweep Algorithmus auf jeden Partition parallel laufen lassen, verbessert die allgemeine Effizienz des GCs.
Es wurde auch vorgeschlagen, dass man die Phasen des Algorithmus parallelisiert. Also ein Thread ist für Markierung und eins für Löschen zuständig. Diese Idee wurde in der Praxis nicht benutzt.

[2] Dieses Phänomen wurde in *generational hypothesis* beschrieben.

Parallele GCs sind in der Regel stop-the-world Algorithmen. In der Praxis wird diese Variante von GCs benutzt. Alle CPUs werden ausgenutzt und die Pause wird kurz gehalten, indem man die Arbeit parallelisiert. Durchsatz wird dabei auch verbessert. Deshalb nennt man solche GCs auch “throughput-collectors”.
Wie jeder parallele Algorithmus, werden Synchronization gebraucht, wenn PGCs zu komplex werden.

2.3.2 Concurrent GC

CGCs versuchen die Wartezeiten zu verkürzen. Deshalb nennt man diese auch “low-latency” Algorithmen.
In solchen Algorithmen läuft ein GC-Prozess (oder Thread) neben der Anwendung. Die Markierung der Objekte werden in diesem Prozess gemacht, während der Mutator ohne Unterbrechung arbeitet. Mit Hilfe von Tri-Colour-Algorithmus(s.o.) werden die Objekte markiert. Falls der Mutator dazwischen noch im Speicher etwas ändert, wird GC informiert und diese Objekte werden wieder *grau* markiert. Deshalb werden sie noch mal kontrolliert.
Solche Algorithmen brauchen immer noch eine Pause. In dieser Pause werden nur die letzten grauen Objekte besucht. Viele Objekte werden vorher schon markiert werden, deshalb ist diese Pause sehr kurz.
Nach der Markierungsphase werden die toten Objekte auch von diesem Prozess gelöscht.
Die heutigen Implementierungen haben leider ihre eigene Probleme:
1. Während GC über die geänderte Objekte informiert wird, werden Write Barriers benutzt. Diese bringen ihren eigenen Overhead, weil diese nach *jedem* Schreibzugriff benutzt wird. Heute ist es weniger problematisch[1], denn Write Barriers sind auch für die Funktion der virtuellen Maschinen nötig. Auch normale GCs brauchen diese. Es ist erwünscht, dass GCs mit kleinen Write Barriers funktionieren und man hat geschafft diese zu bis zur zwei Instruktionen zu verkleinern.
2. Für jedes Objekt wird zusätzlicher Speicher gebraucht.

3 GCs in der Praxis

Hier werden GC-Algorithmen in praktischen Programmierumgebungen (hauptsächlich in .Net Framework und JVM) zusammengefasst.

3.1 .Net

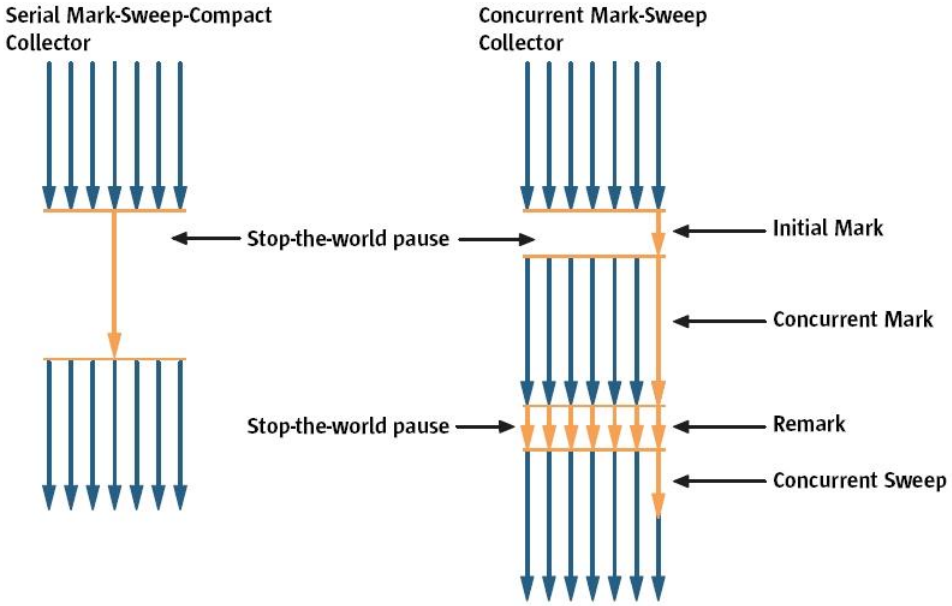
Bis .Net 4.0 gab es nur generationale GCs. Weitere Optimierungen für Effizienz soll existieren aber genaue Funktionsweisen sind nicht veröffentlicht.

3.1.1 Generational GC im .Net Framework

.Net Framework[10] hat normalerweise 3 Generationen aber das ist so gestellt, dass man sie später ändern kann. Die Generation, die die neueste Objekte erhält (Gen0), wird öfter besucht. Der Algorithmus behandelt die Generationen weniger unterschiedlich. Gen0 und Gen1 Objekte werden mit mark-and-sweep gelöscht und die Überlebende auf die nächste Generation befördert. Die Gen2 werden durch mark-and-compact-Verfahren durchgegangen. Hier werden die Überlebenden auch defragmentiert. Nur sehr große Objekte werden nicht defragmentiert, da diese zu bewegen zu viel kostet.

