Mark Compact & Performancemessungen GC

Bernhard Prügl, 0156212

***Abstract*:**
Wenn Programme länger laufen tritt nach Garbage Collection Zyklen ein fragmentierter Speicher auf. Diese Fragmentierung führt zu einer Reihe von Problemen. Mit Mark Compact Strategien versucht man diese Probleme zu lösen. In dieser Arbeit werden die Grundzüge dieses Garbage Collection Konzepts behandelt. Dabei gibt zahlreiche Algorithmen aus diesem Bereich, die am häufigsten verwendeten Systeme werden in dieser Seminararbeit behandelt. Insbesonders geht es sich um die Strategien Zwei Finger System, Adressweiterleitung, Tabellenunterstütze Systeme und Auffädelnde Systeme. Ein weiterer wichtiger Punkt bei der Garbage Collection ist die Auswahl des optimalen Collectors für das eingesetzte Programm. Dieser Punkt wird in Performancemessungen im zweiten Teil dieser Arbeit behandelt.

I. KEYWORDS

Memory management, Garbage Collection, Mark Compact, Performance

II. WARUM MARK - COMPACT:

Mark – Sweep- Algorithmen konzentrieren sich hauptsächlich auf das finden von nicht mehr verwendeten Objekten und deren Freigabe. Wenn das Programm allerdings länger läuft entsteht auf diese Weise ein fragmentierter Speicher. Je länger das Programm läuft umso mehr wird der Speicher fragmentiert.

Diese Fragmentierung wirft eine Reihe von Problemen auf:

- Es ist aufwendiger neue Objekte anzulegen. Das gilt sowohl für große als auch für kleine Objekte. Bei großen Objekten kann es passieren, dass ein neues Objekt nicht angelegt werden kann obwohl noch genügend freier Speicher vorhanden wäre. Das ist dann der Fall wenn der Speicher so fragmentiert ist, dass kein genügend großer zusammenhängender Bereich vorhanden ist. Im Gegensatz dazu steht das Anlegen kleiner Objekte. Hier gibt es das Problem, dass man zuerst suchen muss an welcher Stelle das neue Objekt optimal eingefügt werden kann. Verzichtet man aus Performancegründen auf diese Suche, wird der Speicher nicht optimal ausgenützt. Auch wenn es Strategien gibt um diese Probleme abzuschwächen bleibt die Fragmentierung beim Anlegen neuer Objekte immer ein Problem.
- Ein weiteres Problem ist, dass selbst mit der besten Strategie immer Lücken im Speicher zurückbleiben. Somit kann der Speicher nicht optimal ausgenützt werden.
- Als letzter Punkt kommt es bei fragmentiertem Speicher immer wieder vor, dass nacheinander allokierte Objekte nicht nebeneinander im Speicher stehen. Diesen Effekt nennt man auch schlechte Speicherlokalität. Untersuchungen haben gezeigt, dass nacheinander erzeugte Objekte oft gleichzeitig für eine bestimmte Operation gebraucht werden. Wenn diese Objekte dann nicht beieinander liegen entstehen öfter Cache Misses.

Um diese Probleme zu verhindern oder zu reduzieren wurden die Mark Compact Algorithmen entwickelt. Diese Algorithmen fassen die lebendigen Objekte am Anfang des Speichers zusammen. Diese Zusammenfassung von Speicher kostet zwar Zeit, sparen aber in der Regel bei der weiteren Programmausführung mehr Zeit wieder ein.

III. ÜBERBLICK ÜBER DIE ALGORITHMEN:

Es wurden zahlreiche Algorithmen zur kompaktierung des Speicher entwickelt. Alle diese Algorithmen machen mehrere Durchläufe durch den Speicher. Die Anzahl der benötigten Durchläufe ist je nach Algorithmus unterschiedlich. Man kann jedoch grundsätzlich zwischen folgenden drei Phasen unterscheiden:

1. Markieren der lebendigen Knoten.
2. Kompaktierung des Speichers durch verschieben der Knoten.
3. Aktualisieren der Zeiger auf Knoten die verschoben wurden.

In manchen Algorithmen kann Punkt 2. (Knoten verschieben) und Punkt 3. (Zeiger aktualisieren) auch vertauscht sein. In dieser Arbeit werden die Methoden behandelt mit denen Punkt 2. und 3. ausgeführt werden.

A. Einteilung:

Die wichtigste Einteilung ist sicherlich wie die verschobenen Knoten angeordnet werden. Man unterscheidet hier grundsätzlich drei Arten:

- Zufällige Kompaktierung: Hier werden die einzelnen Speicherblöcke ohne Rücksicht auf ihre ursprüngliche Anordnung verschoben, wodurch natürlich die relative Anordnung der Knoten verloren geht. Diese Algorithmen sind schnell und einfach, führen allerdings im Nachhinein meist zu einer langsameren Programmausführung.
- Gleitende Kompaktierung: Hier wird die relative Anordnung der Speicherblöcke erhalten. Blöcke die vor der Kompaktierung Nachbarn waren sind auch nachher wieder benachbart, lediglich eventuell vorhandene Lücken werden entfernt
- Linearisierende Kompaktierung: Hier werden, so gut als möglich, Speicherblöcke zusammengeführt die Zeiger aufeinander enthalten. Dadurch erhofft man sich eine bessere Ordnung.

Weiters kann man die Algorithmen danach einteilen ob zwei oder drei Durchläufe für die Kompaktierung des Speichers benötigt werden.

Auch der zusätzlich benötigte Speicher ist ein Kriterium für die Algorithmen. Zusätzlicher Speicher wird zum Beispiel für Zeiger benötigt die alte oder neue Adressen speichern müssen.

Es gibt auch noch Algorithmen die bestimmte Zusatzanforderungen stellten. Zum Beispiel kann es sein, dass der Algorithmus nur mit Knoten von einer Größe arbeiten kann. Andere Algorithmen stellen gewissen Mindestgrößen an Header oder Knoten.

B. Die Algorithmen:

In dieser Arbeit werden 4 Algorithmustypen behandelt, immer Anhand eines Beispiels aus dieser Gruppe.

