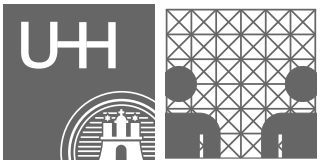

Universität Hamburg, Fachbereich Informatik

Diplomarbeit

Konzepte und Implementierungen moderner virtueller Maschinen

Nebenläufigkeit, automatische Speicherverwaltung und dynamische, optimierende Compiler für Java

Vorgelegt am 19. Dezember 2000

von
Matthias Ernst, Julius-Leber-Straße 27, 22765 Hamburg und
Daniel Schneider, Vereinsstraße 85, 20357 Hamburg

Betreuer:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Bernd Neumann

Danksagung

Für die Unterstützung unserer Arbeit und inhaltliche wie formale Anregungen möchten wir Prof. Dr. J.W. Schmidt und Prof. Dr. B. Neumann danken.

Weiterhin möchten wir uns bei unseren Arbeitskollegen der CoreMedia AG und bei Eberhard Wolff für viele bereichernde Diskussionen bedanken.

19. Dezember 2000

Matthias Ernst und Daniel Schneider

Diese Arbeit ist im Rahmen einer Gruppenarbeit entstanden. Generell wurden dabei die jeweiligen Inhalte der Kapitel von den Autoren einzeln erarbeitet, dann im Team besprochen und evtl. verändert bzw. erweitert. Die Aufteilung stellt sich im einzelnen wie folgt dar:

- Gemeinsam: Kapitel 1 und 6, Anhänge A und B
- Matthias Ernst: Kapitel 5, Abschnitte 2.3, 3.1, 3.2, 3.3
- Daniel Schneider: Kapitel 4, Abschnitte 2.1, 2.2, 3.4, 3.5

Typographische Konventionen

In dieser Arbeit werden unterschiedlich Schriftarten verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist in der folgenden Tabelle wiedergegeben:

Schriftart	**Bedeutung**	**Beispiel**
kursiv	englischer Begriff oder Definition	*header word displacement*
`Typewriter`	Programmtext	`synchronized`

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Virtuelle Maschinen haben sich als Basistechnologie für Programmiersprachen etabliert. Kaum eine moderne Sprachimplementierung verzichtet auf ihren Einsatz — eine „Java Virtual Machine“ (JVM) befindet sich im Lieferumfang praktisch jedes Betriebssystems.

Das Konzept virtueller Maschinen findet bereits seit mehr als drei Jahrzehnten Verwendung. Insbesondere interaktive Entwicklungsumgebungen für LISP, Smalltalk und Self bauen darauf auf. Es muß daher erstaunen, daß die Einführung der Sprache Java im Jahre 1995 diesem Thema zu einem explosionsartigen Wachstum der Forschungsaktivität verhelfen konnte. Dieses Wachstum läßt sich beispielhaft anhand der „ACM Digital Library“ ablesen, die seit 1995 einen sprunghaften Anstieg von Veröffentlichungen zu virtuellen Maschinen verzeichnet (Abbildung 1.1).

Bei näherer Betrachtung lassen sich die Veröffentlichungen grob in drei Kategorien unterteilen:

- Spezielle Eigenschaften der Sprache Java sowie der Einsatz in Server-Umgebungen erfordern Entwicklungsarbeit insbesondere im Bereich der Nebenläufigkeit und Skalierbarkeit.
- Eine Zahl von Techniken existierten bisher lediglich als „proof of concept“, werden jedoch in virtuellen Maschinen für Java zur Produktionsreife gebracht. Hierzu zählt der Bereich dynamischer Optimierungen.
- Weiterhin werden gut erforschte Techniken für den Einsatz in einer JVM angepaßt. Als Beispiele lassen sich etablierte Compiler-Techniken und Algorithmen zur Garbage Collection anführen.

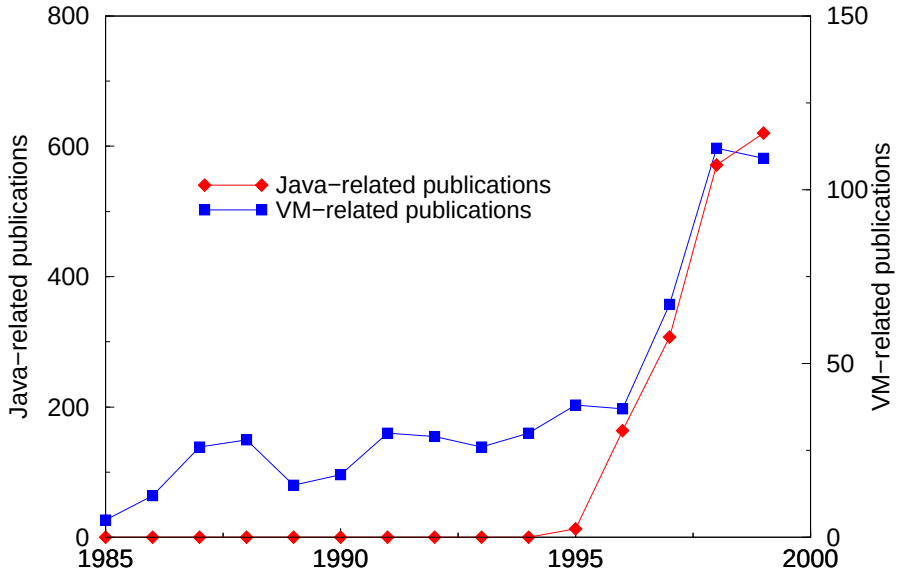

Abbildung 1.1: Veröffentlichungen in der ACM Digital Library

Für eine detaillierte Klärung bedarf es der wissenschaftlichen Einordnung der behandelten Konzepte und Techniken, die in dieser Arbeit vorgenommen wird. Dazu werden ausführlich Architekturen und Realisierungstechniken virtueller Maschinen betrachtet. Das Interesse liegt dabei auf den drei Hauptkomponenten „Programmausführung“, „Speicherverwaltung“ und „Thread-System“.

Es wird gezeigt, daß moderne virtuelle Maschinen hochkomplexe Softwaresysteme darstellen. Aufgrund der engen Verzahnung der Komponenten erfordert ihre Implementierung die Kenntnis einer Vielzahl von Konzepten und Techniken aus allen drei genannten Bereichen. Weiterhin ist ein fundiertes Wissen der Bereiche „Maschinenarchitektur“ und „Betriebssysteme“ erforderlich, um eine effiziente Abbildung der Sprachabstraktionen auf die konkrete Plattform zu realisieren.

1.1 Der Begriff der virtuellen Maschine

Mit dem Begriff der virtuellen Maschine werden vielfach zwei Dinge assoziiert. Zum einen bezeichnet er eine abstrakte Spezifikation, zum anderen eine konkrete Implementierung dieser Spezifikation.

Die Spezifikation einer virtuellen Maschine definiert das Modell einer Maschine zum Ausführen von Berechnungen. Dieses Modell enthält eine Maschinensprache, in der Berechnungsanweisungen für die Maschine erstellt werden können, sowie eine Beschreibung der Semantik der in der Sprache definierten Operationen. Die Art der Operationen impliziert ein bestimmtes Architekturmodell; das gängige Modell ist das der Stackmaschine — sie wird von virtuellen Maschinen wie der Java VM [Gosling, 1995], unterschiedlichen Smalltalk VMs [Goldberg and Robson, 1983] und der Tycoon-2 VM [Gawecki and Wienberg, 1998] verwendet. Ein weniger verbreitetes Modell ist das der Registermaschine — sie wurde beispielsweise von einigen LISP Implementierungen verwendet.

Die Spezifikation der virtuellen Maschine erlaubt die Entkopplung zweier Schritte: zum einen kann „gegen“ die Spezifikation programmiert werden, d.h. es können der Maschinensprache entsprechende Programme erstellt werden. Zum anderen können anhand der Spezifikation konkrete Implementierungen der virtuellen Maschine erstellt werden. Diese Implementierungen sind Computerprogramme, die ihrerseits Programme der abstrakt spezifizierten Maschinensprache ausführen.

