CONTENTS

”Hard Realtime Garbage Collection” - ein Überblick

Rainer Kelz

Abstract— **Sichere Speicherverwaltung durch einen Garbage-Collector hat ihren Preis. Bezahlt wird dieser durch Erschwerung der Vorhersagbarkeit des Programmablaufs. Echtzeitumgebungen sollten jedoch bis ins kleinste Detail nachvollziehbar sein. Nur so kann man gültige Aussagen über zeitliche Garantien treffen. Diese Arbeit beschäftigt sich allgemein mit einer möglichen Lösung des Problems der Vorhersagbarkeit von Garbage Collection für objektorientierte Sprachen und versucht einen Überblick über echtzeittaugliche Garbage Collection Techniken zu geben, und im speziellen mit einer Lösung für die Programmiersprache Java, in Form einer hart echtzeit-geeigneten virtuellen Maschine, der JamaicaVM.**

I. Einleitung

Es gibt viele verschiedene Definitionen von Echtzeit-Systemen. Manche davon weniger ausführlich, manche sogar irreführend. Im folgenden seien einige dieser Definitionen aufgeführt:

1) ”The correctness of a real-time system depends not only on the logical results of the computation, but also on the time at which the results are produced.” [BUW90]
2) ISO 44300 definiert ”Echtzeitbetrieb” als Betriebsmodus eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind und Ergebnisse innerhalb einer definierten Zeitspanne verfügbar sind. Kann das System die zeitlichen Bedingungen nicht einhalten, hat das Ergebnis keine Gültigkeit, oder es entsteht das Risiko schwerwiegender Konsequenzen.
3) POSIX Standard 1003.1 definiert ”Echtzeit” für Betriebssysteme als: ”Echtzeit in Betriebssystemen: die Fähigkeit des Betriebssystems innerhalb einer bestimmten, begrenzten Antwortzeit einen bestimmten Dienst zu gewährleisten”

Echtzeit ist ein oft falsch verstandener Ausdruck, und wird von vielen manchmal mit ”schnell” verwechselt. Die Latenz der Antwort auf eine Anfrage ist nicht so sehr das Problem (es können auch gut und gerne Sekunden, Wochen und Tage sein), vielmehr die Garantie des Systems eine bestimmte, vorher bekannte Schranke einhalten zu können.

Oft sind sogar Algorithmen die die Einhaltung einer solchen Schranke garantieren können, ineffizienter als solche, die nicht darauf ausgelegt sind. Alle obigen Definitionen eines Echtzeitsystems beinhalten in der einen oder anderen Form die Begriffe ”rechtzeitig” und ”Garantie”. Diese Arbeit beschäftigt sich mit Echtzeitsystemen die die Einhaltung ”harter” Zeitschranken garantieren müssen, Multi-Threading unterstützen und die Vorteile dynamischer Speicherverwaltung sowie Garbage Collection nützen.

”Hart” bedeutet hier die unbedingte Gewährleistung einer Ausführungszeit unterhalb einer vorgegebenen Schranke. Betrachtet man den Einsatzbereich von Echtzeitsystemen, hat eine Überschreitung im günstigsten Fall ein wertloses Meßergebnis (Messungen bei Experimenten) und im schlimmsten Fall eine Katastrophe zur Folge (Landeanflug). Garbage Collection in einer solchen Umgebung erwies sich als untragbarer Verursacher von Nichtdeterminismus. Bis vor kurzer Zeit fiel daher bei der Frage nach der Implementierungssprache die Wahl eher auf Hochsprachen wie Ada, C, C++ und Assembler für kleinere Systemteile. Auf Garbage Collection und die Vorteile dynamischer Speicherverwaltung musste wegen besagter Unvorhersagbarkeit größtenteils verzichtet werden.

Die von F. Siebert in [SIE02] vorgeschlagenen Modifikationen gängiger Garbage Collection-Techniken und deren Implementierung in einer JavaVM namens ”JamaicaVM” lösen das Problem der Echtzeitfähigkeit von Java. Auf die Besonderheiten dieser ”Proof-of-Concept”-Implementierung wird in einem späteren Kapitel eingegangen werden.

Rainer Kelz

Dezember 6, 2005

II. Problemdefinition

Im folgenden werden die verschiedenen Probleme, auf die man unweigerlich stößt, sofern man Garbage Collection echtzeittauglich machen möchte, etwas näher erläutert.

A. Vorhersagbarkeit

Um über ein Echtzeitsystem und seine zeitlichen Garantien sinnvolle Aussagen machen zu können, muss es vorhersagbar sein. D.h. es muss für alle möglichen Eingaben zu jeder Zeit ein vorhersagbares Laufzeitverhalten aufweisen. Um dieses Ziel zu erreichen, gibt es mehrere Möglichkeiten, manche davon ziemlich radikal.

In [BUW90] wird beispielsweise die Benutzung verschiedener Sprachfeatures wie Rekursion, dynamische Speicherverwaltung oder dynamische Erzeugung von Prozessen schlichtweg untersagt. Das subtrahiert natürlich von allen modernen objektorientierten Sprachen viele ihrer Schlüsselkonzepte, die sie erst attraktiv gemacht haben. Ein Schritt zurück zugunsten der Verlässlichkeit und Vorhersagbarkeit.

Es mag stimmen, das genannte Sprachkonzepte die Bestimmung der Verlässlichkeit erschweren, es ist aber genauso möglich diese Konzepte so zu implementieren, das sie sich verlässlich und vorhersagbar verhalten, wie z.B. eine festgelegte Rekursionstiefe, klares Scheduling der Garbage Collector Aktivitäten und definierte Speicherquantitäten, die vom Garbage Collector einzusammeln sind.

Die wichtigsten Eigenschaften eines zu programmierenden Echtzeitsystems ist die Vorhersagbarkeit des Ressourcenverbrauchs. Als die zwei wichtigsten Ressourcen wären da zum einen die Laufzeit und zum anderen der Speicherverbrauch.

B. Laufzeit

Eine harte Echtzeit - Implementierung einer Programmiersprache ermöglicht die statische Bestimmung der Zeit, die eine primitive Operation in dieser Sprache zur erfolgreichen Ausführung benötigt. Durch diverse statische Analysetools kann vor der Ausführung festgestellt werden, welche zeitlichen Bedingungen für ein bestimmtes Codestück gelten. Dabei muss die Laufzeit unter den denkbar ungünstigsten Bedingungen festgestellt werden. Dies nennt man die *Worst Case Execution Time* (WCET).

Um dynamische Erzeugung von Prozessen zu ermöglichen, muss der Scheduler so beschaffen sein, dass ein höherpriorer Thread eine niederprioren Thread innerhalb einer bestimmten, unbedingt zu gewährleistenden Zeit ablösen kann. Dies nennt man die *Worst Case Preemption Time* (WCPT).

Bei der Implementierung einer echtzeittauglichen Programmiersprache sollte also darauf geachtet werden, dass die für WCET und WCPT anberaumte Zeit recht klein sei.

C. Speicherverbrauch

Der Hauptanteil der weltweit produzierten Mikroprozessoren wird für eingebettete Systeme verwendet (etwa 90%). Viele dieser Systeme müssen echtzeittauglich sein und harte Zeitschranken einhalten (Motorsteuerungen etc.). Solche Systeme sollen zumeist noch vergleichsweise billig herzustellen sein, und sind daher eher mit kleinem Speicher ausgerüstet.

Von einem echtzeittauglichen Programm muss also im vorhin bekannt sein, wieviel Speicher es benötigen wird. Da zur Berechnung der WCET alle möglichen Exekutionspfade bekannt sein müssen, kann somit auch der Speicherverbrauch des Programms ermittelt werden. Das sagt sich leider leichter als es in Wirklichkeit ist. Da der zu allozierende Speicherplatz nicht unbedingt immer gleich groß sein muss, kann interne und externe Fragmentierung auftreten. Mit interner Fragmentierung ist hier der ”Verschnitt” gemeint, der z.B. durch Ausrichtung (”Alignment”) der allokierten Bereiche entsteht. Bei der Verwendung eines Garbage Collectors wird eine Aussage über den zu erwartenden Speicherverbrauch noch zusätzlich verkompliziert, da das Freigeben von Speicher nicht explizit sondern konditional erfolgt.

Von der Möglichkeit der Verwendung virtuellen Speichers in Form von permanentem Speicher (Festplatten o.ä.) kann man getrost absehen, da virtuelle Speicherverwaltung meist inakzeptables und schwer vorhersagbares Laufzeitverhalten hat. Man müsste nämlich die Kosten für einen Seitenfehler zu WCET und WCPT dazurechnen, und erhielte wahrscheinlich in den meisten Fällen völlig unbrauchbare Werte.