- Zwei Finger Algorithmus: Es werden zwei Zeiger verwendet, einer davon zeigt auf die nächste freie Speicherzelle, der andere sucht die zu verschiebenden Knoten. Beim verschieben der Knoten wird die neue Adresse in der alten Speicherzelle zurückgelassen damit die Zeiger aktualisiert werden können. Diese Algorithmen funktionieren nur mit fix vorgegebenen Knotengrößen. Aus dieser Gruppe wird Edward's Zwei Finger Algorithmus vorgestellt.
- Adressen weiterleitende Algorithmen: Die neuen Adressen werden in ein zusätzliches Feld geschrieben das sich im Header der Knoten befindet. Anhand dieser Werte können jetzt die Zeiger aktualisiert werden und anschließend die Knoten verschoben werden. Aus dieser Gruppe wird der Lisp 2 Algorithmus vorgestellt.
- Tabellenbasierte Algorithmen: Es wird eine Tabelle mitgeführt, welche die Informationen über die Verschiebungen enthält. Anhand dieser Tabelle, welche in den vorhandenen Speicherlöchern gespeichert werden kann, können die neuen Zeigerwerte berechnet werden. Der später behandelt Haddon-Waite Algorithmus stammt aus dieser Gruppe.
- Threading Algorithmen: An jeden Knoten wird eine Liste von Zeigern die ursprünglich an den Knoten zeigten gehängt. Wird der Knoten dann verschoben so wird diese Liste durchlaufen und alle Zeiger in der Liste aktualisiert. Jonkers Algorithmus wird als Vertreter von diesem etwas komplexeren Verfahren behandelt.

IV. Zwei Finger Algorithmus:

Der einfachste vorgestellte Algorithmus ist der Zwei Finger Algorithmus. Der Algorithmus stammt von Edwards Saunders [Sau64]. Er verwendet 2 Durchläufe um den Speicher zu kompaktieren. Dazu werden im ersten Durchlauf alle Knoten in die untere Hälfte geschoben und im zweiten Durchlauf werden die Pointer korrigiert.

Weitere Vorraussetzung ist, dass alle Knoten im Speicher die gleiche Größe haben. Nach der Kompaktierung ist die Reihenfolge wie die Knoten im Speicher liegen verändert. Vor der Ausführung des Algorithmus müssen die lebendigen Knoten markiert und die Anzahl der lebendigen Knoten gezählt sein.

1) Algorithmus:

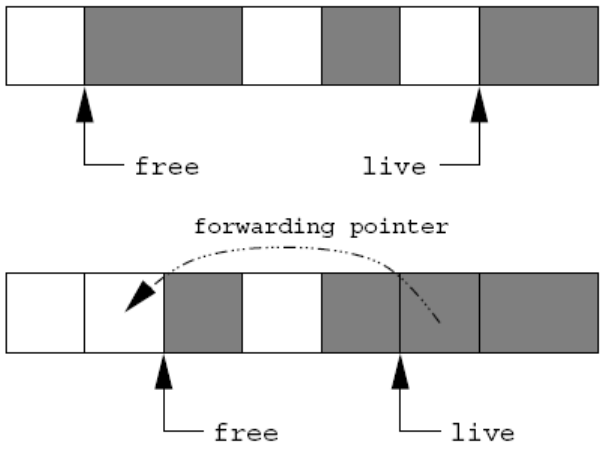


Abbildung 1: Zwei Finger Algorithmus

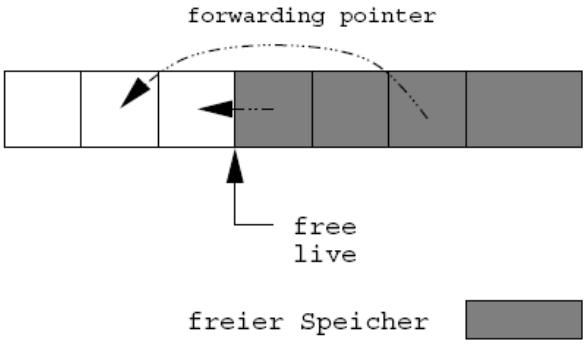


Abbildung 2: Zwei Finger Algorithmus

Für den ersten Durchlauf werden zwei Zeiger verwendet. Der Zeiger "free" sucht immer die erste freie Stelle im Speicher und läuft dabei vom Anfang zum Ende. Der Zeiger "live" startet am Ende des Speichers und sucht in der entgegengesetzten Richtung nach lebendigen Knoten. Sobald "live" einen lebendigen Knoten gefunden hat wird dieser Knoten an die Stelle verschoben welche "free" markiert hat. Anschließend wird die neue Adresse vom verschobenen Knoten "live" in seiner alten Speicherzelle eingetragen. Da die Daten ja schon verschoben wurden kann hier einfach die neue Adresse über die User Daten geschrieben werden. Danach beginnt die Suche von "free" und "live" wieder von neuen bis sich die Zeiger "free" und "live" in der Mitte treffen. Nach diesem Vorgang ist der Speicher partitioniert, unterhalb von "free" befinden sich die lebendigen Knoten, oberhalb ist freier Speicher

Im 2.ten Durchlauf werden noch einmal alle lebendigen Knoten vom Anfang bis Ende durchlaufen. Findet der Algorithmus dabei einen Zeiger der auf einen Knoten zeigt der eine höhere Adresse die Anzahl der lebendigen Knoten hat so weis man, dass dieser Knoten verschoben wurde. Dieser Zeiger wird nun aktualisiert indem die im ersten Durchlauf an der alten Speicherposition hinterlassene Adresse eingetragen wird. Knoten ohne Zeiger oder mit Zeigern die auf Adressen kleiner als die Anzahl der lebendigen Knoten sind müssen nicht mehr weiter bearbeitet werden.

2) Analyse:

Der Algorithmus ist hervorragend obwohl er so einfach und alt ist. Die Komplexität ist linear mit der Größe des Speichers und er benötigt nur 2 Durchläufe um den Speicher zu kompaktieren. Der Aufwand der in jedem Knoten betrieben ist minimal. Da die neuen Adressen einfach im Speicher hinterlassen werden können benötigt er auch keinen zusätzlichen Speicher. Die Zeiger können auch auf interne Wörter in den Knoten zeigen, was andere Algorithmen eventuell nicht können.
Der große Nachteil dieses Algorithmus ist, dass die relative Ordnung zwischen den Knoten zerstört wird. Deshalb ist er auch nicht geeignet, wenn die Kompaktierung die Performance des Programms verbessern soll. Ein weiterer Nachteil ist das er nur Knoten einer fixen Größe bearbeiten kann. Dieses Problem lässt sich etwas entschärfen, indem man den Speicher in Regionen für verschiedene Knotengrößen einteilt und den Algorithmus dann auf die einzelnen Regionen des Speichers separat anwendet.