Diese Maschinensprache kann dabei entsprechend dem spezifizierten Maschinenmodell unterschiedlich abstrakt sein. Es kann sich beispielsweise um für einen Menschen verständlichen Quellcode oder binär kodierte abstrakte Syntaxbäume handeln. Im Fall der häufig verwendeten Stackmaschinen umfaßt die Maschinensprache Befehle zur Manipulation eines abstrakten Stacks im Zusammenhang mit einem abstrakten Hauptspeicher. Durch das Fehlen von Registern entsteht eine sehr einfache Maschinensprache mit wenigen Befehlen — da aus diesem Grunde jeder Befehl durch nur ein Byte dargestellt werden kann, wird eine solche Maschinensprache häufig als Bytecode bezeichnet. Dieser Bytecode wird üblicherweise von einem Compiler aus dem Sourcecode einer höheren Programmiersprache generiert. Als erster Bytecode wird gemeinhin der von [Nori et al., 1974] erstellte „P-Code-Interpreter“ für die Programmiersprache USCD-Pascal bezeichnet.

Neben der Spezifikation der Programmiersprache liegt somit auch die Spezifikation einer Plattform vor, auf die erstere abgebildet wird. Hierbei können sich Probleme von Inkompatibilitäten oder *impedance mismatches* ergeben. Weiterhin besteht die Möglichkeit, mehrere Programmiersprachen auf die abstrakte Plattform zu übersetzen. Die hiermit verbundenen Möglichkeiten und Probleme sind nicht Teil dieser Arbeit.

Die Aufgabe der Implementierung einer virtuellen Maschine ist eine möglichst effiziente Abbildung des Bytecodes auf die jeweilige konkrete Plattform, auf

der die Maschine abläuft. Anders formuliert, muß die VM das Programm in ähnlicher Zeit und mit vergleichbarem Speicherbedarf ausführen wie ein direkt übersetztes Programm. Dabei erfüllt sie, abhängig von der implementierten Sprache, verschiedene Dienste wie Garbage Collection, Nebenläufigkeit, Überprüfen und Auslösen von Ausnahmen, Sicherheitprüfungen, Aufruf von Funktionen anderer Sprachen und Persistenz.

Für Java wurde die Spezifikation der virtuellen Maschine in [Lindholm and Yellin, 1996] festgehalten. Mittlerweile existiert eine Vielzahl von Implementierungen verschiedener Hersteller (Anhang A beschreibt eine Auswahl). Für die meisten Sprachen existiert hingegen keine gesonderte Spezifikation einer virtuellen Maschine, bzw. die Implementierung stellt die Spezifikation dar.

1.2 Ziel und Aufbau der Arbeit

Diese Arbeit soll über die Architektur und Techniken zur Realisierung virtueller Maschinen Auskunft geben. Dabei wird deutlich, welche Aufgaben und Zielvorstellungen zu der wachsenden Komplexität ihrer Implementierungen beitragen.

Die Auswahl der konkreten Techniken erfolgt dabei nach zwei Kriterien:

1. Ihre Relevanz in aktuellen Implementierungen virtueller Maschinen — dies betrifft vor allem Techniken, die möglicherweise schon länger bekannt sind, die aber einen de facto Standard in ihrem Bereich darstellen.

2. Ihre Relevanz vor dem Hintergrund der aktuellen Forschung. Hier wurden auch Techniken berücksichtigt, die unter Umständen die Grenzen des technisch Machbaren aufzeigen und die (noch) nicht für den Produktionseinsatz geeignet sind.

Auf diese Weise soll die Arbeit sowohl einen Überblick über den status quo gebräuchlicher VM-Implementierungen, als auch eine Vorschau auf mögliche Techniken von virtuellen Maschinen der nächsten Generation geben.

In Kapitel 2 werden zunächst Grundlagen für das Verständnis von virtuellen Maschinen vermittelt. Dabei werden die Hauptkomponenten „Programmausführung“ (2.1) , „Speicherverwaltung“ (2.2) und „Thread-System“ (2.3) in einzelnen Abschnitten behandelt. Leser, die mit den Grundkonzepten der entsprechenden Komponenten vertraut sind, können den jeweiligen Abschnitt überspringen.

In den darauf folgenden Kapiteln 3–5 werden jeweils fortgeschritte Techniken zur Implementierung von virtuellen Maschinen in Hinblick auf eine der drei genannten Hauptkomponenten vorgestellt. Kapitel 6 fasst zum einen die Ergebnisse der Arbeit zusammen und gibt zum anderen einen Ausblick auf die zu erwartenden Entwicklungen. Im Anhang A finden die im Laufe der Arbeit vorgestellten Begriffe und Techniken zudem Anwendung in Form eines Vergleichs ausgewählter Implementierungen virtueller Maschinen. Ein Glossar der relevanten Begriffe ergänzt die Arbeit.

Kapitel 2

Komponenten virtueller Maschinen

Moderne virtuelle Maschinen beinhalten drei große Subsysteme, die eng miteinander kooperieren. Der vom Quellcode-Compiler erstellte abstrakte Zwischencode muß durch einen Interpreter oder dynamischen Compiler ausgeführt werden – es existiert also eine Komponente zur **Programmausführung**.

Viele Sprachen, die auf virtuellen Maschinen ausgeführt werden, unterstützen den Ablauf mehrerer Kontrollflüsse. Diese Fähigkeit zur **Nebenläufigkeit** kann direkt in der implementierten Sprache verankert sein (z.B. Modula-2 Coroutinen), oder scheinbar ausschließlich durch bestimmte Bibliotheksfunktionen implementiert sein (z.B. Smalltalk „Processes“). In beiden Fällen muß die virtuelle Maschine den Ablauf mehrerer Kontrollflüsse unterstützen, da die Konsistenz interner Strukturen der VM bei nebenläufigem Zugriff gewährleistet sein muß.

Weiterhin setzen moderne Sprachen eine automatische **Speicherverwaltung** voraus – hierunter fällt die effiziente Allokation von Speicher sowie Garbage Collection, die unerreichbar gewordenen Speicher wieder dem freien Speicher zuführt.

In diesem Kapitel werden die Aufgaben der drei Subsysteme analysiert, das entsprechende Grundlagenwissen eingeführt und die vielfältigen Anknüpfungspunkte zwischen den Komponenten aufgezeigt. Dies geschieht vor allem im Kontext objektorientierter Sprachen, insbesondere Java.

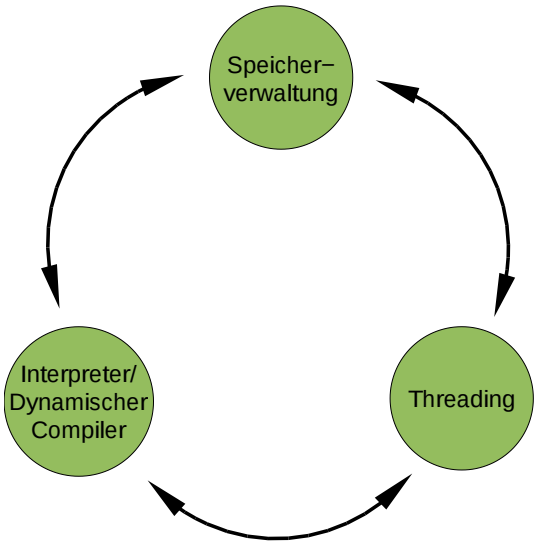

Abbildung 2.1: Die drei Hauptkomponenten einer virtuellen Maschine

2.1 Programmausführung

Für das Verständnis der ausführenden Komponente einer virtuellen Maschine ist zunächst die Ausgangsbasis von Interesse: der Instruktionssatz und das damit verbundene Ausführungsmodell der abstrakten Maschine. Im folgenden Abschnitt 2.1.1 wird daher zunächst das Modell der JVM diskutiert und anschließend die Vorgehensweisen zur optimierten Übersetzung von Bytecode in Maschinencode erarbeitet.