D. Scheduler

Ein typisches Echtzeitsystem muss eine Reihe von Aufgaben erfüllen, sei es nun eine immer wiederkehrende Aufgabe, wie das Einschalten eines Förderbands, um ein Werkstück weiterzubewegen, oder das rechtzeitige Reagieren auf eine Eingabe, z.B. Anhalten des Förderbands, sollte erkannt werden, dass ein Notfall eingetreten ist.

Von einem echtzeittauglichen Scheduler wird verlangt, dass er die verschiedenen anstehenden Aufgaben so reiht, das jede Aufgabe innerhalb der gegebenen Schranken abgearbeitet werden kann. Betrachtet man simple, zyklische Scheduler, die eine fixe Zykluszeit zusichern, so sind diese im besten Fall für eine kleine Anzahl von Threads geeignet. Idealerweise handelt es sich bei den Aufgaben dieser Threads um repetitive, kleine Arbeiten. Threads, deren Aufgabe die Reaktion auf ein ausserordentliches, aperiodisches Ereignis ist, müssen ständig nach dem Auftreten dieses Ereignisses fragen und werden im schlimmsten Fall mit der vollen Zykluszeit als Verzögerung aktiv.

Ein eher anwendbares Schema für einen Echtzeit-Scheduler ist Preemptive Priority Scheduling. In seiner Basisform (alle Threads bekommen Prioritäten zugewiesen, der Scheduler wählt immer den höherprioren von den Wartenden) ist diese Art des Steuerung allerdings anfällig für Inversion der Prioritäten, wobei ein niederpriorer Thread einem höherprioren Thread zuvorkommen kann.

Ein Beispiel: Drei Threads A,B und C. A besitzt die höchste Priorität, C die niedrigste. Thread C sperrt eine Ressource R, wird aber von A unterbrochen, welcher sodann versucht, ebenfalls R zu sperren. Das geht aber erst nachdem C fertig ist und R freigibt. A muss also auf C warten, solange C die Ressource hält. Soweit wäre dieses Verhalten noch nicht unvorhersehbar, und es wäre möglich eine statische Analyse durchzuführen um die WCET von A für seine kritische Region und das Erhalten von R festzustellen. Es ist leider aber durchaus möglich, das C von B unterbrochen wird, oder von beliebig vielen Threads, die höhere Priorität haben als C. A müsste dann im schlimmsten Fall auf die Beendigung aller dieser niederprioren Threads warten, obwohl es eigentlich höhere Priorität besitzt.

Ein Konzept namens Prioritätsvererbung schließt eine solche Situation aus (ein unterbrochender Thread erbt die Priorität des Unterbrechers, sobald dieser Zugriff auf eine gerade gehaltene Ressource verlangt). Ein immer noch in Betracht zu ziehender Faktor ist das schwer vorhersagbare Problem eines Deadlocks.

In [RAJ95] wird die sogenannte ”Optimal Priority Inheritance Policy” beschrieben, die garantiert, dass

1) ein höherpriorer Thread maximal einmal von einem niederprioren Thread blockiert werden kann,
2) keine Ketten von Blockaden entstehen können (durch gehaltene Ressourcen),
3) keine Deadlocks auftreten,
4) keine bessere Methode als diese gefunden werden kann.

Eine solche Zeitablaufsteuerung ist also gut geeeignet für Echtzeitsysteme, da sie Aussagen über eine sogenannte Worst Case Preemption Time (WCPT) erlaubt.

E. Anforderungen

Im Hinblick auf oben genannte Schwierigkeiten möchte ich nun überleiten zu einer genaueren Definition der Anforderungen an einen echtzeittauglichen Garbage Collector:

1) Ein echtzeittauglicher Garbage Collector muss für die Schreib- und Lesezugriffe auf Arrays und Objekte eine WCET zusichern.
2) Der Garbage Collector darf den Wechsel von niederpriorem Thread auf höherprioren Thread (”Preemption”) nicht länger als eine festgelegte Zeit lang unterbinden. Er darf also die WCPT des Schedulers nicht verändern.
3) Der Garbage Collector muss das Auffinden und Freigeben von unbenütztem Speicher garantieren.
4) Der Garbage Collector muss exakt sein.
5) Der Garbage Collector muss garantieren, dass Allokationen nicht aufgrund von fragmentiertem Speicher fehlschlagen.
6) Der Garbage Collector muss garantieren, dass er schnell genug die benötigte Menge an Speicher freigibt. (er darf der Applikation nicht ”hinterher hinken”)

In einige Garbage Collector-Algorithmen und Implementationen, die obige Anforderungen erfüllen,

soll ein kurzer Einblick gegeben werden. Auf eine ganz spezielle Implementation in Form der JamaicaVM von Siebert [SIE02], wird auch im Detail eingegangen werden.

Manche Algorithmen und allgemeine Vorgehensweisen die unter dem Begriff ”echtzeittauglich” geführt werden, wie zum Beispiel die ”Tretmühle” von Baker in [BAK92], erfüllen leider einige der im vorigen Kapitel präsentierten Anforderungen nicht (WCET für eine Allokation), oder tun dies nur unter anderweitig unannehmbaren Bedingungen, wie z.B. der Beschränkung auf fixen Objektgrößen. Sie werden daher im folgenden nicht angeführt.

III. Hauptsächlich nicht-kopierende Garbage Collectoren

Der von Bacon et al. in [BAC03] vorgeschlagene Garbage Collector ist ein Hybrid aus nicht kopierendem und kopierendem Modell. Die Betonung liegt allerdings auf nicht-kopierend. Solange noch genügend Speicher vorhanden ist, nutzt der Garbage Collector die Vorteile von nicht-kopierenden Algorithmen. Wird der Speicher knapp, so kommt der kopierende Teil des Algorithmus zum Tragen, und Teile des Speichers werden defragmentiert. Dieser Teil involviert limitiertes Kopieren von Objekten. Es wurde gezeigt, dass diese Kombination zweier verschiedener Techniken imstande ist, geringen Laufzeit- und Speicheroverhead zu garantieren.

Der Algorithmus ist prinzipiell unabhängig vom verwendeten Scheduling-Schema, sei es nun abhängig von der Zeit (kurze Intervalle) oder der Arbeit (kritische Aufgaben zuerst, also Priority Scheduling). Bacon et al. zeigen aber, das man durch Verwendung von zeitbasierten Schedulingverfahren generell bessere Resultate erzielt. Auf die Hauptmerkmale des Algorithmus wird in folgenden Unterkapiteln eingegangen.

A. Isolierte Listen freien Speichers

Der Speicher ist in Seiten fixer Größe unterteilt, welche wiederum in Blöcke einer bestimmten Größe unterteilt sind. Dies wird in Abbildung 1 zu veranschaulichen versucht. Objekte werden in der kleinstmöglichen Blockklasse alloziert.

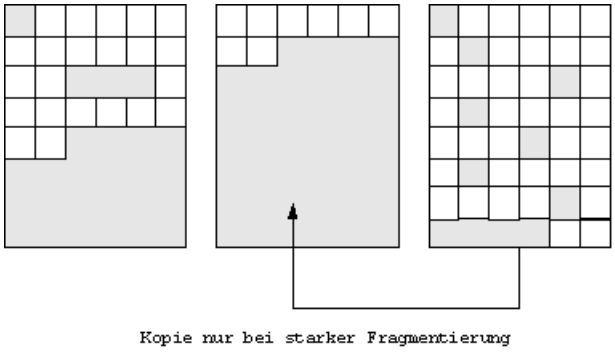

Fig. 1. Isolierte Listen

B. Hybrides Modell

Objekte werden üblicherweise nicht kopiert. Wenn eine Speicherseite fragmentiert wird, (passiert nur durch freigeben eines Objekts in der Seite durch den Garbage Collector), werden die in der Seite enthaltenen Objekte in eine andere, fast volle Seite kopiert.

C. Lesebarriere

Wenn kopiert werden muss, so geschieht dies folgendermaßen: Jedes Objekt hat eine Weiterleitungsreferenz (”Forwarding Pointer”), die normalerweise auf das Objekt selbst zeigt. Wird das Objekt kopiert, wird diese Referenz auf das kopierte Objekt umgebogen. Der Mutator sieht also nur Objekte, die sich im ”to-space” befinden. Anders als bei einem rein kopierenden Garbage Collector sind *to-* und *fromspace* aber nicht strikt disjunkt, sondern mehr oder weniger stark vermischt. Die Benutzung einer argumentierbar relativ laufzeitaufwändigen Lesebarriere (Lesezugriffe sind häufiger als Schreibzugriffe) ist zunächst etwas verwirrend, Bacon et al. zeigen aber, das die Verwendung eines optimierenden Compilers annehmbare Resultate für den Laufzeitoverhead ergeben.