V. Lisp 2 Algorithmus (Adressenweiterleitend):

Der Lisp 2 Algorithmus ist ein Beispiel für einen Adressenweiterleitenden Algorithmus. Er benötigt einen zusätzlichen Speicher in Zeigergröße im Header jedes Knoten. Neben dieser Funktion kann dieser Zeiger nur noch zum markieren der lebendigen Knoten benutzt werden (z.B. durch den Eintrag von null verschiedener Werte). Die Aktualisierung der Zeiger funktioniert hier durch einen Eintrag der zukünftigen Adresse im zusätzlichen Feld im Header der Knoten. Im Gegensatz zum vorherigen Algorithmus benötigt er 3 Durchläufe durch den Speicher, ist aber trotzdem sehr schnell. Dafür kann er aber auch problemlos Knoten verschiedener Größe bearbeiten.

1) Algorithmus:

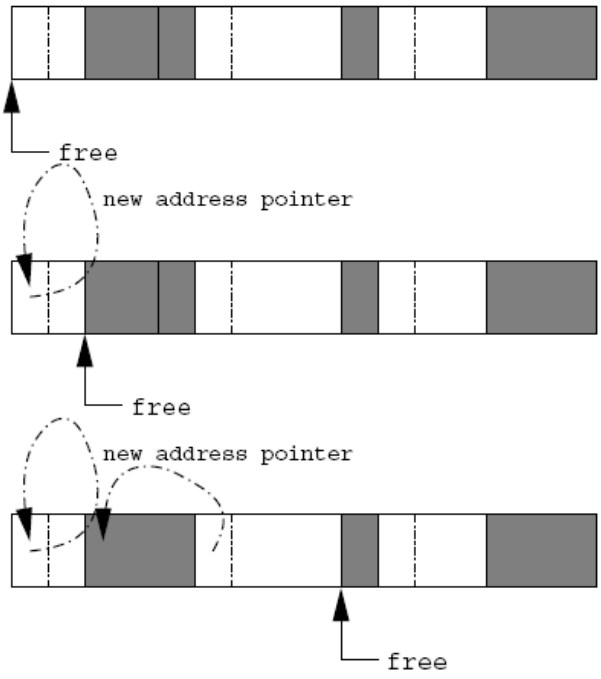


Abbildung 3: Lisp 2 Algorithmus

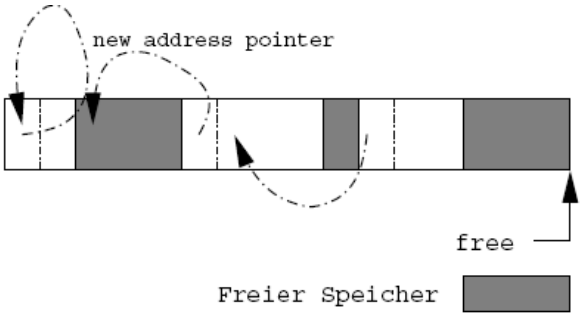


Abbildung 4: Lisp 2 Algorithmus

Im ersten Durchlauf werden die neuen Adressen der Knoten berechnet. Dabei durchläuft ein "free" Zeiger den Speicher von Anfang bis Ende. Für jeden lebendigen Knoten wird die neue Adresse in das "forwarding address" Feld geschrieben. Dieser Wert berechnet sich aus der Summe der Größe der bisherigen lebendigen Knoten. Zusätzlich können in diesem Durchlauf beieinander liegende Garbage-Knoten zu einem einzigen großen Knoten zusammengefasst werden, damit die Geschwindigkeit der folgenden Durchläufe verbessert wird.

Im zweiten Speicherdurchlauf werden die internen Zeiger aktualisiert. Die neuen Werte werden dabei mithilfe des "forwarding address" Zeigers berechnet. Außerdem wird bei Bedarf der Wurzelzeiger aktualisiert.

Im dritten finalen Durchlauf werden die Knoten tatsächlich an ihre, schon im ersten Durchlauf berechnete, neue Adresse verschoben. Dabei wird auch das "forwarding address" Feld gelöscht. Am Ende dieses Durchlaufs zeigt der "free" Zeiger an den Beginn des freien Speichers und links vom "free" Zeiger sind alle lebendigen Knoten gesammelt.

2) Analyse:

Der Lisp 2 Algorithmus erhält im Gegensatz zu dem vorherigen Algorithmus die relative Anordnung zwischen den Knoten. Diese Erhaltung der relativen Anordnung kostet allerdings ein zusätzliches Zeigergroßes Feld im Speicher. Außerdem muss der Speicher 3 mal durchlaufen werden. Trotz dieses zusätzlichen Durchlaufs ist der Algorithmus aber sehr schnell und hat ebenso wie der Zwei Finger Algorithmus asymptotisch lineare Laufzeit.

VI. HADDON-WAITE ALGORITHMUS (TABELLENBASIERT):

Der Zwei Finger Algorithmus braucht keine zusätzlichen Speicher, zerstört allerdings die relative Anordnung der Knoten zueinander. Der Lisp 2 Algorithmus erhält die relative Anordnung um den Preis eines zusätzlichen Zeiger Feldes in jedem Knoten. Tabellenbasierte Algorithmen erhalten die relative Anordnung ohne zusätzlichen Speicher zu verwenden. In einer Tabelle werden die Informationen über die neuen Positionen der einzelnen Knoten gespeichert.

Diese Tabelle wird direkt im Speicher gespeichert und bei Bedarf ausgehend vom Beginn des Speichers Richtung Ende verschoben. Einzige Vorraussetzung für den Algorithmus ist also, dass der kleinste Knoten mindestens zwei Adressen enthalten kann damit die Tabelle in den entstehenden Lücken gespeichert werden kann.

1) Algorithmus:

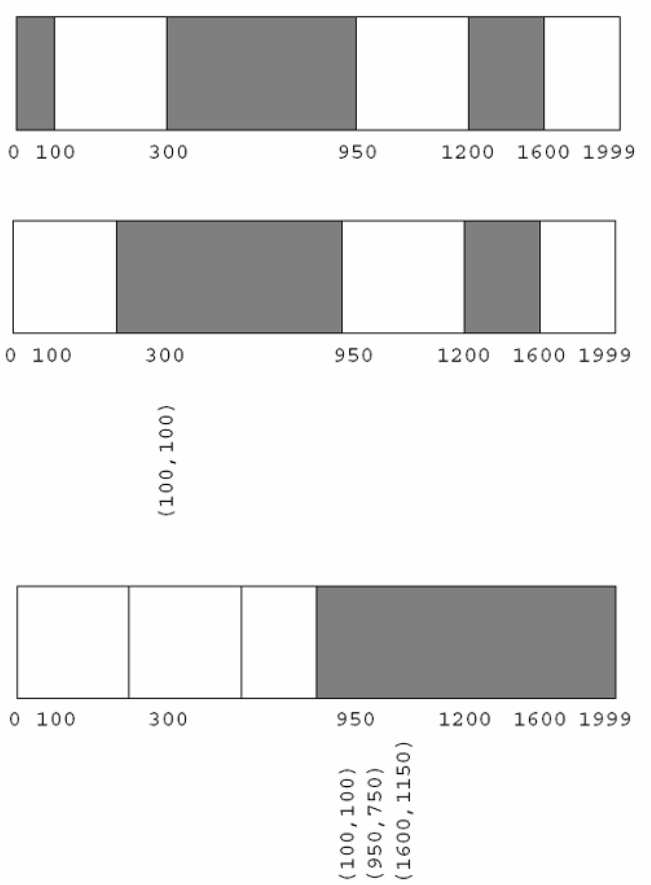


Abbildung 5: Haddon-Waite Algorithmus

Im ersten Durchlauf wird der Speicher vom Anfang bis zum Ende durchlaufen. Trifft der Algorithmus dabei auf einen lebendigen Knoten so wird dieser an seine neue Position verschoben, die sich genauso wie beim Lisp 2 Algorithmus aus der Summe der bisherigen lebendigen Knoten berechnet ist. Anschließend wird die Information über diese Verschiebung in die Tabelle eingetragen. Ein Eintrag in der Tabelle besteht aus der alten Startadresse des verschobenes Blockes und der Differenz um wie viel der Block nach vorne verschoben wurde. Die Tabelle wird dabei an das Ende der schon bearbeiteten Knoten geschrieben. Wenn ein neuer Knoten verschoben wird so wird die Tabelle bei Bedarf nach hinten verschoben. Dadurch, dass der kleinste Knoten die Größe 2 hat ist, immer genug Platz für die Tabelle vorhanden. Das Verschieben der lebendigen Knoten und das erzeugen der Tabelleneinträge wird jetzt solange fortgesetzt,

bis sich alle lebendigen Knoten am Anfang des Speichers befinden.

Durch das Verschieben der Tabelle kann es vorkommen das die Tabelle am Ende unsortiert ist. Deshalb muss sie am Ende sortiert werden. Dieses Sortieren hat theoretisch eine asymptotische Laufzeit von O(n.log(n)). In der Praxis ist dieser Aufwand jedoch meistens wesentlich kleiner. Alternativ besteht auch noch die Möglichkeit für jeden Eintrag in der Tabelle anzugeben an welcher Position er stehen soll wodurch die asymptotische Laufzeit auf O(n) sinkt. Dazu muss allerdings dann die Mindestgröße pro Knoten erhöht werden, da ein solcher Tabelleneintrag dann mehr Platz braucht.

Der zweite Durchlauf wird zum Aktualisieren der Zeiger verwendet. Trift der Algorithmus hier auf einen internen Zeiger so werden per binärer Suche in der Tabelle 2 Einträge (a,s) und (a', s') gesucht, so dass die Adresse p des Ziels zwischen a und a' liegt. Die neue Zieladresse des Ziels ist dann p – s, es wird also der Zeiger um den gleichen Wert reduziert wie der ursprüngliche Block verschoben wurde. Auch dieser Vorgang ist von der Ordnung O(n.log(n)). Befindet sich nach der Tabelle mit den Verschiebeinformationen noch genügend Platz, so kann danach eine Hash Tabelle erstellt werden um die Suche zu beschleunigen. Messungen haben gezeigt, dass diese Hash Tabelle ca. doppelt so groß als die ursprüngliche Tabelle sein sollte.

2) Analyse:

Der Haddon-Waite Algorithmus benötigt keinen zusätzlichen Speicher und erhält auch noch die relative Ordnung. In diesen Punkten ist er den zwei vorherigen Algorithmen klar überlegen. Allerdings erkauft man sich diese Vorteile durch eine längere Laufzeit von O(n.log(n)). Bei größeren Knoten kann man allerdings durch die Verwendungen eines dritten Wertes pro Tabelleneintrag (Positionseintrag für die leichtere Sortierung) und das Erstellen einer Hash Tabelle am Schluss fast immer eine Laufzeit von O(n) erreichen. Aber auch in diesem optimalen Fall ist die Laufzeit auch mit nur zwei Durchläufen im Vergleich zu den zwei Vorgängern am längsten. Das ist deshalb der Fall weil der Aufwand pro Durchlauf doch relativ hoch ist.

VII. JONKERS ALGORITHMUS (THREADED):

Das Problem beim updaten des Speichers ist, dass man alle Zeiger suchen muss, und sie so anpassen muss das sie auf die neue Adresse der verschobenen Knoten zeigen. Alle bisher vorgestellten Algorithmen durchsuchen dazu den Speicher nach Zeigern und berechnen dann den neuen Wert für die Zeiger. Es gibt allerdings noch eine andere Möglichkeit wie man den Zeigern ihren neuen Wert zuweist. Dieser Vorgang nennt sich Threading und wurde von David Fischer [Fis74] vorgestellt.

1) Threading:

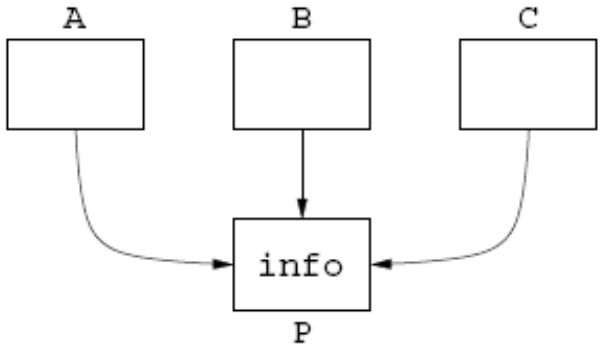


Abbildung 6: Threading

Am Beginn zeigen die Zeiger (hier A,B und C) auf das Info Feld. Damit beim Aktualisieren der Zeiger nicht jeden lebendigen Knoten nach Zeiger durchsuchen zu müssen werden die Zeiger folgendermaßen umgeordnet. Da alle Zeiger auf P von P aus gefunden werden sollen werden die Knoten so aufgefädelt, dass sie in einer verknüpften Liste ausgehend von P hängen. Dieser Vorgang wird "Threading" oder Auffädeln genannt.