Dazu werden in den beiden folgenden Abschnitten die spezifischen Probleme die bei der Optimierungen von Java als objektorientierter Sprache (2.1.2) sowie die Implikationen dynamischen Ladens von Code und der damit verbundenen dynamischen Übersetzung (2.1.3) geschildert.

Abschnitt 2.1.4 faßt die Ergebnisse zusammen und gibt einen Ausblick auf Kapitel 3, das Implementierungstechniken von optimierenden Ausführungseinheiten gewidmet ist.

2.1.1 Das Maschinenmodell der JVM

Die für Java spezifizierte virtuelle Maschine ist im Groben in Abbildung 2.2 dargestellt und mit den relevanten Instruktionsnamen versehen. Auf einem gemeinsamen Objektspeicher operieren mehrere Threads; jeder dieser

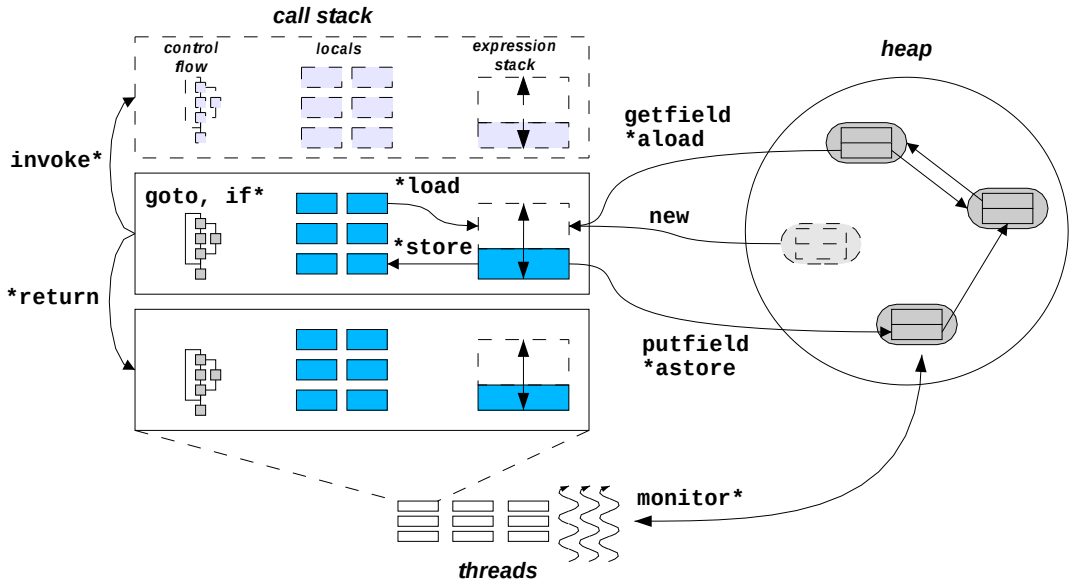

Abbildung 2.2: Ausführungsmodell von Java

Threads besitzt einen eigenen Aufruf-Stack. Jede Aktivierung innerhalb eines Stacks beinhaltet zwei Speicherbereiche: lokale Variablen und einen Stack für Berechnungen. Dieser Stack ist von zentraler Bedeutung: er ist Operanden- und Ergebnisspeicher für arithmetische Operationen, wird gleichermaßen zur Parameter- und Ergebnisübergabe verwendet und ist Zwischenstation beim Transfer zwischen lokalen Variablen und Heap. In dieser Hinsicht ist das Java-Modell mit dem von Smalltalk identisch und wird als *Stack-Maschine* bezeichnet.

Mit einer Zahl von Bytecode-Instruktionen kann der Kontrollfluß innerhalb einer Methode gesteuert werden: bedingte und unbedingte Sprünge, sowie Ausnahmebehandlung und sogar Subroutinen. Weiterhin existieren zwei Instruktionen zur Synchronisierung von Threads.

Die virtuelle Maschine von Java ist getypt. Sie unterscheidet mehrere Basisdatentypen sowie Objekt- und Array-Referenzen. Der Instruktionssatz enthält für jeden Typ Varianten der Instruktionen, beispielsweise existieren fünf Varianten, um eine lokale Variable auf den Stack zu laden. Auf diese Weise erklärt sich die Zahl von rund 200 Instruktionen.

Um die Integrität der virtuellen Maschine zu gewährleisten, beispielsweise, um zu verhindern, daß eine Ganzzahl fälschlicherweise als Objektreferenz interpretiert wird, muß die VM unter anderem eine Typprüfung des Bytecodes

vornehmen. Nur Code, der eine solche Prüfung zulässt und dabei den Typregeln entspricht, wird ausgeführt. Bei näherer Betrachtung wird deutlich, daß die Typisierung des Bytecode daher keinerlei Vorteil darstellt: zu Verifikationszwecken wird der Typ jeder Variablen ohnehin rekonstruiert und bräuchte daher nicht in jeder Instruktion mit angegeben werden — eine Typisierung der Objektfelder und Methodensignaturen würde hier ausreichend Informationen liefern und den Bytecode schlanker gestalten. Die Typisierung der Variablen ist auch für die Speicherverwaltung von Interesse, näheres hierzu findet sich in Abschnitt 2.2.2.

Die Spezifikation sieht das inkrementelle Laden und Binden von Klassen vor. Hierbei muß die Kompatibilität der Klassen untereinander geprüft werden.

2.1.2 Optimierung von Java-Programmen

In diesem Abschnitt sollen die Möglichkeiten und Probleme der Optimierung von Java-Programmen durch eine optimierenden Compiler, der nativen Maschinencode erzeugt, dargestellt werden. Dabei zeigt sich zunächst, daß die Bytecode-Repräsentation von Java-Programmen nicht im Hinblick auf optimierende Compiler entworfen wurde. Während der abstrakte Instruktionssatz zur Ausführung durch einen Interpreter durchaus geeignet erscheint, sind wichtige Informationen durch die lineare Repräsentation des Codes und durch unstrukturierten Kontrollfluß stark verschleiert, da die Abbildung von Java-Quellcode auf Bytecode nicht strukturerhaltend ist. Es erfordert aufwendige Analysen, um beispielsweise die Struktur einer `for`-Schleife und die damit verbundenen Informationen über die Schleifenvariable zu rekonstruieren. Gängige Compileranalysen suchen gar nach spezifischen Bytecodemustern, wie sie von Quellcodecompilern wie `javac` oder `jikes` erzeugt werden.

Das nahezu klassische Problem bei der Optimierung objektorientierter Sprachen besteht in ihrem auszeichnendem Mechanismus: in reinen objektorientierten Sprachen (Smalltalk-80, Tycoon-2) sind alle Datentypen Teil einer Klassenhierarchie, die durch einen Vererbungsbeziehung verbunden ist (*„everything is an object“*). Alle Operationen auf diesen Datentypen werden durch dynamischen Methodeaufruf, d.h. der empfängertyp-spezifischen Auswahl einer Methode zur Laufzeit, implementiert. Dementsprechend besteht Programmcode dieser Sprachen praktisch ausschließlich aus Methodenaufrufen, die ihren Abschluß in systemeigenen oder extern implementierten Methoden haben.