Es gibt zwei fundamental unterschiedliche Varianten einer solchen Lesebarriere. ”Faule” und ”ehrgeizige” Lesebarrieren unterscheiden sich dadurch, das eine ”faule” Barriere das Umbiegen der Weiterleitungsreferenz zum Zeitpunkt der Benutzung (Dereferenzierung) durchführt, wohingegen dies bei einer ”ehrgeizige” Implementierung der Lesebarriere schon beim Laden passiert. Die ehrgeizige Methode bringt also beim Zugriff auf große Arrays (z.B. in Schleifen) einen großen Performancevorteil, weil das Umbiegen nur einmal stattfindet. Unangenehmerweise muss jedoch jedes Register und der Stack nach der ursprünglichen Referenz abgesucht, und diese dann ersetzt werden.

Bacon et al. geben als durchschnittlichen Laufzeitoverhead für ihre (”ehrgeizige”) Implementierung 4% an.

D. Inkrementelles Mark & Sweep

Als eigentlicher Kern des Garbage Collectors wird ein inkrementeller Mark & Sweep Algorithmus verwendet. Die Vorgehensweise ähnelt sehr der von Yuasa in [YUA90] vorgeschlagenen ”snapshot-at-the-beginning” - Strategie.

Bei Yuasa werden alle Objekte, die zu Beginn eines Garbage Collector - Zyklus erreichbar sind, bis zum nächsten Zyklus konserviert (das besagte Abbild (”Snapshot”)). Ausserdem wird gewährleistet, das der Speichergraph in keiner gefährlichen Weise vom Mutator modifiziert wird. Eine Schreibbarriere wird benutzt um diese Bedingung zu erfüllen. Soll eine Referenz zu einem noch nicht als lebend identifizierten Objekt (weiß markierte Objekte) überschrieben werden, so wird die Referenz auf einem speziellen Garbage Collector-Stack gesichert. Dieser Stack wird dazu benutzt, Referenzen auf Objekte zu halten, die zwar leben, aber noch nicht besucht wurden (grau markierte Objekte). Alle erreichbaren Objekte werden so entweder vom Collector selbst oder von der Schreibbarriere als lebend identifiziert (schwarz markiert).

Der Unterschied zum originalen Vorbild liegt in einer kleinen Modifikation der Markierungsphase. Jede Referenz die ein Objekt im from-space referenziert, das aber schon kopiert wurde, wird auf das entsprechende Objekt im to-space umgebogen. Somit können nach Abschluß einer Markierungsphase die schon kopierten Objekte im Abbild aus dem letzen Garbage Collector-Zyklus freigegeben werden.

Die WCET für Operationen auf Referenzen ist sehr klein. Die meiste Zeit ist keine Lesebarriere vonnöten. Die WCET der Schreibbarriere ergibt sich aus den Kosten für das Ablegen des alten Referenzwerts auf dem Stack und den Kosten für das Markieren des Objekts, das vom alten Wert referenziert wurde. Lese- und Schreibzugriffe auf Referenzen stellen also keine große Bedrohung für die Einhaltung von WCET-Schranken dar.

Größere Allokationen oder viele kleine Allokationen mit hoher Frequenz, wenn sie von einem hochprioren Thread durchgeführt werden, sind allerdings unter Umständen doch ein Problem. Dieser Unannehmlichkeit kann man jedoch z.B. durch eine geeignete Schedulingstrategie entgegenwirken. Anstatt ein bestimmtes Quantum an Garbage Collector-Arbeit pro Allokation durchzuführen, wird nach Ablauf einer fixen Zeit (oder nach X Instruktionen) ein bestimmtes Quantum an Garbage Collector-Arbeit verrichtet.

E. Arraylets

Um die WCET beim Kopieren sehr großer Objekte, wie Arrays, gering halten zu können, werden Arrays in kleinere Blöcke fixer Größe aufgeteilt. Diese sogenannten ”Arraylets” helfen dabei, die für das Lesen und Kopieren von Arrays benötigte Arbeit einschränken zu können, da inkrementell kopiert werden kann, und helfen ausserdem, externe Fragmentierung zu vermeiden.

F. Verquickung

Die Garbage Collector-Aktivität ist eng mit dem Mutator verbunden. Es gibt keinen eigenen Thread für den Garbage Collector. Mit einem Zyklus ist ein kompletter Durchlauf der drei Phasen Mark-Sweep-Defragment gemeint.

G. Zusammenfassung

Bacon et al. geben an, das durch die Verwendung ihres Hybriden nicht mehr als 4% der Daten kopiert werden müssen (Daten aus Messungen). Weil der Grad der Fragmentierung begrenzt ist, gibt es also eine feststellbare obere Grenze für den Speicherverbrauch. Der Collector hat zusätzlich einen kleineren Speicheroverhead als ein kopierender, da er primär als nicht-kopierender, inkrementeller Garbage Collector fungiert.

Es wird ausserdem gezeigt, dass bei der Verwendung eines optimierenden Java-Compilers eine effiziente Implementierung einer Lesebarriere in Software möglich ist. Als Optimierungen wurden unter anderem ”Common Subexpression Removal” und ”Barrier-Sinking” genannt. Die Implementierung und Evaluierung des Kollektors hat gezeigt, das eine gut vorhersagbare Benutzungsrate des Mutatoren möglich ist. Bacon et al. geben eine Benutzungsrate von 45% für den Mutator an, während der Garbage Collector aktiv ist und weiters einen Speicheroverhead, der das 1.6 bis 2.5 -fache des von der Applikation benötigten Speichers ausmacht.

IV. Henriksson's Scheduling Strategie

Henriksson identifiziert in [HEN98] drei Operationen, die Garbage Collector - Aktivität auslösen können. Lese- und Schreibzugriffe auf Zeiger, sowie Speicherallokation. Sein Ansatzpunkt, um WCET und WCPT zu minimieren, ist im wesentlichen die günstige zeitliche Verteilung von Garbage Collector - Aktivität. Er vertritt die Auffassung, das hochpriore Threads gar nicht von Garbage Collector-Aktivität unterbrochen werden dürfen. Während der Ausführung dieser Threads anfallende Arbeit muss in den Pausen zwischen der Aktivierung hochpriorer Threads, also während der Ausführung niederpriorer Threads abgearbeitet werden.

A. Prioritätsverteilung

Die Verteilung der Prioritäten sieht folgendermaßen aus:

- Klasse der hochprioren Threads
- Klasse Garbage Collector-Aktivität hochpriorer Threads
- Klasse der niederprioren Threads, Garbage Collector-Aktivität niederpriorer Threads

Eine genauere Betrachtung dieser Verteilung schließt die Möglichkeit des "Verhungerns" ("Thread-Starvation") eines niederprioren Threads mit ein. Um dem entgegenzuwirken, ist hochpriore Garbage Collector-Aktivität insofern beschränkt, als nur solange Speicher freigegeben wird, bis garantiert werden kann, das hochpriore Threads immer genügend freien Speicher vorfinden. Eine graphische Darstellung der Prioritätsvertielung findet sich in Abbildung 2.

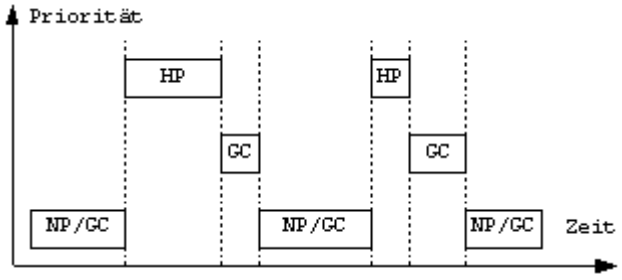

Fig. 2. Prioritätsverteilung

Ein Weiterdenken dieses Ansatzes wäre die Benutzung von Idle-Phasen des Prozessors für Garbage Collector-Aktivität (hätte automatisch niedrigste Priorität). Das würde jedoch nur die durchschnittliche Performanz der Applikation erhöhen, und hätte wenig Einfluss auf die schlechtestmögliche Performanz, die in Echtzeitsystemen eine so wichtige Größe darstellt.