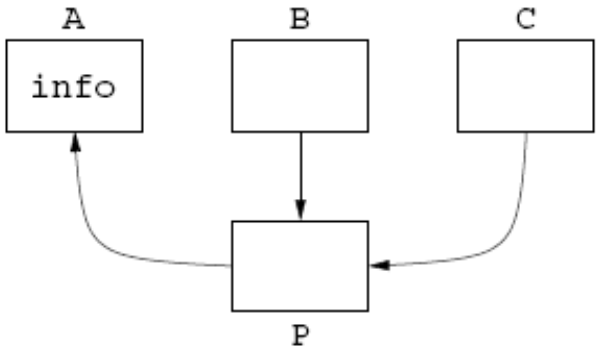


Abbildung 7: Threading

Beim Threading geht man folgendermaßen vor. Es werden die Knoten der Reihe nach untersucht. Als erstes findet der Algorithmus den Knoten A. Indem die Inhalte von A und P vertauscht werden wird der Knoten aufgefädelt. Jetzt enthält also A die Information in P, während P einen Zeiger auf A enthält. Da aus den Vorraussetzungen hervorgeht das Header und Zeiger verschiedene Werte enthalten müssen, weiß man jetzt welche Knoten bereits aufgefädelt sind. Als nächstes wird der Knoten B gefunden. Jetzt tauscht man wider die Information zwischen B und P aus und somit enthält B einen Zeiger auf A und P einen Zeiger auf B. Als letztes wird in

diesem Beispiel der Knoten C aufgefädelt so P dann einen Zeiger auf C enthält und C auf B zeigt. Die somit entstandene Liste ist in der gezeigten Abbildung zu sehen.

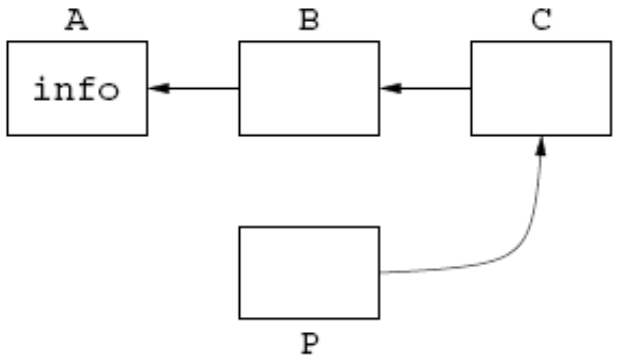


Abbildung 8: Threading

Sobald jetzt die neue Adresse von P bekannt ist kann die Liste durchlaufen werden und jeder Zeiger wieder auf neue Adresse von P gerichtet werden. Am Ende dieses Durchlaufs kann die Information wieder auf P zurück geschrieben werden, da sie ja im letzten Knoten der Liste, nämlich A, liegt.

2) Algorithmus:

Für den Algorithmus von Jonkers [Jon79] werden aufgrund des verwendeten Threadings folgende drei Bedingungen angegeben.

- Zeiger dürfen nur auf den Header eines Knoten zeigen.
- Header müssen groß genung sein um einen Zeiger enthalten zu können.
- Header dürfen nur Werte enthalten die von Zeigern verschieden sind.

Nach dem Markieren der lebendigen Knoten werden noch zwei Durchläufe zur Kompaktierung des Speichers benötigt. Der erste Durchlauf bearbeitet die Zeiger die vorwärts auf andere Knoten zeigen und weist ihnen den neuen Wert von P zu. Der zweite Durchlauf aktualisiert die rückwärts zeigenden Zeiger und verschiebt die Speicherblöcke.

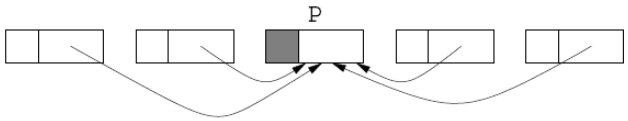


Abbildung 9: Jonkers Algorithmus

Anhand eines herausgegriffen Knoten P soll nun der Algorithmus erklärt werden. Ausgehend von der gezeigten Abbildung werden am Beginn alle Knoten aufgefädelt bis der Knoten P erreicht ist. Dabei wird der Speicher vom Anfang bis zum Ende durchlaufen und jeder Zeiger auf den Algorithmus trifft wird wie vorher beim threading beschrieben in der entsprechenden Liste aufgefädelt. Dabei wird die Größe der bis jetzt erreichten lebendigen Knoten wird in der Variable "free" mitgeschrieben.

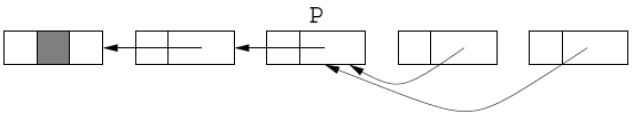


Abbildung 10: Jonkers Algorithmus

Erreicht der Algorithmus jetzt den Beispielknoten P so können alle bis jetzt aufgefädelten Knoten aktualisiert werden. Dazu wird in alle aufgefädelten Zeiger der Wert von "free" geschrieben und die Information die an den Beginn der Liste verschoben wurde kommt wieder in den Knoten P. Nun zeigen alle vorwärtszeigenden Zeiger auf das zukünftige P.

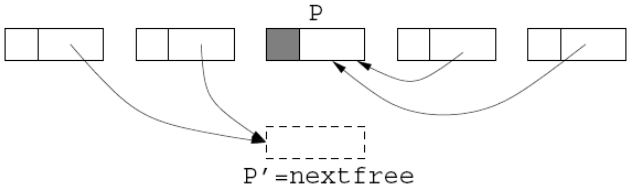


Abbildung 11: Jonkers Algorithmus

Zum Abschluss des ersten Durchgangs werden alle rückwärtszeigenden Zeiger auf P aufgefädelt. Referenzen auf den Speicherblock selber zählen dabei als rückwärtszeigende Zeiger.

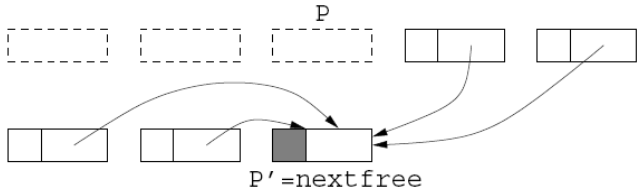


Abbildung 12: Jonkers Algorithmus

Im zweiten Durchlauf werden die rückwärtszeigenden Zeiger aktualisiert und die Knoten verschoben. Dazu wird bei jeder Verschiebung eines Knoten überprüft ob Zeiger darauf gethreaded sind. Sind solche vorhanden werden vor der Verschiebung die Zeiger mit der neuen Adresse des Knoten P aktualisiert. Auch bei diesen Algorithmus gilt, dass die Geschwindigkeit verbessert werden kann indem im ersten Durchlauf freie Knoten zusammengefasst wurden.