Java ist im Gegensatz zu den o.g. Sprachen keine reine objektorientierte Sprache. Sie kennt, wie auch C++, sogenannte Basisdatentypen (*integer,*

float, byte, etc.), die es erlauben, Berechnungen mit diesen Typen ohne dynamischen Methodenaufruf zu implementieren. Außerdem ist es in Java möglich, den statische Methodenbindung zu erreichen, indem eine Methode als `private`, `static` oder `final` deklariert wird. Generell gilt aber: wird ein objektorientierter Programmierstil angewendet, tritt auch in Java-Programmen eine hohe Dichte an dynamischen Methodenaufrufen auf. Diese sind aus zwei Gründen relativ ineffizient gegenüber ihren statisch gebundenen Pendants:

1. Die zu Empfänger und Signatur der Nachricht passende Methode muß zunächst gefunden werden. Dazu wird in einer „naiven“ Implementierung die Methodentabelle der Empfängerklasse und ihrer Superklassen nach der entsprechenden Nachricht durchsucht. Findet die Maschine hier keinen Eintrag, so wird eine entsprechende Ausnahme ausgelöst. Bei einer solchen „naiven“ Lösung wird also bei jedem Versenden einer Nachricht eine große Zahl von Instruktionen ausgeführt.

2. Der anschliessend ausgeführte Sprung ist indirekt, d.h. er führt an eine berechnete Adresse. Moderne Prozessoren sind hingegen auf eine Vorhersage des Kontrollflusses angewiesen, um die tiefen *Pipelines* (ihre Ausführungseinheiten) befüllen zu können. Aufgrund schlechter Vorhersagetechniken im Prozessor führt eine hohe Dichte indirekter Sprünge zu häufigem Stillstand der *Pipelines*[Driesen et al., 1995].

Insbesondere führt jedoch der hohe Anteil an Sprüngen zu sehr kleinen optimierbaren Codeeineheiten. Dies behindert die Anwendung praktisch aller aus dem klassischem Compilerbau bekannten Optimierungen, da diese Seiteneffekte der gerufenen Methoden nicht in Betracht ziehen können und daher viele Optimierungen über Aufrufe hinweg nicht möglich sind.

Diese „Grundprobleme“ der Optimierung objektorientierter Sprachen werden seit langem erforscht und es existieren verschiedene Ansätze, um sie zu mildern. Die wichtigsten Techniken sind *virtual function tables*, *inline caching* und *inlining* (für eine genauere Betrachtung siehe [Driesen, 1999]):

virtual function tables (vtables) reduzieren den Aufwand der Methodenauswahl. Dazu wird den Methoden im Bereich einer Vererbungshierarchie eine fortlaufende Nummer zugewiesen, wobei in jeder Klasse mit der höchsten Nummer der Superklasse begonnen wird. Methoden, die bereits in einer Superklasse deklariert wurden, erhalten keine neue Nummer.

Enthält eine Klasse m (ererbte oder implementierte) Methoden, so sind diese von $0..m-1$ durchnumeriert. Somit läßt sich eine einfach indizierbare Methodentabelle (*vtable*) erstellen. Der dynamische Methodenaufruf reduziert sich darauf, die zum Empfänger gehörige *vtable* zu laden und über den Methodenindex die Adresse der passenden Methode zu erhalten.

inline caching: Für einen bestimmten Aufrufort wird die Adresse der zuletzt gewählten Methode zusammen mit der Klassen-ID des Empfängers *inline*, also direkt im erzeugten Code, gespeichert. Soll der Aufruf erneut ausgeführt werden, so wird zunächst die Empfängerklasse mit der zuvor gespeicherten verglichen. Stimmen diese überein, so kann direkt an die zuvor gespeicherte Methodenadresse verzweigt werden. Andernfalls wird eine normale Methodensuche ausgeführt, und das Ergebnis wird an Stelle der alten Daten gespeichert.

inlining: Kann zur Übersetzungszeit entschieden werden, welche Methode an einer bestimmten Aufrufstelle mit großer Wahrscheinlichkeit aufgerufen werden wird, so kann der Methodenkörper — unter Beachtung gewisser Einschränkungen — in den Methodenkörper der aufrufenden Methode eingebettet werden. Diese Methode entspricht im Wesentlichen der Prozedurintegration eines klassischen Compilers, es muß jedoch wie beim *inline caching* ein Test in den Code integriert werden, ob die zur Übersetzungszeit getroffene Annahme über die Empfängerklasse zutrifft. Anderfall muß eine normale Methodensuche durchgeführt werden.

Vtables erlauben die Durchführung der Methodensuche mit zwei Indirektionen (*receiver* $\rightarrow$ *vtable* $\rightarrow$ *method*). Allerdings wird ein indirekter Sprung ausgeführt, der genannte Probleme im Zusammenhang mit modernen Prozessoren aufweist.

Durch *inline caching* wird im Erfolgfall nur eine Indirektionen zum Auffinden der Methode benötigt. Noch wichtiger ist aber, daß das folgende Sprungziel dem Prozessor direkt im Maschinencode zur Verfügung steht. Statt indirekter Sprungvorhersage muß das Ergebnis des Klassentests vorhergesagt werden; diese binäre *branch prediction* wird im Allgemeinen weit besser unterstützt.

Inlining vermeidet im Vergleich zu *inline caching* noch den Aufwand eines Methodenaufrufes. Zudem wird dabei die zur Verfügung stehende Codeeinheit vergrößert und ermöglicht damit eine bessere Anwendung klassischer Optimierungen. Allerdings muß die resultierende Codegröße in Betracht gezogen

werden, weshalb unbegrenzte Anwendung von *inlining* nicht gewinnbringend ist. Hierzu sind entsprechende Heuristiken anzuwenden, die in Abschnitt 3.2 beleuchtet werden.

Ein weiterer Umstand, der die Optimierung von Java-Programmen durch ein Erzwingen vieler bedingter Sprünge behindert, betrifft die Behandlung von Ausnahmen. Dieser begründet sich aus der Kombination zweier Faktoren:

- Wie die meisten Programmiersprachen mit Ausnahmebehandlung erfordert Java eine *präzise* Implementierung dieses Mechanismus — dies ist in [Gosling et al., 1996] wie folgt definiert:

  > *Exceptions in Java are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the Java program.*

  Kurz gesagt, müssen sowohl der Typ einer ausgelösten Ausnahme als auch der Programmzustand den zu erwartenden Größen bei einer sequentiellen Bearbeitung des Quellcodes entsprechen.

- In Java-Programmen kann eine Vielzahl von Sprachkonstrukten aufgrund implizit assoziierter Laufzeittests eine Ausnahme auslösen. Dazu zählen Tests auf die Null-Referenz im Fall von Methodenaufrufen und Feldzugriffen, Index-Überprüfungen bei Array-Operationen, arithmetische Fehler und andere.

Stellt man Ausnahmen als zusätzliche Kanten im Kontrollflußgraph dar, so erhält man einen Graphen mit vielen relativ kleinen *basic blocks*, die für sich wenig Gelegenheit zur Optimierung bieten.

Ein Beispiel verdeutlicht das Problem: In Abbildung 2.3 wird eine Methode in Java-Code (i) und in einer optimierten Zwischenrepräsentation[1] (ii) dargestellt. Der Ausdruck `1/c` in (i-4) ist schleifenkonstant; seine Auswertung

[1] Hier wurde der Einfachheit halber Java als Zwischenrepräsentation gewählt. Ein Compiler wird natürlich eine angemessenere interne Repräsentation wählen.

```
(i)         void fill(Object[] a, Float[] b, int c){
     1:         for(int i = 0; i < a.length; i++){
     2:             int hash = a[i].hashCode()
     3:             a[i] = null;
     4:             b[i] = new Float(hash * 1/c);
     5:         }
            }


(ii)        void fill(Object[] a, Float[] b, int c){
     1:         int x = a.length;
     2:         int i = 0;
     3:     L0: if(i >= x) goto L1;
     4:         Object y = a[i];
     5:         int hash = y.hashCode(); <--
     6:         a[i] = null;               |
     7:         int w = 1/c;          -------
     8:         float z = hash * w;
     9:         b[i] = new Float(z);
    10:         i = i + 1;
    11:         goto L0:
    12:     L1:
            }
```

Abbildung 2.3: Darstellung einer Java-Methode in unoptimierter Form (i) und in einer optimierten Zwischenrepräsentation (ii)

sollte daher vor der Schleife erfolgen. Darstellung (ii) verdeutlicht jedoch die Abhängigkeit der Instruktion (ii-7) von (ii-5). Enthielte nämlich `a` an erster Position `null`, so muß korrekterweise eine `NullPointerException` ausgelöst werden, und keine `ArithmeticException` im Falle von `c == 0`. Wäre die Länge von `a` beispielsweise gleich 0, so dürfte ein Nullwert von `c` gar keine Ausnahme nach sich ziehen.