Durch diese Einteilung in Prioritätsklassen, wie sie in ähnlicher Form bei Betriebssystemen zu finden sind, wird hochprioren Threads eine idealisierte Sicht auf den Speicher gegeben. Keine Unterbrechung durch Garbage Collector-Aktivität, kein zusätzlicher Overhead bei Allokationen oder Zeigeroperationen. Für einen niederprioren Thread sieht die Sache so aus, dass er hin und wieder für kurze, beschränkte Zeiten von einem inkrementellen Garbage Collector unterbrochen wird. Henriksson nennt diese Strategie *semi concurrent garbage collection*, da Garbage Collector-Aktivität nur mit den hochprioren Threads verschränkt passiert, mit niederprioren Threads jedoch sequentiell.

B. Aufteilung des Speichers

Der Speicher ist in einen vom Garbage Collector betreuten Bereich und einen normalen C-Style Heap geteilt.

C. Wurzelzeigererkennung

In dem von Henriksson entwickelten Prototypen sind für Wurzelzeiger sowohl auf dem Stack, als auch auf dem Heap spezielle Datenstrukturen, Garbage Collector-Stacks genannt, vorhanden. Diese beeinhalten detaillierte Informationen über den Ort von Wurzelzeigern. Im Prototypen werden diese Informationen über eigens zu verwendende Methoden für Allokation und Deallokation aktualisiert. Jedes Objekt muss einen genau definierten Header haben und auf eine spezielle Datenstruktur verweisen, die das Layout des Objekts hinsichtlich der Position der Zeiger im Objekt definiert. Im Prototypen müssen diese Informationen vom Programmierer sozusagen "händisch" zur Verfügung gestellt werden, da von einem nichtkooperierenden Compiler ausgegangen wird.

D. Speicherinitialisierung

Henriksson schlägt vor, einen ausreichenden Teil an Speicher im *tospace* vorzuinitialisieren. Dies geschieht Aufgrund der Tatsache, das Zeigerfelder

in neuen Objekten einen wohldefinierten Initialwert haben müssen, um nicht fälschlicherweise als gültiger Zeiger (non-null) weiterverfolgt zu werden. Der für neue Allokationen von hochprioren Threads zu benutzende Speicherbereich wird also mit 0 vorinitialisiert. In Abbildung 3 wird dieser Sachverhalt genauer illustriert. Niederpriore Threads müssen den Speicherbereich selber initialisieren. Argumentiert wird damit, dass die Initialisierung die meiste Zeit einer gesamten Allokationsoperation in Anspruch nimmt. Die Größe des vorzuinitialisierenden Bereichs muss dem Garbage Collector bei seiner Initialisierung mitgeteilt werden. Diese Größe stellt daher die maximale Größe des von einem hochprioren Thread allokierbaren Speicherbereichs dar.

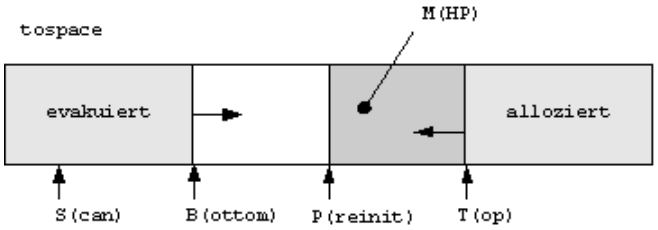

Fig. 3. Preinitialisierter Bereich

E. "Faule" Evakuierung

Die Idee bei der faulen Evakuierung ist im Wesentlichen, dass die eigentliche Evakuierung eines Objekts vom *fromspace* in den *tospace* während der Ausführung eines hochprioren Threads solange aufgeschoben wird, bis ein niederpriorer Thread an die Reihe kommt.

Angenommen, es gibt drei Objekte A,B und C. A und B sind bereits in den *tospace* evakuiert worden, C befindet sich noch immer im *fromspace*. In A gibt es einen Zeiger x, der auf das Objekt C verweist. Tritt in einem hochprioren Thread nun eine Zuweisung der Form B.y = A.x; auf, so passiert folgendes: Die Schreibbarriere erkennt, dass ein Objekt referenziert wird, dass sich noch im *fromspace* befindet, und ausserdem noch nicht evakuiert wurde. Die Barriere stellt nun anhand eines dafür vorgesehenen Zeigers fest, ob schon Speicher für eine solche Evakuierung im *tospace* reserviert wurde. Wenn nicht, wird ein solcher Bereich reserviert. In diesem Bereich wird ein Zeiger angelegt, der auf das Original im *fromspace* zeigt. Dieser Zeiger stellt auch den eigentlichen Unterschied zu Standard Copying Collectoren dar. B.y zeigt nun auf den reservierten Speicherbereich, und arbeitet bei jedem Zugriff auf das Original im *fromspace*.

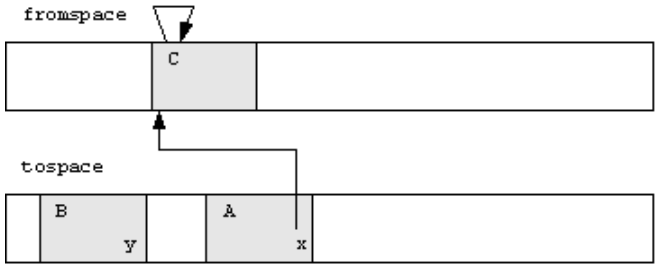

Fig. 4. Ausgangssituation

In Abbildung 4 ist die Ausgangssituation dargestellt. A.x hält eine Referenz auf ein Objekt im *fromspace*. Das Programm stößt nun auf die Instruktion

```
B.y = A.x;
```

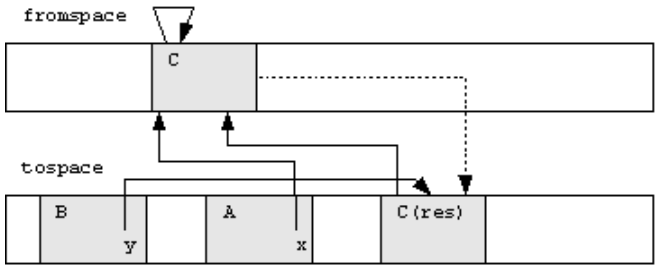

Fig. 5. Reservierung des Speichers

Abbildung 5 zeigt, was nach einem Abfangen des Statements

```
B.y = A.x;
```

passiert. Der gestrichelte Pfeil ist ein temporärer Zeiger in C, der auf den reservierten Speicherbereich C(res) zeigt. Der solide Pfeil von C(res) auf C ist der "forwarding pointer" auf C im *fromspace*. Alle Zugriffe auf C von B aus laufen nun über diesen Zeiger.

In Abbildung 6 ist die Situation nach dem Pausieren des hochprioren Threads gezeigt. Der Garbage Collector hat das Objekt C in den reservierten Bereich kopiert, und den "forwarding pointer" von C(old) auf die *tospace*-Kopie umgebogen. Der Garbage Collector hat bis jetzt nur

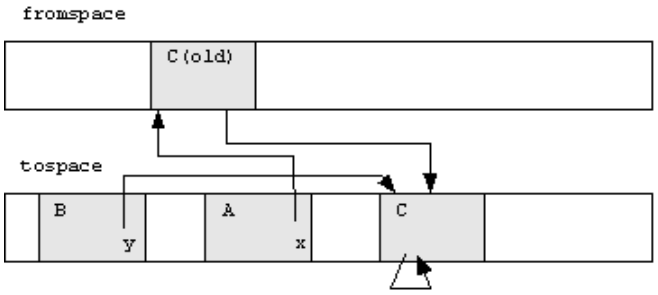

Fig. 6. Nach der Pausierung des hochprioren Threads

die Arbeit getan, die während der Ausführung des hochprioren Threads angefallen ist. Die Referenz von A.x auf C(old) wird später umgebogen werden, wenn A vom Garbage Collector gescannt wird. Der beschriebene Fall kann natürlich nur dann auftreten, wenn A grau ist, also noch nicht gescannt wurde. Wäre es schon gescannt, gäbe es keine Referenzen mehr in den *fromspace*.

Um genügend Platz für Objekte zu haben, die für die Evakuierung vorgemerkt wurden, gibt es den Zeiger B(unevakuiert), wie in Abbildung 7 zu sehen. Die Schreibbarriere ist dafür zuständig, den Zeiger B um die Größe des Objekts zu versetzen.