3.1.2 CGC in .Net 4.0

Mit .Net 4.0 wurde einen parallelen Concurrent GC (mit ähnlichen Eigenschaften und Problemen wie CGC in Java) eingeführt. Rein parallele GCs sind nicht vorgekommen in .Net aber dieser GC benutzt auch parallele Threads während der unvermeidlichen Stop-The-World-Pause (s.u.).



3.2 JVM

3.2.1 Generational GC in der JVM

Wie in .Net alle GCs in (standard)JVM arbeiten nach Prinzip der Generational GC. Die Standard-JVM hat 2 Generationen. In der JVM werden mehrere Arten von GCs angeboten. Sie versuchen meistens mit der ersten Generation anzufangen. Deshalb besuchen sie diese Generation öfter. Teilweise funktionieren Algorithmen nur auf eine Generation (siehe die Tabelle).

Algorithmus	Young Generation	Old Generation
Serieller GC	Ja	Ja (mark-and-compact)
Paralleler GC	Ja (mark-and-copy)	Nein. (mit seriellem GC)
Parallel Compacting Collector	Ja	Ja
Concurrent Collector	Nein	Ja
G1(s.u)	Ja	Ja

3.2.2 Parallele GCs in der JVM

Es gibt zwei parallele GCs in der JVM. *Parallel Compacting Collector* wurde als eine Verbesserung der älteren parallelen GC entwickelt. Diese Variante hat erfolgreich die allgemeine Laufzeit des GC verkürzt und hat weniger Nachteile als CGC. Daher ist es in der Praxis gut einsetzbar.

3.2.3 CGCs in der JVM

Die Forschungen der Firma Sun entwickeln *State of the Art* in diesem Thema weiter. Heutige CGCs in JVM und in .Net sind sehr ähnlich wie der Algorithmus, der in Paper [1] beschrieben wurde. Obwohl er Probleme hat (s.o), hat man dieser Algorithmus in diese virtuellen Maschinenen eingebaut und er ist eine Alternative zur seriellen und parallelen Algorithmen, die man wählen kann.

G1 (Garbage First) Der CGC in heutigen JVM läuft nur auf der alten Generation. Er hat noch keine Strategie zur Bereinigung der neuen Generation. Der neue Algorithmus G1, dessen Testversion veröffentlicht wurde, soll auf beiden Generationen laufen und wie der heutige CGC in JVM, Vorteile der Nebenläufigkeit und Parallelismus mitbringen. Außerdem soll der Reaktionszeit des GC parametrisiert werden können. Der vollständige Version soll mit Java 7 veröffentlicht werden[11][12][13].

4 Fazit

Ein großes Problem der modernen Programmiersprachen war der Bedarf an schnelle Speicherverwaltung. Garbage Collectors werden daher weiterentwickelt, um dieses Problem zu lösen.
Forschern ist es gelungen die Möglichkeit des Parallelismus auszunutzen um den GC-Algorithmus zu verbessern. Die Entwickler haben heute mehrere GC-Algorithmen in virtuellen Maschinen zu wählen. Es liegt daran, dass man noch keine ultimative Lösung gefunden hat.
Es gibt immer noch kein GC, der für jedes System geeignet ist. Man muss immer noch an Computern denken, die nicht Multicore sind und GCs für Echtzeitsystemen (Es gibt solche GCs aber hier wurden sie nicht genannt) müssen weitergeforscht werden. Nebenläufige GCs müssen auch perfektioniert werden. Die Idee ist gut aber die heutigen Implementierungen haben Probleme und deshalb werden CGCs in der Praxis nicht benutzt.

Die Forschungen gehen weiter und Sun arbeitet gerade an G1, der praktisch eine Verbesserung der CGC ist. Dieser Algorithmus verspricht auch schwache Echtzeitreaktionen, was auch eine Entwicklung für Echtzeit-GCs ist.

Literatur

[1] Tony Printezis and David Detlefs. A Generational Mostly-concurrent Garbage Collector. 2000

[2] Herbert Stoyan. Early Lisp History (1956-1959).
http://www8.informatik.uni-erlangen.de/html/lisp/histlit1.html

[3] Sun Microsystems. Memory Management in the Java HotSpot™ Virtual Machine. 2006
http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[4] J.D. Meier, Srinath Vasireddy, Ashish Babbar, and Alex Mackman. Improving .NET Application Performance and Scalability.
http://msdn.microsoft.com/en-us/library/ff647790(v=pandp.10).aspx

[5] H. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection

[6] Fridtjof Siebert. Hard Realtime Garbage Collection (Dissertation an Universität Karlsruhe), 2002

[7] Dijkstra et. al. On-the-Fly Garbage Collection: An Exercise in Cooperation 1978

[8] D Doligez, G Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. 1994

[9] Stuart A. Yeates and Michel de Champlain. Design Patterns in Garbage Collection. 1997

[10] Jeffrey Richter. Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework.
http://msdn.microsoft.com/de-de/magazine/bb985010(en-us).aspx

[11] Garbage First Garbage Collection, Deflefs et. al. 2008

[12] Garbage First GC
http://java.sun.com/javase/technologies/hotspot/gc/g1_intro.jsp

[13] G1 talk from JavaOne. 2008
http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-5419&yr=2008

[14] MSDN Library eintrag über Garbage Collection in .Net Framework 4:
http://msdn.microsoft.com/de-de/library/ee787088.aspx