Ein ähnliches Problem erwächst theoretisch aus der Definition des Java-Speichermodells [Gosling et al., 1996]. Wie in Abschnitt 2.3.2 näher erläutert, ist das Speichermodell schwer verständlich, widersprüchlich und verhindert die Generierung effizienten Codes, da die Umordnung von Instruktionen durch den Compiler stark eingeschränkt sind.

Allerdings wird die aktuelle Spezifikation praktisch von jeder VM verletzt

— darüber hinaus ist ein neues Speichermodell in der Entwicklung. Insofern stellt das aktuelle Java Memory Model kein faktisches Problem für die Implementierung von Optimierungen dar.

Ist es gelungen, durch geeignete Techniken die Abstraktionseinheiten ausreichend zu vergrößern, so ist eine anschließende Optimierung nach klassischen Techniken eines Compiler-Backends möglich. Diese Techniken verfolgen dabei drei Ziele:

1. Es wird versucht, redundante Berechnungen zu vermeiden. Wird ein Ausdruck in einer Funktion mehrfach ausgewertet, so kann dies durch Zwischenspeicherung des Ergebnisses vermieden werden.

2. Der Zugriff auf den Hauptspeicher zum Auslesen von Daten stellt bei der Ausführung in modernen Prozessoren einen wachsenden Engpaß dar. Auch moderne Architekturen mit *pipelines* können dieses Problem nur mildern. Indem optimierter Code Register verwendet und in einem geeigneten Zugriffsmuster auf den Cache zugreift, werden die Zugriffe auf den Hauptspeicher minimiert.

3. Superskalare Prozessoren verfügen über mehrere Ausführungseinheiten, die parallel Aufgaben ausführen können. Um die Parallelität dieser Architekturen auf Instruktionsebene besser auszunutzen, müssen die Abhängigkeiten der einzelnen Instruktionen voneinander verringert werden.

Die verwendeten Optimierungen umfassen Techniken wie „Constant Folding and Propagation“, „Algebraische Vereinfachungen“, „Common Subexpression Elimination“, „Loop Unrolling„, „Software Pipelining“ und Registerallokation (siehe dazu Anhang B).

Insgesamt läßt sich festhalten, daß die Optimierung eines Java-Programms aus den beiden Komplexen der objektorientierten Optimierung und der klassischen Compiler-Optimierung besteht. Erst wenn das durch dynamische Methodenaufrufe und Besonderheiten der Java-Sprachdefinition [Gosling et al., 1996] hervorgerufene Problem kurzer optimierbarer Einheiten gelöst ist, können die klassischen Optimierungen greifen. Der dynamische Compiler einer virtuellen Maschine muß diese beiden Komplexe implementieren, um effizienten Code zu generieren.

2.1.3 Dynamische Übersetzung

Auf den ersten Blick scheint die Verwendung dynamischer Compiler, die erst zur Laufzeit Übersetzungsvorgänge durchführen, nur Nachteile im Vergleich zu ihren statischen Pendants mit sich zu bringen. Sie brauchen Rechenzeit, die eigentlich nutzbringender im Anwendungscode verbracht würde, und sie belasten den Hauptspeicher durch ihren eigenen Objektcode sowie umfangreiche Datenstrukturen, die zur Übersetzung benötigt werden. Warum also stellt dynamische Übersetzung nichtsdestotrotz die überwiegende Ausführungstechnik virtueller Maschinen dar?

Zunächst bietet die Java-Plattform die Möglichkeit, das Programm zur Laufzeit zu erweitern. Statische Systeme unterstützen diesen dynamisch geladenen Code maximal durch Interpretation. Bei Anwendungen, die fremden oder dynamisch erzeugten Code integrieren, wie z.B. Applikationsserver, läßt sich der genaue Umfang aktiver Klassen gar nicht bestimmen, so daß ein statischer Übersetzungsprozeß immer Gefahr läuft, relevante Teile zu ignorieren.

Weiterhin hat ein dynamischer Compiler exaktes Wissen über die ausführende Maschine. Besonderheiten wie der Instruktionssatz oder die Zahl der Prozessoren, lassen sich vorteilhaft zur Codeerzeugung auswerten. Beispielsweise unterscheidet der dynamische Compiler der Hotspot VM zwischen SPARC-Prozessoren der Revisionen V8 und V9; letztere bietet „Compare-And-Swap“- sowie „Conditional-Move“-Instruktionen. Gleiches wäre beispielsweise für dedizierte Hardware wie die MMX-Erweiterungen der Intel IA32-Architektur denkbar.

Der Hauptvorteil eines dynamischen Compilers liegt jedoch in der parallelen Ausführung mit dem Programm begründet: *Optimierungsentscheidungen können auf Basis des Programmzustands getroffen und gegebenenfalls sogar revidiert werden.* Das bedeutet, ein dynamischer Compiler hat gegenüber einem statischen einen Informations-[2] und Flexibilitätsvorsprung, denn ihm stehen zur Laufzeit folgende Informationen zur Verfügung:

- Die Menge der tatsächlich benutzten Klassen. Dies kann für statische Bindung und Inlining von Methodenaufrufen genutzt werden[Dean, 1996].
- Das Aufrufprofil polymorpher Aufrufstellen. Auch dieses lässt sich für statische Aufrufbindung und Inlining-Entscheidungen

[2]Statische Compiler könnten theoretisch auf ein Profil des letzten Programmlaufs zurückgreifen — eine beliebte Technik von Herstellern von C-Compilern zur Erzielung hoher SPEC-Benchmarks.

nutzen[Hölzle et al., 1991].

- Laufzeitkonstanten. Wird eine Methode für einen Wert einer Variablen spezialisiert, können u.U. enorme Gewinne durch anschließende Optimierungsschritte wie *constant-propagation* und *strength-reduction* erzielt werden. Allerdings existieren keine Lösungen, die Spezialisierungsziele automatisch zu ermitteln; hier bieten sich deklarative Ansätze an, die in Java allerdings nicht vorliegen[Poletto et al., 1998]. Die Spezialisierung auf Parametertypen wird mit Erfolg durch [Gawecki, 1991] angewendet.

Insbesondere der erste Punkt unterliegt einer bedeutsamen Einschränkung. Durch das dynamische Laden von Klassendefinitionen läßt sich der Umfang eines Java-Programms beliebig erweitern. Der zuvor erzeugte Code könnte somit nicht länger korrekt sein. Daher muß die Laufzeitumgebung folgendes sicherstellen: Code, der auf einer — nun widerlegten — Annahme über das Programm beruht, darf nicht länger ausgeführt werden; zumindest jedoch mit keinem Exemplar der neu geladenen Klassen. Techniken zur Feststellung und Bereinigung solcher Situationen werden in Abschnitt 3.3 weiter diskutiert.

Grundsätzlich sei darauf hingewiesen, daß durchaus hochoptimierende statische Übersetzer für Java existieren. Produkte wie *TowerJ*[TowerJ, 2000] oder *NaturalBridge BulletTrain*[NaturalBridge, 2000] belegen, daß alternative Ausführungsmodelle realisierbar sind. Immerhin erzielte TowerJ lange Zeit die beste Performance im Volano Benchmark[Volano, 2000].