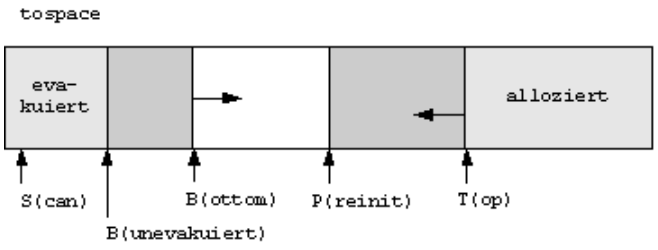

Fig. 7. Platz für zur Evakuierung vorgemerkte Objekte

1) Der hochpriore Garbage Collector: Wann immer ein hochpriorer Thread pausiert, und kein anderer hochpriorer Thread laufen möchte, wird der hochpriore Garbage Collector angestoßen. Er ist dafür zuständig, sämtliche angefallene Arbeit zu erledigen, und ausserdem genügend Speicher für hochpriore Threads vorzuinitialisieren. Dieser Prozess hat eine Priorität die kleiner ist, als die kleinste der hochprioren und größer als die größte der niederprioren Prozesse.

2) Arbeitsquantum des Garbage Collectors: Um das Ausmaß der vom Garbage Collector durchzuführenden Arbeit feststellen zu können, also ob der Collector laufen soll oder nicht, muss man eine minimale Garbage Collection - Rate feststellen. Ist die durchgeführte Speicherbereiniguns-Arbeit größer als diese Rate, so ist garantiert, dass alle Objekte aus dem *fromspace* komplett evakuiert wurden, bevor der *tospace* voll ist. detaillierte Angaben zur Bestimmung dieser Rate finden sich in [HEN98].

V. Hard Realtime Garbage Collection

Von Siebert wurde in [SIE02] ein Ansatz zur Garbage Collection vorgeschlagen, der eine hartechtzeittaugliche Implementierung einer JavaVM ermöglicht. Dieser Ansatz wurde in einer ”Proof-of-Concept” Implementierung, der JamaicaVM realisiert.

Der vorgeschlagene Garbage Collector unterliegt allerdings einige Einschränkungen. Er ist strikt monoprozessortauglich. Eine parallele Implementation unterläge der (signifikanten) Einschränkung, dass mehrere Threads (Mutatoren) die denselben Heap modifizieren, nicht parallel ausgeführt werden können. Es muss ausserdem eine bekannte Schranke für das Maximum an erreichbarem Speicher geben.

A. Arbeitsweise

Der in der JamaicaVM implementierte Garbage Collector basiert auf dem 1976 vorgeschlagenen Mark & Sweep Algorithmus ([DIJ76]). Es werden drei Farben zur Markierung von Objekten verwendet, und zwar wie folgt :

1) Weiß - identifiziert noch nicht besuchte Objekte, die vielleicht auch gar nicht erreichbar sind
2) Grau - identifiziert noch zu besuchende Objekte, die unter Umständen auch weiße Objekte referenzieren.
3) Schwarz - identifiziert bereits besuchte Objekte, die graue und/oder schwarze Objekte referenzieren.

Die Invariante ”kein schwarzes Objekt darf ein weißes Objekt direkt referenzieren” muss zu jeder Zeit vom Mutator und natürlich vom Garbage Collector respektiert werden. Wir werden allerdings bei der Diskussion der Snychronisation Points sehen, dass der Ausdruck ”zu jeder Zeit” in ”an jedem Synchronisation Point” umgewandelt werden kann, was einige Erleichterung bringen wird.

Die Mark-Phase beginnt mit dem Markieren der Wurzelreferenzen als grau. Die Phase setzt sodann fort, indem alle grauen Objekte besucht, und von ihnen referenzierte Objekte grau markiert werden, sofern sie nicht schon schwarz sind. Jetzt wird das geradeeben gescannte Objekt schwarz markiert. Dies setzt sich solange fort, bis es keine grauen Objekte mehr gibt. Von den Wurzelreferenzen aus nicht erreichbarer Speicher ist somit immer noch weiß markiert. Während der Sweep-Phase wird der weiße Speicher an die free-Liste angefügt.

Da der Garbage Collector inkrementell arbeitet, läuft währenddessen der Mutator ebenfalls, und modifiert Referenzen. Es muss durch Schreibbarrieren gewährleistet sein, dass die oben genannte Invariante durch den Mutator nicht verletzt wird. Jedes weiße Objekt, das referenziert wird, muss grau markiert werden.

Um diesen Algorithmus echtzeitauglich zu machen, mussten natürlich etliche Modifikationen vorgenommen werden. Es stellte sich unter anderem die Frage, wann Garbage Collector-Aktivität zugelassen werden soll, und unter welchen Bedingungen der Garbage Collector gewährleisten kann, genügend Speicher zu bereinigen. Durch die Einführung von sogenannten ”Synchronization Points” wurden diese Fragen ausreichend gut beantwortet. In der Tat sind diese Synchronisationspunkte die einzigen Zeitpunkte im Programm, an denen Garbage Collection - Aktivitäten stattfinden dürfen. Auch sind es die einzigen Punkte an denen ein Threadwechsel passieren kann. Weitere Probleme wie die hinreichende Optimierung der Root-Scanning Phase und das Problem der Speicherfragmentierung mussten ebenso überwunden werden.

B. Garbage Collector Aktivität

Der ursprüngliche Algorithmus von Dijkstra et al. baut auf die Zusammenarbeit zwischen Collector und Mutator. Diese Zusammenarbeit wurde in der implementierten VM erleichtert und vertieft. Die oben erwähnten ”Synchronization Points” sind im wesentlichen zusätzliche Codestücke, die zur Compile-Zeit eingefügt werden. Nur zu diesen Zeitpunkten darf der Garbage Collector aktiv sein, und nur zu diesen Zeitpunkten dürfen Threadwechsel stattfinden.

Graphisch dargestellt ist dieser Sachverhalt in Abbildung 8. Die grauen Flächen entlang den Linien stellen Thread-Aktivität dar. Dunklere Flächen

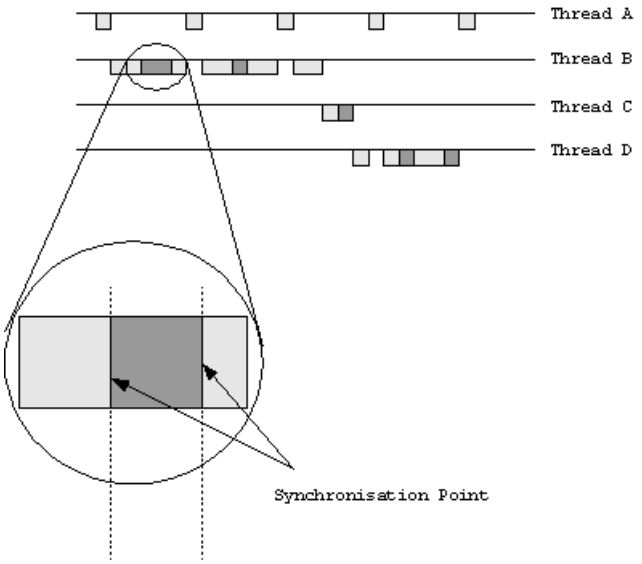

Fig. 8. Garbage Collector Aktivität

bedeuten Garbage Collector Aktivität. Einfach ausgedrückt bedeutet das, dass der Mutator - Thread dafür herangezogen wird, Zeit (CPU Zyklen) zur Verfügung zu stellen, um den Speicher zu bereinigen. Sinvollerweise z.B. bei jeder Allokation. Das freizugebende Speicherquantum wird als Funktion des noch freien Speichers angesehen. Es wurde gezeigt, dass durch diese Bedingungen gewährleistet ist, dass für jedes neue Objekt am Heap der entsprechende Platz freigemacht wird.

Ein weiteres zeitliches Problem stellte die ”Root Scanning” Phase des Garbage Collector dar. Betrachtet man Objekte auf dem Heap als Knoten und Referenzen als Kanten erhält man im Endeffekt einen gerichteten Graphen. Dieser Speichergraph ist zunächst nur durch Referenzen auf dem Stack, in den Registern und in globalen Variablen zugänglich. Diese Referenzen werden auch Wurzelreferenzen genannt.

Um im späteren Verlauf des Garbage Collector - Zyklus feststellen zu können, ob unerreichbarer Speicher freigegeben werden kann, muss der Garbage Collector also zunächst alle Wurzelreferenzen identifizieren. Es gibt mehrere Ansätze, diese Aufgabe zu bewältigen.