3) Analyse:

Der Algorithmus von Jonkers vereint die Vorteile der vorherigen 3 Algorithmen. Er benötigt nur 2 Durchläufe, erhält die relative Ordnung der Knoten, hat lineare Laufzeit und benötigt keinen zusätzlichen Speicher.
Diese Vorteile erkauft man sich aber durch eine hohe Komplexität bei den einzelnen Durchläufen. Jeder lebendige Knoten muss dreimal vom Algorithmus bearbeitet werden. Zuerst beim aktualisieren der vorwärtszeigenden Zeiger, dann beim bearbeiten der rückwärtszeigenden Zeiger und schließlich noch beim Threading.

sehr viele Knoten verändert was das Risiko für Cache-Misses erhöht.

Generell kann man sagen, dass es oft empfehlenswert ist Mark Compact mit anderen Algorithmen zu kombinieren damit man nicht bei jeder Garbage Collection eine Kompaktierung machen zu müssen. Es gibt Heuristiken, welche vorschlagen, wann zusätzlich zu einer Garbage Collection auch eine Kompaktierung gemacht werden sollte.

VIII. Zusammenfassung der Algorithmen:

Algorithmus	Art	Knotengröße	Durchläufe	Zus. Speicher	Laufzeit
Zwei-Finger	Zufällig	Fest	2	Keiner	O(M)
Lisp 2	Gleitend	Beliebig	3	1 Zeiger pro Knoten	O(M)
Tabellenbasiert	Gleitend	Beliebig	2	Keiner	O(M.logM)
Threaded	Gleitend	Beliebig	2	Zeigergroße Header	O(M)

Geeigneter Algorithmus muss entsprechend Vorgaben gewählt werden.

Der Zwei Finger Algorithmus ist zwar sehr schnell und einfach, hat allerdings den großen Nachteil, dass die Reihenfolge der Knoten verändert wird. Außerdem ist die Knotengröße fix. Kann man diese Vorgaben nicht realisieren oder bedeuten sie zuviel Einschränkungen oder Aufwand muss ein anderer Algorithmus gewählt werden. Der Lisp 2 Algorithmus verschiebt die Knoten gleitend und das, trotz seiner drei Durchläufe deutlich schneller als die Tabellenbasierten und Threaded Algorithmen da alle drei Durchläufe relativ einfach sind. Dafür muss man allerdings einen zusätzlichen Zeiger pro Knoten in Kauf nehmen. Bei vorwiegend kleinen Knoten wird dieser Algorithmus dann auch nicht optimal sein. Tabellenbasierte Algorithmen kommen mit zwei Durchläufen aus und brauchen keinen zusätzlichen Speicher. Diesen Vorteil erkauft man sich aber mit zusätzlicher Laufzeit für das Sortieren der Tabelle und die Suche darin. Threaded Algorithmen schaffen die Kompaktierung mit zwei Durchläufen bei einer Laufzeit von O(n). Allerdings sind hier wieder Einschränkungen im Header in Kauf zu nehmen. Ist es nicht möglich Zeiger eindeutig von Nicht-Zeiger zu unterscheiden kann dieser Algorithmus nicht verwendet werden. Außerdem werden hier in den Schritten wo gethreadede Knoten aufgelöst werden

IX. PERFOMANCE VON GC:

Im zweiten Teil der Arbeit geht es um die Performance der einzelnen Garbage Collection Strategien. Die Arbeit baut

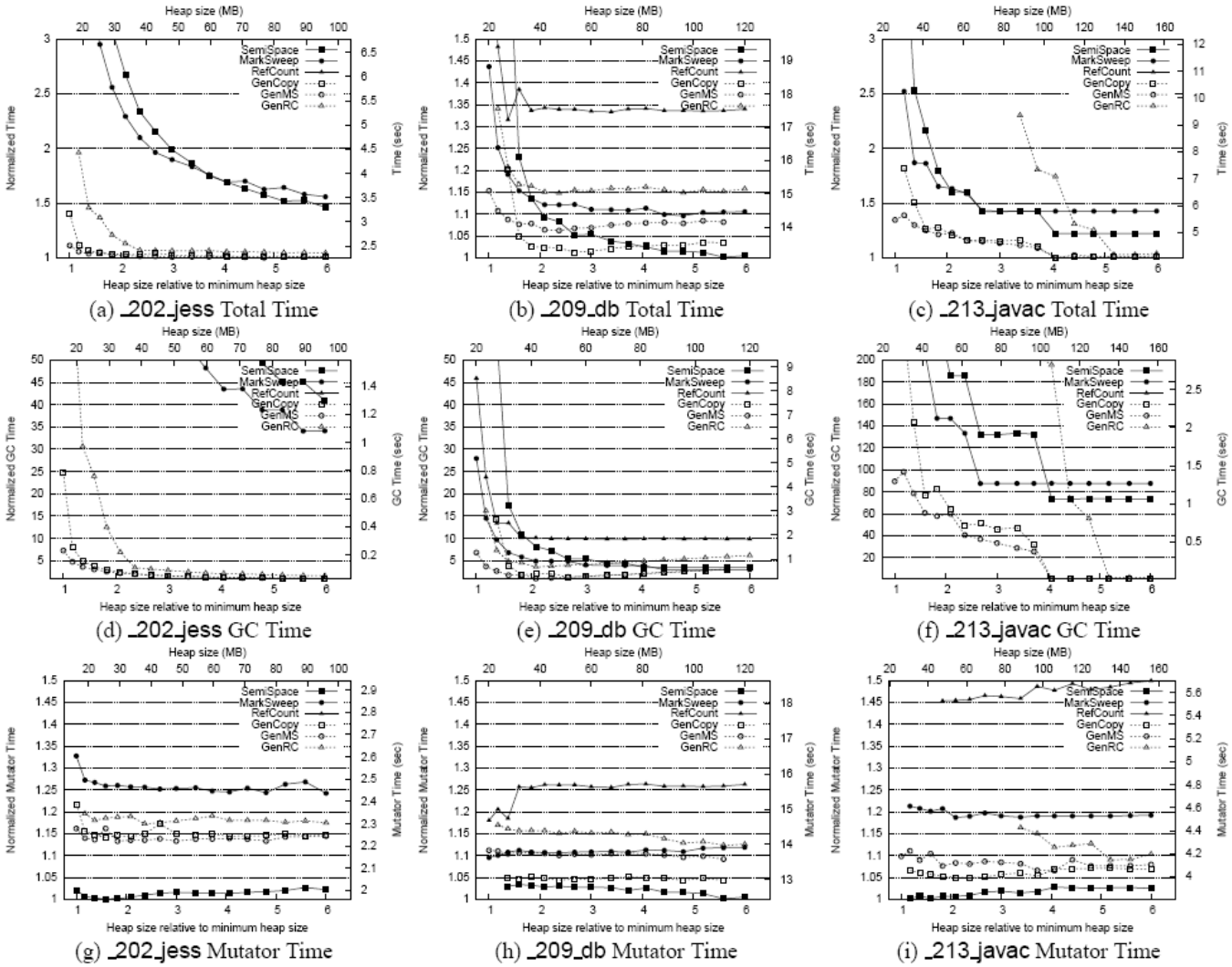
dabei auf die Messungen von Stephen M Blackburn, Perry Cheng und Kathryn S McKinley in [Bla04] auf. Diese Messungen wurden 2004 erstellt.