2.1.4 Zusammenfassung

Die optimierte Übersetzung einer objektorientierten Programmiersprache wie Java kann zweischichtig betrachtet werden:

- Eigenschaften wie maschinennahe Basistypen, vordefinierte Kontrollstrukturen und native Arrays eröffnen die Möglichkeit, wohlverstandene und allgemeingültige „klassische“ Optimierungstechniken anzuwenden, und machen somit die Erzeugung hochperformanten Codes mit durchaus vergleichbarer Geschwindigkeit zu Sprachen wie C möglich.
- Um die Voraussetzungen für deren nutzbringende Anwendung zu schaffen, sind allerdings komplexe und unter Umständen nur transient gültige Transformationen auf höherer Ebene vonnöten. Diese betreffen Spezifika der Sprache Java wie präzise Ausnahmebehandlung, *dynamic dispatch* und das Speichermodell.

Weitergehende Optimierungen betreffen die potentielle Eliminierung von Synchronisierungsoperationen sowie eine optimierte Speicherverwaltung thread-lokaler Objekte.

Um die dynamischen Aspekte der Java-Plattform vollständig zu unterstützen sowie profilgesteuerte Optimierungen zu ermöglichen und auch zu revidieren, ist die Implementierung einer dynamischen Compilerkomponente zwingend erforderlich.

Kapitel 3 untersucht daher die Besonderheiten dynamischer Übersetzung sowie Ansätze, die Spezifika der Sprache Java in effizienten Code zu übersetzen. Es werden ein Kostenmodell sowie die Verwendung dynamischer Profilinformationen zur Steuerung des Übersetzungsprozesses dargestellt. Weiterhin werden einige globale Analysen erläutert, die über den Bereich einer Methode hinausgehen und somit komplexere Transformationen erlauben.

2.2 Speicherverwaltung

Die Speicherverwaltung einer virtuellen Maschine ist eine Komponente, deren Aufgabe die Zuweisung und Überwachung des gesamten von der Maschine benutzten Speichers ist.

In diesem Abschnitt werden die Grundbegriffe der Speicherverwaltung virtueller Maschinen aufgezeigt. Dazu wird in 2.2.1 zunächst auf den Bereich der Allokation und daraufhin in 2.2.2 auf den Bereich der Garbage Collection eingegangen. Bei der Darstellung der Grundbegriffe werden die Grundanforderungen, die an die Speicherverwaltung einer modernen virtuellen Maschine gestellt werden, herausgearbeitet und in 2.2.3 zusammengefaßt.

Eine gute Vertiefung des Themas Speicherverwaltung ist in [Jones and Lins, 1996] zu finden; einen Überblick zu diesem Bereich bietet [Wilson, 1992].

2.2.1 Allokation

Unter Speicherallokation wird die Zuordnung von Speicherbereichen zu Programmsymbolen verstanden. Der Unterteilung von [Jones and Lins, 1996] folgend unterscheiden wir prinzipiell drei Methoden der Speicherallokation:

1. Statische Allokation: Der benötigte Speicher wird zur Übersetzungszeit statisch vergeben, d.h. die Symbole im Programmtext werden bereits

vom Compiler einer bestimmten (u.U. relativen) Speicheradresse zugeordnet. Sprachen, die allein statische Allokation verwenden, können keine Rekursion unterstützen, da die lokalen Variablen einer Funktion nur einmal angelegt werden. Weiterhin muß die Größe einer Datenstruktur bereits zum Zeitpunkt der Übersetzung bekannt sein, d.h. dynamische Datenstrukturen können mit dieser Form der Allokation nicht implementiert werden. Ausschließlich statische Allokation wird z.B. von Fortran 77 [Nyhoff and Leestma, 1995] verwendet – modernere imperative Sprachen wie Pascal und C verwenden sie lediglich für die Allokation von globalen Variablen.

2. Stack-Allokation: Bei der Stack-Allokation werden Parameter und lokale Variablen beim Aufruf einer Funktion auf einem Stack abgelegt. Der auf dem Stack hierdurch belegte Bereich wird Rahmen (*frame*) genannt. Nach Beendigung der Funktion wird der Rahmen wieder abgebaut, d.h. der Speicherbereich steht der nächsten gerufenen Funktion zur Verfügung. Diese Form der Allokation eignet sich besonders gut für prozedurale Sprachen. Ihre Vorteile gegenüber der statischen Allokation sind im Wesentlichen:

   - Rekursion: Da die lokalen Variablen nicht direkt der Funktion sondern einer Aktivierung zugeordnet sind, können Funktionen auch rekursiv verwendet werden. Der Code einer Funktion wird so von ihren Daten getrennt und erlaubt so mehr als eine Aktivierung zur Zeit.
   - Datenstrukturen dynamischer Größe: da ein Rahmen eine dynamische Größe hat, ist es möglich, den Speicherbedarf der enthaltenen lokalen Variablen erst zur Laufzeit festzulegen. Somit könnte also die Größe eines Arrays, das auf dem Stack angelegt wird, erst zur Laufzeit festgelegt werden.

   Eine weitere Konsequenze von Stack-Allokation ist, daß eine gerufenen Aktivierung niemals länger als die aufrufende existieren kann. Dies ist die Grundannahme, die zwar blockstrukturierten Sprachen zugrundeliegt, in (meist funktionalen) Sprachen mit Funktionsabschlüssen (*closures*) jedoch nicht mehr gilt. Weiterhin stellt die Tatsache, daß die Größe des Rückgabewertes zur Übersetzungszeit feststehen muß, und damit für eine bestimmte Funktion konstant ist, eine erhebliche Begrenzung im Programmiermodell dar.

3. Heap-Allokation: Der Hauptnachteil der Stack-Allokation ist die Tatsache, daß dynamische Datenstrukturen nicht als Rückgabewert einer

Funktion dienen können. Dieses Problem wird durch Heap-Allokation (oftmals als „dynamische Speicherverwaltung“ bezeichnet) behoben. Hier werden mit Hilfe einer Bibliotheksfunktion, die das Laufzeitsystem der verwendeten Sprache bereitstellt (in Pascal z.B. `new`), Objekte beliebiger Größe und unbegrenzter Lebenszeit in einem ausgezeichnetem Speicherbereich, dem Heap, angelegt. Dadurch können dynamische Datenstrukturen angelegt werden und auch als Rückgabewert einer Funktion dienen.

Heap-Allokation wirft jedoch neue Probleme auf: da die Verwendung des Speichers nicht mehr statisch vom Compiler ermittelt werden kann, muß die Speicherfreigabe ebenso wie die Allokation unter Programmkontrolle erfolgen. Dabei ist zu ermitteln, welche Speicherbereiche freigegeben werden dürfen und welche vom laufenden Programm noch benötigt werden. Klassische imperative Sprachen wie Pascal und C überlassen diese Entscheidung dem Programmierer, indem sie explizite Bibliotheksfunktionen bereitstellen, mit deren Hilfe ein Speicherbereich als unbenutzt markiert und somit dem Laufzeitsystem „zurückgegeben“ wird (in Pascal z.B. `dispose`). Das Programmiermodell der prozeduralen, imperativen Programmierung baut ohnehin hauptsächlich auf Stack-Allokation auf, so daß diese „Unannehmlichkeit“ akzeptiert wird.

Mit der Verwendung von abstrakten Datentypen in modularen, funktionalen und objektorientierten Sprachen führt diese „manuelle Speicherverwaltung“ jedoch zu erheblichen Problemen, da die Zuständigkeit für die Freigabe eines Speicherbereiches nicht mehr eindeutig festzulegen ist. Funktionale und objektorientierte Sprachen wie Lisp, Smalltalk und Java sind daher mit automatischer Speicherverwaltung ausgestattet, die selbst ermittelt, welche Programmobjekte nicht mehr verwendet werden können, und diese daraufhin dem vorhandene Freispeicher hinzufügt.

2.2.2 Garbage Collection

Die Nachteile manueller Speicherverwaltung sind hinlänglich bekannt. Wird die Freigabe von nicht mehr benötigtem Speicher dem ablaufenden Programm überlassen, gibt es zwei potentielle Fehlersituationen: Wird die manuelle Speicherfreigabe vom Programmierer vergessen, so kann der Speicherbedarf eines laufenden Programms stetig wachsen, was (endlichen Speicher vorausgesetzt) früher oder später zur Beendigung des Programmes führen muß.