C. Wurzelidentifikation

1) Konservativ: Ein Ansatz ist die konservative Referenzidentifikation. Hier weiß der Garbage Collector nur, das in einer bestimmten Region im

Speicher wahrscheinlich Referenzen zu finden sein werden, und kann solche nur aufgrund von Bitmustervergleichen identifizieren. Ein unangenehmer Nebeneffekt dieser Vorgehensweise könnte aber sein, dass so auch eine ”unglücklich” aussehende Variable anderen Typs fälschlicherweise als Referenz identifiziert wird, und ein völlig willkürlich gewählter Speicherbereich vom Garbage Collector verschont bleibt. Ein weiterer Nebeneffekt ist der hohe Zeitaufwand. Ein solcher Garbage Collector ist also nicht exakt und liefert daher auch keinerlei Zusicherungen über freigegebenen Speicher oder etwa eine WCET. Diese Unzulänglichkeiten machen konservative Wurzelidentifikation unbrauchbar für einen echtzeitauglichen Garbage Collector.

2) Exakt: Im Kontrast dazu stehen die exakten Garbage Collectoren. Durch spezielle Kennzeichnung von Referenzen (z.B. durch entsprechende Hardwareunterstützung) können Referenzen exakt identifiziert, und somit garantiert werden, dass vom Mutator unerreichbarer Speicher freigegeben wird. Eine WCET lässt sich leider immer noch nicht genau angeben.

3) Zusätzliche Informationen: Der nächste Schritt in Richtung Echtzeit geschieht durch Verfügbarmachung genauerer Information über die Wurzelreferenzen. Im Falle der JamaicaVM passiert dies durch Kopie der ”lebenden” Referenzen (Referenzen die einen Synchronization Point überdauern) in Arrays auf dem Heap, die alle durch eine fixe Wurzelreferenz erreichbar sind. Abbildung 9 soll dies verdeutlichen.

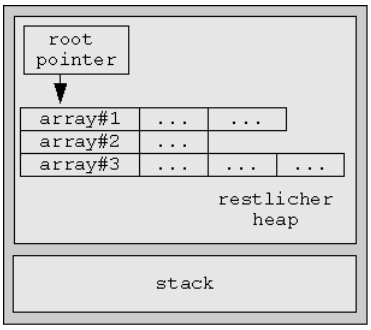

Fig. 9. Root Pointers auf dem Heap

Die Phase der Wurzelidentifikation ist somit auf die Markierung dieser speziellen Wurzelreferenz reduziert worden. Die Garbage Collector Inkremente können nun Wurzelreferenzen wie jedes andere Objekt auf dem Heap behandeln. Es ist weiters garantiert, dass alle noch gebrauchten Objekte früher oder später vom Garbage Collector erreicht und als ”lebend” eingestuft werden, und es lässt sich eine (sehr kleine) WCET für diese Phase des Garbage Collector Zyklus angeben.

Diese Technik beseitigt auch eine der ursprünglichen Schwierigkeiten mit dem Algorithmus von Dijkstra. Während der Markierungsphase veränderte Wurzelreferenzen könnten zur Folge haben, dass referenzierte Objekte vom Markierer nicht gefunden würden. So werden jedoch vor jedem Garbage Collector-Inkrement die Wurzelreferenzen auf den Heap kopiert, und es ist durch die eine globale Wurzelreferenz sichergestellt, dass die Markierungsphase sie findet.

D. Fragmentierung

Das Problem der Speicherfragmentierung in Bezug auf konservative Garbage Collectoren wird anhand von vier gut bekannten und oft angewandten Algorithmen in [ZOR92] ausführlich behandelt. Jeder Garbage Collector - Algorithmus muss Fragmentierung auf die eine oder andere Weise in den Griff bekommen. Echtzeittauglichkeit im Sinne einer Garantie für eine (kleine) WCET ist durch gängige Speicherdefragmentierung (Mark & Compact) aber nicht gegeben.

Die JamaicaVM geht auch hier einen anderen Weg. In dieser Implementierung ist der Speicher in Blöcke von fixer Größe unterteilt. Objekte werden als verkettete Listen solcher Blöcke im Speicher abgelegt. Somit tritt das Problem der Fragmentation erst gar nicht auf (siehe Abbildung 10).

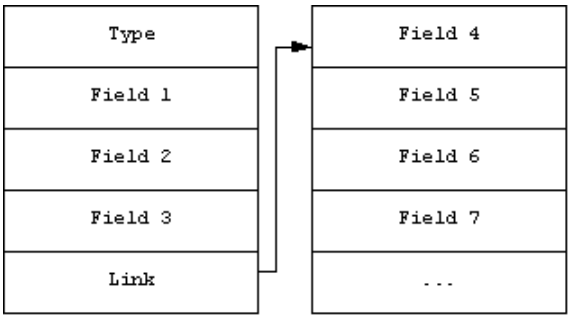

Fig. 10. Zusammensetzung von Objekten aus Speicherblöcken fixer Größe

Da Java-Objekte relativ klein sind (12-23 Bytes im Durchschnitt), kann auch die Blockgröße relativ klein gewählt werden. Der tatsächliche Wert kann

beim Feintuning einer Applikation dem Compiler mitgeteilt werden. Die Zeitkomplexität eines Zugriffs auf ein Feld in einem Objekt ändert sich durch die Verwendung dieses Speichermodells von O(1) auf O(p). Dadurch, dass die Größe des Objekts zur Compilezeit bekannt ist, lässt sich aber durchaus eine WCET für den Feldzugriff angeben.

Immer noch ein Problem stellen sehr große Objekte, oder Objekte wie Arrays dar, die beliebige Größe aufweisen können. Es wird eine Baumstruktur analog Abbildung 11 vorgeschlagen.

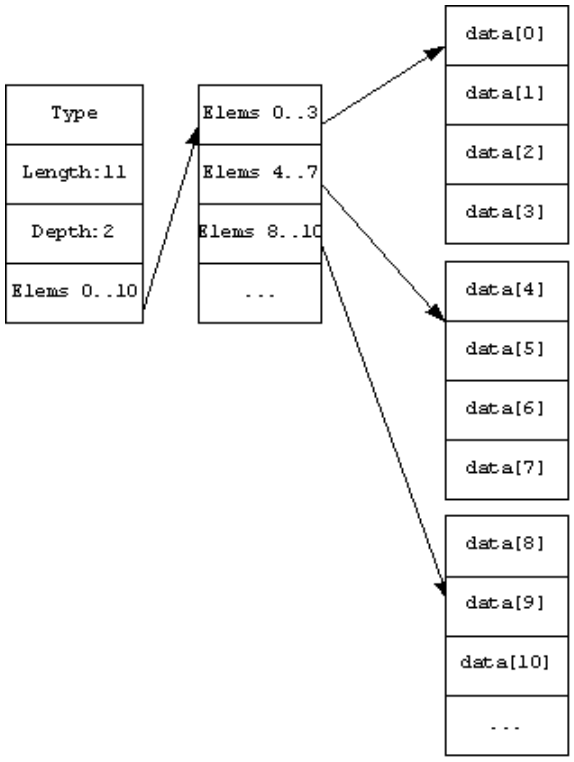

Fig. 11. Baumstruktur für große Objekte wie Arrays

Die Zeitkomplexität für einen Zugriff sinkt somit auf O(ln(size)). Java-Arrays brauchen Informationen über ihren Typ und ihre Länge. Diese Werte finden sich im Arrayheader, zusammen mit Informationen über die Tiefe des Baums. Siebert selbst bezeichnet O(ln(size)) als schockierend langsam, im Vergleich mit O(1), relativiert dann aber, und nennt ein Beispielsystem mit einem 16MB großen Heap, der in fixe Blöcke der Größe 32Byte unterteilt ist. In einem solchen System kann die maximale Baumtiefe den Wert 7 nicht überschreiten. Die WCET für einen Zugriff auf einem solchen System ist somit beschränkt.

Durch das sehr häufige Auftreten von Arrays in Java, wird jedoch der Array-Zugriffscode zu einem potentiellen Flaschenhals. Es wäre schön, auf flache Arrayrepräsentationen zurückgreifen zu können, sollten genügend Speicherblöcke hintereinander vorhanden sind. Das geschieht in der JamaicaVM einfach durch setzen des Tiefen-Feldes im Arrayheader. Der Zugriffscode auf das Array ändert sich dadurch nicht, es brauchen keine Ausnahmeregelungen getroffen werden, und die Baumstruktur kann quasi als Sicherheitsnetz verwendet werden, wenn der Speicher zu fragmentiert ist.