Behandelt werden die drei grundsätzlichen Strategien "copying semi-space", "mark-sweep" und "reference counting". Für alle diese drei Strategien wurde zum Test sowohl ein Collector der über den ganzen Heap läuft, als auch ein Generationen System eingesetzt. Alle Tests wurden mit MMTk einen Java Speicherverwaltungswerkzeug durchgeführt. Durchgeführt wurden die Messungen auf einen 1,9 GHzAMD Athlon XP 2600+. Der L1 Cache des Prozessors ist 64KB groß, der L2 Cache ist 512 KB groß. Der Athlon war bestückt mit 1GB dual channel Speicher mit 333MHz.

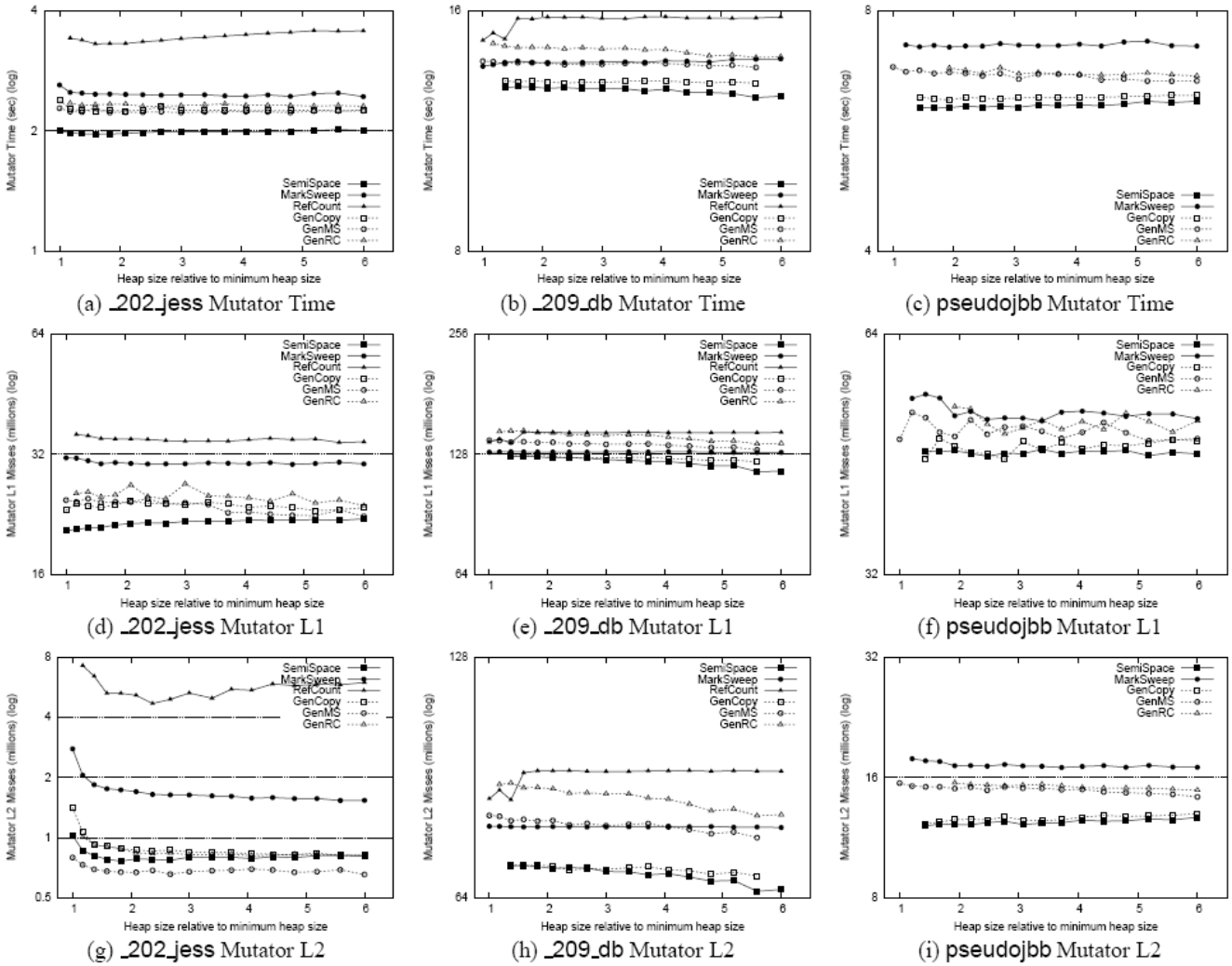
X.

A. Laufzeitmessungen komplettes Programm und GC:

In dieser Graphik kann man alle 6 Algorithmentypen vergleichen. Es zeigt sich, dass ein Generationensystem fasst immer eine deutliche Besserung des Laufzeitverhaltens mit sich bringt. Außerdem fällt sofort auf das Reference Counting immer deutlich schlechter ist als die beiden anderen Systeme, oft ist es sogar so schlecht das es nicht einmal mehr Platz findet in der Graphik. Generell kann man sagen, dass meist entweder das Generationensystem mit einem Copying Verfahren oder jenes mit Mark Sweep die optimale Lösung ist.
Weiters kann man in dieser Graphik sehen, dass eine Vergrößerung des Heaps bis zu einen gewissen Punkte eine deutliche Reduzierung der Anzahl der GC Durchläufe und auch der gesamten GC mit sich bringt. Dieser Zusammenhang ist aber nicht notwendigerweise eine schöne Kurve sondern kann, wenn die GC ungünstig fällt sogar wieder in die Gegenrichtung gehen.