Wird umgekehrt ein Speicherbereich freigegeben, der noch in Benutzung ist, und wird dieser Bereich vom Laufzeitsystem einem anderen Objekt zugeteilt, so führt dies zur „Fehlinterpretation“ eines Bereiches d.h. es tritt ein Typfehler auf [Cardelli and Wegner, 1985]. Auch wenn Hilfsmittel zum Auffinden von Programmfehlern im Bereich der Speicherfreigabe zu Verfügung stehen, ist die Fehlerquelle extrem schwer zu lokalisieren, da der resultierende Laufzeitfehler als Symptom in der Regel erst später als der verursachende Programmfehler auftritt.

Im objektorientierten Paradigma wift manuelle Speicherverwaltung eine Reihe weiterer Probleme auf:

- Manuelle Speicherverwaltung verhindert Datenkapselung: Um festzustellen, wann ein Objekt zu letzten Mal verwendet wird (und somit nicht mehr benötigt wird), ist vielfach globales Anwendungswissen vonnöten. Zu Beispiel kann in einer Funktion, die einen Stack manipuliert, in der Regel nicht entschieden werden, ob ein Element, das durch `pop` entfernt wurde, weiterhin benötigt wird oder nicht.

- Information Hiding und manuelle Speicherverwaltung sind unverträglich: Durch Verstecken der Implementierung eines bestimmten Programmteils sollen die Abhängigkeiten zwischen einzelnen Modulen gering gehalten werden. Insbesondere soll eine veränderte Implementierung einer bestimmten Schnittstelle keine Auswirkungen auf etwaige Klienten haben. Auf der anderen Seite könnte die veränderte Implementierung einer Funktion zur Optimierung bestimmte Objekte replizieren, um z.B. den Durchsatz eines Servers zu erhöhen. Diese Änderung müßten etwaige Klienten jedoch berücksichtigen – die Klienten wären also von der Implementierung der Funktion abhängig.

- Probleme der statischen Analyse von Kontrollflüssen: Um bestimmen zu können, wann ein Objekt nicht mehr benötigt wird, wäre eine statische Analyse des Kontrollflusses vonnöten. Dies ist in prozeduralen Sprachen zumindest in einer konservativen Nährung vielfach möglich. In objektorientierten Sprachen wird aufgrund dynamischer Methodenaufrufe der Kontrollfluß jedoch wesentlich durch die verarbeiteten Daten bestimmt. Somit ist es für den Compiler nahezu unmöglich, die Lebenszeit eines Objekts zu bestimmen.

Die genannten Probleme werden durch die Verwendung eines Garbage Collectors größtenteils behoben. Die Aufgabe des Garbage Collectors ist die Erkennung und Freigabe nicht mehr benötigten Speichers. Hierzu ist üblicherweise

Unterstützung des Compilers und des Laufzeitsystems vonnöten (besonders bei der Frage, ob sich in einer bestimmten Speicherzelle eine Referenz befindet), es bedarf jedoch keinerlei Unterstützung durch den Programmierer. Die Garbage Collection funktioniert also als „Automatic Memory Reclamation“.

Im folgenden Abschnitt werden die Grundbegriffe der Garbage Collection erläutert. Danach werden die Algorithmen zur Garbage Collection „Mark-Sweep“, „Mark-Compact“ sowie „Copying Collection“ kurz vorgestellt. Eine nähere Betrachtung dieser Algorithmen ist hilfreich, um die in Kapitel 4 geschilderten Techniken besser zu verstehen, da praktisch alle aktuellen Implementierungen zur Garbage Collection für virtuelle Maschinen auf einem oder mehreren dieser Algorithmen basieren.

Grundbegriffe der Garbage Collection für virtuelle Maschinen

Allen Implementierungen von Garbage Collection ist ein Modell der „Lebendigkeit“ (*liveness*) von Daten gemein. Um eine Speicherzelle wiederverwenden zu können, muß der Collector beweisen, daß sie vom laufenden Programm (dem sogenannten *mutator*) nicht mehr benötigt wird. Dieser Beweis wird in der Regel auf Basis der Erreichbarkeit dieser Speicherzelle geführt. Ist die Zelle von einer Wurzel im System aus transitiv erreichbar, so wird sie als lebendig (*live*) bezeichnet. Ist dies nicht der Fall, so kann sie von der Anwendung nicht mehr erreicht werden und somit dem Freispeicher zugeführt werden. Als Wurzeln können dabei näherungsweise alle globalen und lokalen Variablen der in der Ausführung befindlichen Funktionen verwendet werden[3]. In [Jones and Lins, 1996] wird die folgende Formel für eine formalere Definition von *liveness* angeführt. Die lebendigen Objekte bezeichnen die kleinste Menge *live*, die unter Referenzierung abgeschlossen ist:

$$live = \{N \in Nodes \mid (\exists r \in Roots.r \rightarrow N) \vee (\exists M \in live.M \rightarrow N)\} \quad (2.1)$$

Dabei bezeichnet $Nodes$ die Menge aller Speicherzellen, $Roots$ bezeichnet die Menge der Wurzeln im System. Um nun die Menge *live* eindeutig bestimmen zu können, muß die Relation „$\rightarrow$“ („referenziert“) konkret definiert und diese Information dem Garbage Collector zugänglich sein. D.h. der Collector muß zum Zeitpunkt der Collection für alle Wurzeln und alle Knoten entscheiden können, welche anderen Knoten von ihnen referenziert werden. Anders ausgedrückt muß er für jede bei der Garbage Collection besuchte Speicherzelle

[3]Die Menge der lokalen Variablen läßt sich teilweise noch weiter einschränken, beispielsweise wegen begrenzter Sichtbarkeitsbereiche

bestimmen können, ob sie eine Referenz oder einen Skalar (z.B. einen Integer) enthält.

Ist der Collector aufgrund mangelnden Wissens über seine Umgebung nicht immer in der Lage, diese Information zu erlangen, so muß er eine sichere Annahme treffen — ein solche Implementierung wird als daher →*konservativ* bezeichnet. Kann hingegen den Typ jeder Speicherzelle (Referenz oder Skalar) zum Zeitpunkt der Garbage Collection bestimmt werden, so spricht man von →*exakter* oder *type accurate* Garbage Collection.

Konservative Garbage Collection wird beispielsweise zur Speicherverwaltung in C-Programmen eingesetzt, da hier der Compiler und das Laufzeitsystem keine Typinformationen über die einzelnen Speicherzellen bereitstellen. Konservative Garbage Collection muß alle Speicherzellen, deren Typ nicht zweifelsfrei bestimmt werden kann, als Referenz behandeln. Da es sich hierbei jedoch um einen „Irrtum“ handeln könnte, es sich also doch um einen Skalar handeln könnte, darf er den Inhalt der Zelle keinesfalls ändern. Daraus ergibt sich einen große Schwäche konservativer Collectoren: sie dürfen in diesem Fall Objekte auf dem Heap nicht bewegen, was bei länger Programmausführung zu einer Fragmentierung des Speichers führt – würden sie nämlich ein Objekt bewegen, so müßten sie alle Referenzen auf dieses anpassenen und würden dabei Gefahr laufen, einen Skalar des ablaufenden Programmes zu verändern, der zufällig wie eine Referenz auf das bewegte Objekt „aussah“.

Diese Problem kann umgangen werden, indem nicht direkte Referenzen sondern Indirektionen (*handles*) verwendet werden. Hierbei werden alle Objekte von einer Tabelle aus referenziert. Referenzen zwischen den einzelnen Objekten werden indirekt als Indizes in die Objekttabelle realisiert. Wird ein Objekt bewegt, so muß lediglich der Eintrag in der Objekttabelle modifiziert werden, da dieser die einzige echte Referenz auf das Objekt darstellt. Diese Methode verlangsamt also die häufige Operation der Dereferenzierung und wird daher in modernen virtuellen Maschinen nur selten verwendet.