E. Synchronisation Points

Ein Synchronisation Point ist im Grunde ein kurzes Codestück, das zur Compilezeit eingefügt wird. Das Pseudocodestück in Abbildung 12 zeigt die Struktur eines solchen Punktes.

```
if(threadswitch_required == true) {
 ...thread-informationen speichern...
 ...threadswitch erlauben...
}
```

Fig. 12. Synchronisation Point

Die Ablöse eines Threads durch einen anderen wird vom Scheduler durch das Setzen des globalen Flags *threadswitch_required* veranlasst. Synchronisation Points sind somit die einzigen Punkte im Code, an denen ein Threadwechsel vollzogen wird. Es sind dies auch die einzigen Punkte, an denen Garbage Collector-Aktivität stattfindet. Das hat mehrere Auswirkungen, auf die im folgenden näher eingegangen wird.

F. Monoprozessortauglichkeit

Durch die Beschränkung von Threadwechsel auf Synchronisation Points ist gewährleistet, dass immer nur ein Thread, der Garbage Collection benötigt, aktiv ist. Das ganze System baut darauf, das der heap immer nur von genau einem Mutator verändert wird. Das schließt die Benutzung von Parallelprozessorsystemen weitestgehend aus, da mehrere Threads auf parallelen Prozessoren denselben Heap nicht verändern dürften.

G. Platzierung von Synchronisation Points

Synchronisation Points müssen häufig genug eingefügt werden, um eine schnelle Threadablöse

zu gewährleisten. Stellen, an denen Synchronisation Points vorhanden sein müssen, sind lange, lineare Codesequenzen, im Inneren von Schleifen und vor potentiell rekursiven Aufrufen. Im Garbage Collector Code selbst müssen sich ebenfalls solche Punkte befinden. Nach jedem Markieren oder Freigeben eines Blocks, nach jedem Garbage Collector-Inkrement also, muss ebenfalls ein Synchronisation Point eingefügt werden.

Nach erfolgter statischer Analyse ist es also für eine gegebene Plattform möglich, eine obere Grenze für das Zeitintervall zwischen zwei Synchronisation Points anzugeben. Um einen Synchronisation Points in Software realisieren zu können, wird eine globale Semaphore benötigt. Die Threadablöse wird durch das Freigeben und den sofortigen Versuch, die Sempahore wiederzuerlangen bewerkstelligt.

1) Invarianten: Das hat zur Folge, das einige Restriktionen, die für einen Mutator-Thread gelten, zwischen zwei Synchronisation Points nicht, oder nur beschränkt gelten müssen. Wenn zum Beispiel ein Mutator eine Referenz von einem schwarz markierten Objekt auf ein weiß markiertes Objekt setzen möchte, so ist das durchaus möglich. Dadurch werden auch aggressive Compileroptimierungen des Codes zwischen zwei Synchronisation Points möglich. Code für eine Schreibbarriere kann z.B. für Instruktion Scheduling herangezogen werden. Die oben genannte, für ein Drei-Farben Markierungsschema typische Invariante kann also temporär durch Optimierungen verletzt werden, solange sie vor dem nächsten Synchronisation Point wiederhergestellt wird.

2) Locking: Es sind weiters keine Locking-Mechanismen auf globale Datenstrukturen notwendig. Das heißt, das sogar Schreibbarrieren und Allokationen von Objekten keine Locks benötigen, weil ja nur immer ein Thread aktiv ist. Der Code zwischen zwei Synchronisation Points ist automatisch atomar.

3) (Un-)Exakte Referenzinformationen: Es wird keine exakte Information über Referenzen zwischen zwei Synchronisation Points benötigt, da hier auch kein Garbage Collection stattfinden kann. Jede lokale Referenz also, deren Lebensspanne keinen Synchronisation Point beeinhaltet, muss nicht in das Root-Array kopiert werden.

H. Speichern von Wurzelreferenzen

An jedem Synchronisation Point müssen die gerade lebenden Referenzen in eine spezielle Datenstruktur auf dem Heap kopiert werden, da es ja an jedem solchen Punkt möglich ist, dass der Garbage Collector zu laufen beginnt. Die Dauer dieser Operation ist kritisch für die Performanz der Applikation. Es sollte also danach getracht werden, so wenig Referenzen wie möglich auf den Heap zu kopieren. Der Zeitpunkt an dem Referenzen kopiert werden, ist sehr wichtig. Sollte also die Lebensspanne einer Referenz einen oder mehrere Synchronisation Points überdauern, so bieten sich im Grunde zwei Möglichkeiten an, diesen Zeitpunkt zu wählen - entweder direkt vor einem Synchronisation Point (”late saving”) oder direkt bei Definition der Referenz (”early saving”). Eine Lebensspanne kann nun aber mehrere (Re)Definitionen beeinhalten, als auch mehrere Enden haben. Abbildung 13 soll illustrieren, wie die Lebensdauer einer Referenz aussehen kann.

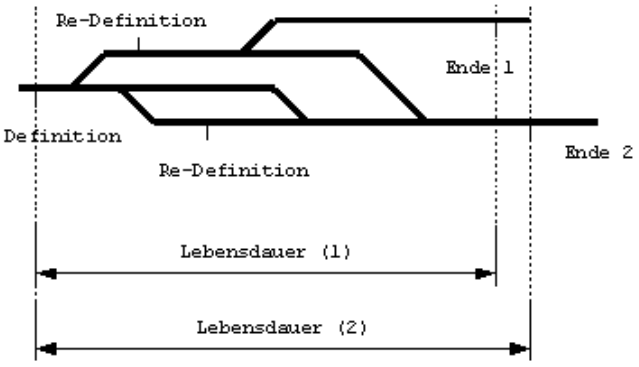

Fig. 13. Die Lebensdauer einer Referenz

Beim frühen Sichern muss also zu jeder (Re)Definition und zu jedem Ende der entsprechende Code eingefügt werden, der die Referenz in das Root-Array schreibt, auf dem neuesten Stand hält, und wieder löscht.

Frühes Sichern hat unter Umständen den Nachteil, das es durch Synchronisation Points, die auf Grund von Verzweigungen im Code gar nicht erreicht werden, ausgelöst wird. Spätes Sichern hat den Nachteil, das unter Umständen viele Referenzen oft unnötig kopiert und wieder gelöscht werden.

Wie meistens der Fall, liegt die optimale Strategie sozusagen ”in der Mitte”, und wird von Siebert ”mixed” genannt. Für Referenzen deren Lebenspan-

nen einen Call Point enthalten, also einen Methodenaufruf, wird die ”early saving” Strategie gewählt, da es sehr wahrscheinlich ist, das der Call Point auch ausgeführt wird, und eine Sicherung der Referenz vonnöten ist. Für Lebensspannen, die keinen Methodenaufruf beeinhalten, kann die ”late saving” - Strategie gewählt werden. Der Code zur Sicherung der Referenzen erfolgt konditional, wie in Abbildung 14 zu sehen.

```
ref<1> = ...;
...
ref<n> = ...;
if(threadswitch_required == true)
{
 save_to_root_array(ref<1>);
 ...
 save_to_root_array(ref<n>);
 //...threadswitch erlauben...
 V(global_semaphore);
 P(global_semaphore)
 clear_from_root_array(ref<1>);
 ...
 clear_from_root_array(ref<n>);
}
...
```

Fig. 14. Konditionales (spätes) Sichern

I. Schreibbarrieren

Die Aufgabe einer Schreibbarriere ist die Aufrechterhaltung der Garbage Collector Invarianten, wie z.B. die Bedingung, dass kein schwarz markiertes Objekt ein weiß markiertes Objekt direkt referenzieren darf, während der Garbage Collector seine Arbeit verrichtet. Um diese Invariante gewährleisten zu können, muss die Schreibbarriere ein Objekt grau markieren, falls eine Referenz aus diesem Objekt auf ein noch weißes Objekt zeigt.

```
...
if{ref != null) {
    Object **colour = adr_of_colour(ref);
    if((*colour) == white)
    {
        (*colour) == greyList;
        greyList = ref;
    }
} obj->f = ref;
...
```

Fig. 15. Schreibbarriere in der JamaicaVM

Der Vorgang des grau-markierens wird in der JamaicaVM analog dem Codefragment in Abbildung 15 durchgeführt. Graue Objekte werden in einer verketteten Liste dargestellt, deren Ende durch eine spezielle nicht-valide Referenz, *last_grey* gekennzeichnet ist. Die Farben Weiß und Schwarz werden ebenfalls durch nicht-valide Referenzen codiert.

J. Heapeinteilung

Der Heap ist, wie schon erwähnt, in Blöcke fixer Größe eingeteilt. Um die Farbkodierung der Blöcke zu realisieren wird zusätzlicher Speicher benötigt. der Garbage Collector benötigt auch noch zusätzliche Informationen, welches Feld in jedem Block eine Referenz darstellt.