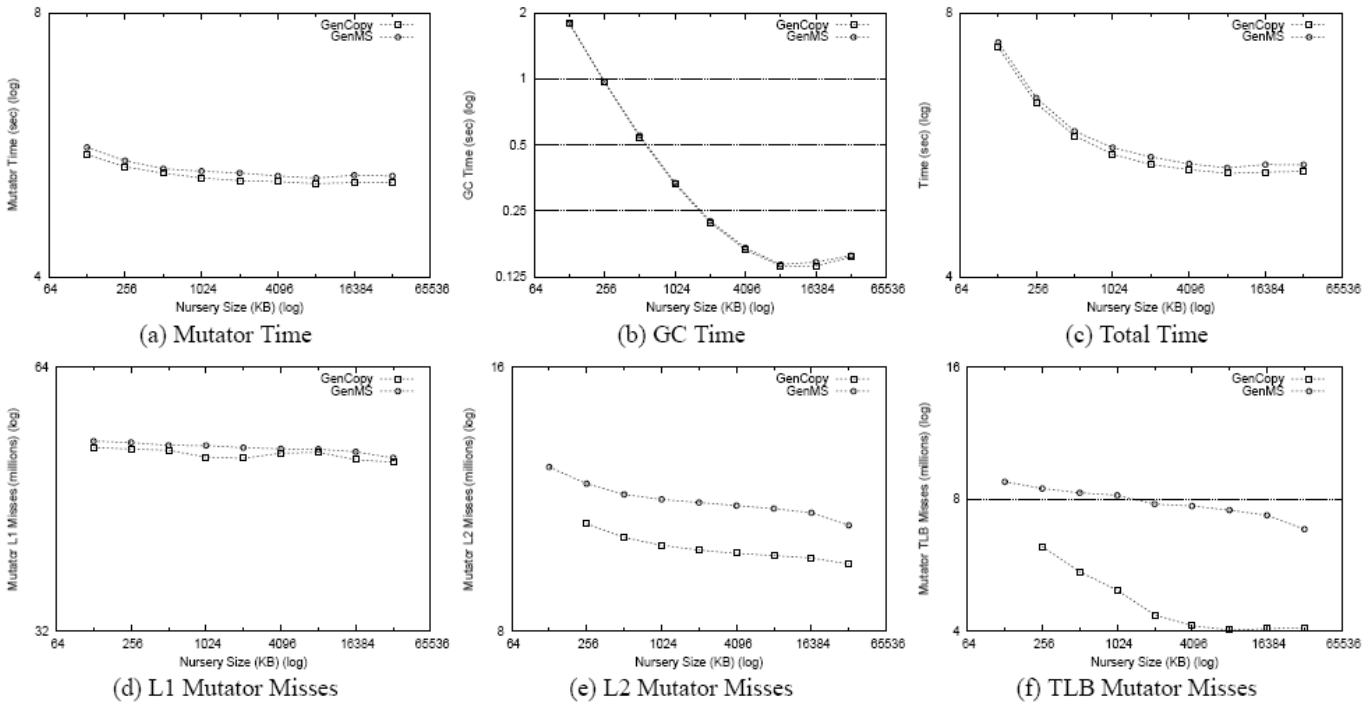


B. Messungen Cache Misses:



In dieser Graphik sieht man die Cache Misses bei den einzelnen GC Strategien. Dabei ist die Achse mit den Cache Misses logarithmisch. Im Vergleich zwischen Semi Space und Mark Sweep sieht man, dass Semi Space hier immer deutlich weniger Cache Misses hat. Reference Counting ist auch hier der klare Verlierer. Die Generationensysteme reihen sich ca. in der Mitte zwischen Semi Space und Mark Sweep ein. Auch das Reference Counting System ist hier nicht deutlich schwächer. Eine Besonderheit sind die L1 Cache Misses, hier wird mit dem Generationensystem von MarkSweep die beste Trefferrate erreicht. Mann muss allerdings bei diesen Graphiken beachten, dass nur die Laufzeit des Mutators gemessen wurde, also die GC noch nicht eingerechnet ist. Bezieht man auch die GC Zeit ein so erreichen die Generationensystem schnell wieder eine bessere Performance als das Semi Space Verfahren.

C. Performancemessungen in Generationensystemen:



In dieser Graphik sieht man den Einfluss der Aufteilung des Speichers zwischen alten und neuen Objekten zeigt. Es hat sich ja in den vorherigen Graphiken gezeigt, dass die Systeme MarkSweep und Copying, beide mit Generationensystem die beste Performance erreichen. Jetzt soll noch ermittelt werden wie groß der Speicher für die neuen Objekte optimal sein soll. Der Heap wurde dabei relativ groß mit 900MB angenommen und der Speicher für neue Objekte von 128KB bis 32MB skaliert. Hier besteht wieder logarithmischer Zusammenhang.

Die Graphik links oben zeigt eine immer weitere Verbesserung in der Mutator Zeit. Allerdings überwiegt ab einer gewissen Größe die GC Zeit. Es stellt sich also heraus, dass es so um die 4 bis 8 MB einen optimalen Punkt gibt. Das ist auch deutlich zu sehen in der Graphik über die Gesamtzeit rechts oben.

XI. REFERENZEN

[Jon96], Jones, R, Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Kapitel 5 John Wiley 1996

[Bla04], Stephen M. Blackburn, Perry Cheng, Kathryn S. McKinley: Myths and realities: the performance impact of garbage collection. ACM SIGMETRICS Performance Evaluation Review 32(1), 2004

[Hicks97], Michael W. Hicks, Jonathan T. Moore, Scott M. Nettles: The measured cost of copying garbage collection mechanisms. ACM SIGPLAN conference on Functional programming, SIGPLAN Notices 32(8)

[WiDe06], http://de.wikipedia.org/wiki/Garbage_collection

[WiEn06], http://en.wikipedia.org/wiki/Garbage_collection

[Reg02], Jens Regensburg jens@ps.uni-sb.de Mark – Compact Garbage Collection www.ps.uni-sb.de/courses/gc-ws01/exams/MarkCompact.pdf

[Fis74], David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. Information Processing Letters, 3(1):25–32, July 1974.

[Had67], B. Haddon. Waite: A compaction procedure for variable length storage elements, 1967.

[Jon79], H. B. M. Jonkers. A fast garbage compaction algorithm. Information Processing Letters, 9(1):25–30, July 1979.

[Sau74], R. Saunders. The lisp system for the q–32 computer, 1974.

XII. BIOGRAPHIE

Bernhard Prügl wurde am 16 September 1980 in Linz geboren. Er maturierte 2000 an der HTBLA für Elektronik / Zweig Informatik in Steyr. Seit 2001 macht er ein Masterstudium für Informatik an der Kepler Universität Linz.