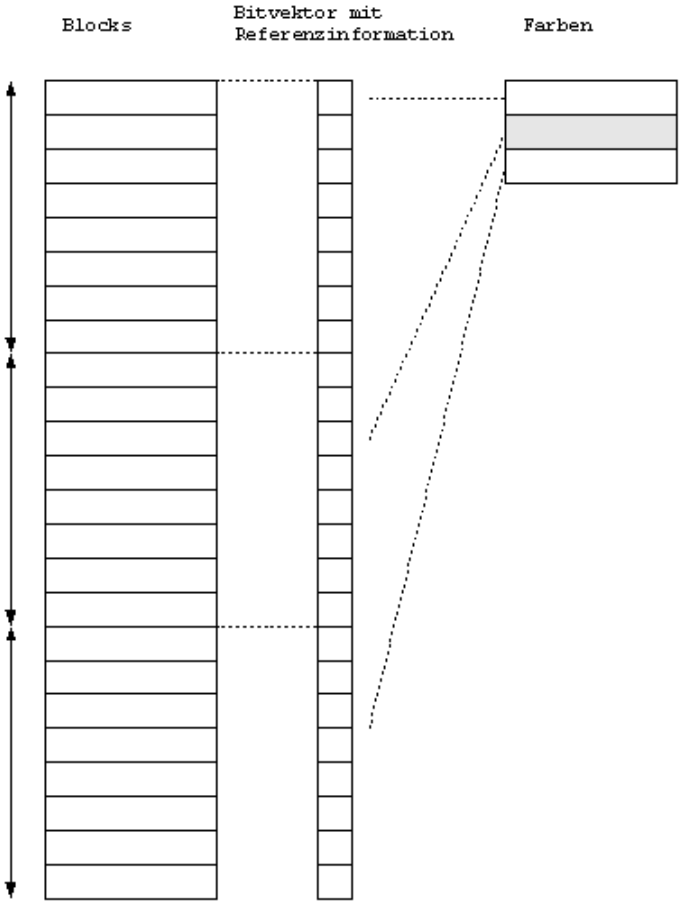

Fig. 16. Die Einteilung des Heap

Um diese Informationen abzubilden, existieren parallel zu den Blöcken zwei zusätzliche Arrays. Das eine ist ein Bitvektor, der für jedes Wort in einem Block (Wort == 4 Byte; die Größe einer Referenz) speichert, ob es eine Referenz ist. Das

zweite Array hält für jeden Block ein Wort, in dem die Farbe des Blocks codiert ist.

K. Markierungsphase

Zu Beginn der Markierungsphase wird der erste Block aus der Liste für graue Elemente genommen. Da sich zu diesem Zeitpunkt alle lebenden Wurzelreferenzen auf dem Heap befinden, werden auch diese früher oder später erreicht. Es wird festgestellt, welche Adresse der Farbeintrag dieses Blocks hat, um ihn aus der Liste der grauen Blöcke zu entfernen. Der Block wird sodann schwarz markiert. Anschließend wird der Bitvektor für diesen Block untersucht, und im folgenden alle Referenzen, die in weiß markierte Blöcke zeigen zur Liste der grauen Blöcke hinzugefügt. Das stellt das Ende eines Markierungsinkrements dar. Es läßt sich eine kleine WCET für zu verrichtende Arbeit angeben.

L. Aufräumphase

Die Aufräumphase Sweep Phase traversiert alle Blöcke und fügt die weißen Blöcke zur free-List hinzu, und markiert alle schwarzen Blöcke wieder weiß. Ist ein Block frei, so wird er übersprungen. In der JamaicaVM wird eine spezielle Farbcodierung für freie Blöcke benutzt - die Farbe ”free”. Siebert bemerkt selbst, dies stelle einen Mißbrauch der Farbmetapher dar, meint aber zugleich, es würde dem Verständnis zuträglich sein. Sollten ohnehin zusammenhängende Bereiche des Speichers frei geworden sein, so werden diese ”verschmolzen”. Das geschieht durch Speichern der Länge des freien Bereichs im ersten Block des Bereichs. Der Collector ist nun in der Lage, große Bereiche freien Speichers überpsringen zu können. Es muss die Invariante gelten, dass es keine zwei aufeinanderfolgenden freie Bereiche gibt. Diese werden immer verschmolzen. Während der Aufräumphase gibt es keine grauen Blöcke, und so muss auch die Schreibbarriere keine Blöcke grau markieren.

M. Garantien

Die durchgeführten Modifikationen an der Mark & Sweep Technik, sowie die Einführung eines fragmentierungsabweisenden Speicheraufteilungsschemas, als auch die vorgeschlagene zeitbasierte Schedulingstrategie führen zur genauen Feststellbarkeit von WCET und WCPT. Es können somit nach statischer Analyses des kompilierten Programms genaue Angaben bezüglich dieser Größen gemacht werden. Siebert gibt in [SIE02] auch formale Beweise für diese Feststellbarkeit an (insbesondere in Kapitel 9 und 10).

N. Sprachfeatures

Im Gegensatz zu bisherigen, limitierten Echtzeit-Implementierungen für Java bietet die JamaicaVM volle Sprachunterstützung an. Alle nützlichen Elemente, wie dynamische Speicherverwaltung, Vererbung, Reflection, dynamisches Classloading, JNI, etc... sind verwendbar. Eine Testversion kann auf der Website ”www.aicas.com” heruntergeladen werden.

VI. Zusammenfassung

Zu guter Letzt möchte ich hier noch einmal feststellen, dass diese Arbeit einen *reinen Überblick* über die Schwierigkeiten beim Einsatz von Garbage Collectoren in Echtzeitumgebungen und deren Umgehung geben soll. Besagter Überblick über drei bestehende, hart echtzeittaugliche Garbage Collectoren wurde, so hoffe ich ausreichend gewährleistet. Eine besonders interessante Implementierung einer JavaVM, die JamaicaVM wurde näher betrachtet.

Ich möchte zusammenfassend vielleicht bemerken, dass echtzeittaugliche Garbage Collection Techniken nicht so sehr vom implementierten Prinzip abhängig sind. Von weitaus größerer Wichtigkeit schienen mir die Fragen nach dem richtigen Zeitpunkt und der richtigen Quantität der Garbage Collector-Aktivität zu sein.

Echtzeittauglichkeit einer Garbage Collector-Implementierung ist wahrscheinlich also primär ein Scheduling-Problem. Ausserdem kommt es auf die kleinsten Arbeitseinheiten an, die ein Garbage Collector-Algorithmus zu leisten imstande ist. Hält man diese sehr klein, können sie inkrementell vollzogen werden, und es lassen sich vernünftige Aussagen über WCET und WCPT machen.

Danksagungen

Der Autor möchte insbesondere seinen Kollegen Markus Straub und Ingomar Wesp für ihre Hilfe danken.

REFERENCES

[BAC03] D.F. Bacon, P. Cheng, V.T. Rajan, *A real-time garbage collecor with low overhead and consistent utilization*, ACM SIGPLAN Notices, New Orleans, Ausgabe Jänner 2003

[BAK92] H.G. Baker, *The Treadmill, real-time garbage collection without motion sickness*, ACM SIGPLAN Notices, Vol. 27, Nr. 3, Ausgabe Mai 1992

[BUW90] A. Burns, A.J. Wellings, *Real-Time Systems and Their Programming Languages* Addison-Wesley 1990

[DIJ76] E. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens, *On the fly garbage collection: an exercise in cooperation*, Aus "Lecture Notes in Computer Science, No. 46." Springer-Verlag, New York 1976

[HEN98] R. Henriksson, *Scheduling Garbage Collection in Embedded Systems* Ph.D. These, Lund Institute of Technology, Juli 1998

[RAJ95] R. Rajukumar, L. Sha, J.P. Lehoczky,K. Ramamritham, *An optimal priority inheritance policy for synchronization in real-time systems*, Aus Sang H. Son, editor, *Advances in Real-Time Systems*, Prentice Hall 1995

[SIE02] F. Siebert, *Hard Realtime Garbage Collection In Modern Object Oriented Languages*, Dissertation Universität Karlsruhe; Books on Demand Ausgabe Mai 2002

[YUA90] T. Yuasa, *Real-Time Garbage Collection on General-Purpose Machines*, Journal of Systems and Software, Vol. 11, Nr. 3, Ausgabe März 1990

[ZOR92] B. Zorn, *The Measured Cost of Conservative Garbage Collection*, Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado 1992