In virtuellen Maschinen befinden sich Laufzeitsystem, dynamischer Compiler und Garbage Collector gemeinsam unter der Kontrolle der Maschine, d.h. einen Kooperation zur Erstellung von Typinformationen zur Garbage Collection ist prinzipiell möglich. Da exakte Garbage Collection, wie oben erwähnt, den Vorteil hat, Objekte auf dem Heap bewegen zu können, wird Garbage Collection in modernen virtuellen Maschinen prinzipiell durch diese Technik implementiert.

Da für exakte Garbage Collection Typinformationen zu jeder besuchten Speicherzelle benötigt werden, müssen Laufzeitsystem oder Compiler diese Infor-

mationen bereitstellen. Hier werden prinzipiell zwei Wege unterschieden:

1. Markierung (*tagging*): Jeder Wert enthält die Typinformation selbst. Ein oder mehrere Bits der Repräsentation des Wertes werden für die Speicherung der Typinformation verwendet. Beispielsweise ließe sich das niedrigstwertige Bit folgendermaßen verwenden: ist es gesetzt, so handelt es sich um einen nach links verschobenen Skalar, ist es nicht gesetzt, so handelt es sich um eine Referenz. Um zu garantieren, daß Referenzen immer auf `0` enden, wird der gesamte Heap dabei so verwaltet, daß alle im Heap gespeicherten Objekte an durch vier teilbaren Positionen beginnen (sie sind *word aligned*); moderne Prozessorarchitekturen unterstützen oftmals ohnehin nur solche Wortzugriffe.

   Diese Methode hat den Vorteil, daß die Typinformation eines Wertes direkt in diesem enthalten ist. Damit ist der Typ jedes Wertes stets einfach zu ermitteln — ungeachtet dessen, ob er im Heap, im Runtime-Stack oder in Registern gespeichert ist. Nachteilig erweisen sich der Verlust an Darstellungsbreite oder -Präzision sowie vor allem die zusätzlichen Bitverschiebungen, die vor und nach arithmetischen Operationen zur Konversion ausgeführt werden müssen.

2. Referenztabellen (*reference maps*): Die Typinformation wird in einem zugeordneten Speicherbereich gehalten. Jeder Speicherzelle wird ein bestimmter Bereich zugeordnet, aus der ihr Typ hervorgeht. Dies kann z.B. durch den Verweis eines Heap-Objektes auf seine Klassenbeschreibung zur Beschreibung der Objektfelder geschehen. Für Stacks und Register werden üblicherweise Zuordnungstabellen (*stack maps, register maps*) angelegt, die den Typ des jeweils enthaltenen Wertes beschreiben.

   Diese Methode hat den Vorteil, daß die arithmetischen Operationen der genutzten Hardware unverändert angewandt werden können. Das bedeutet auf der anderen Seite, daß nur an Punkten, für die solche Tabellen vorliegen, eine Garbage Collection durchgeführt werden kann. Solche Punkte heißen dementsprechend „GC Points“.

   Ist der Typ einer Stack- oder Registervariablen nur vom aktuellen Punkt der Codeausführung abhängig, so besteht eine *control point dependency*. Bestimmt hingegen der Weg, auf dem dieser Punkt erreicht wurde, den Typ einer Zelle, so besteht eine →*control path dependency*. Eine Coderepräsentation ohne *control path dependency* ist für die Verwendung von *reference maps* vorzuziehen, da so einer Instruktion fest eine solche Tabelle zugeordnet werden kann – diese Zuordnung kann

ggf. bereits vom Compiler vorgenommen werden. Enthält der Code hingegen *control path dependencies*, so müssen dynamisch Typinformationen gepflegt werden oder eine Garbage Collection an zweideutigen Positionen verhindert werden.

Exaktheit ist also ein wünschenswertes Attribut eines Collectors – um diese Exaktheit zu erzielen ist jedoch ein gewisser Aufwand zu treiben.

Der „Mark-Sweep"-Algorithmus

„Mark-Sweep" ist ein sehr einfacher Algorithmus zur Garbage Collection, der aber in Abwandelungen auch in aktuellen Systemen Verwendung findet. Sein Name ergibt sich aus den beiden Phasen benannt, in denen die Garbage Collection ausgeführt wird:

1. Mark: In der ersten Phase werden alle Objekte, die von den System-Roots erreichbar sind – die also *live* sind – markiert (Siehe Abb. 2.4-(i) und 2.4-(ii)). Üblicherweise werden zu diesem Zweck alle Wurzeln auf einen Stack gelegt und dann die jeweils oberste Referenz bearbeitet, bis der Stack leer ist. Bei der Bearbeitung wird zunächst geprüft, ob das referenzierte Objekt bereits markiert wurde. Ist das nicht der Fall, so wird sein „Mark Bit" gesetzt (dies ist üblicherweise ein Bit im Objektkopf) und alle Referenzen, die in diesem Objekt enthalten sind, werden auf den Stack gelegt. Wenn die Mark-Phase abgeschlossen ist, sind alle erreichbaren Objekte besucht worden, die Komplexität K dieser Phase ist also proportional zur Größe der Menge *live* (siehe Formel 2.1). Es gilt also $K(MarkPhase) \in O(|live|)$. Nach der Mark-Phase geht der Algorithmus in die Sweep-Phase über.

2. Sweep: In dieser Phase wird der Heap linear durchlaufen und unmarkierte Speicherbereiche werden einer oder mehreren Freispeicherlisten zugefügt – folgende Allokationen werden dann aus diesen Listen bedient. Während der linearen Suche durch den Heap werden weiterhin alle Markierungen zurückgesetzt, damit diese bei der nächsten Garbage Collection zur Verfügung stehen (siehe Abb. 2.4-(iii)). Die Komplexität dieser Phase ist proportional zur Zahl der Objekte, und damit gilt $K(SweepPhase) \in O(heapsize)$.

Der Vorteil dieses Algorithmus ist seine Einfachheit (ein Vorteil den man nicht unterschätzen sollte, da die Fehlersuche im Garbage Collector auch

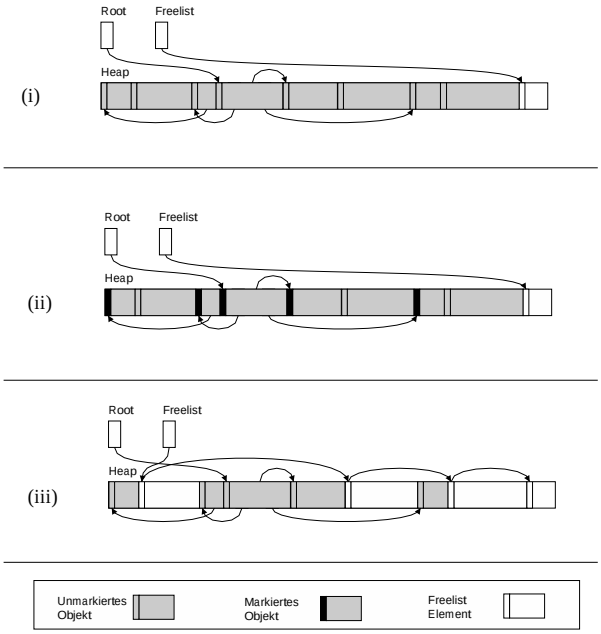

Abbildung 2.4: Ablauf des „Mark-Sweep“-Algorithmus

bei einfachen Algorithmen bereits sehr komplex ist). Er leidet aber unter einer Vielzahl von Nachteilen – so kann der zur Markierung verwendete Stack schlimmstenfalls auf die Größe des Heaps wachsen. Es existieren verschiedene Erweiterungen des Algorithmus, die dieses Problem beheben oder zumindest abmildern. [Knuth, 1973] schlägt vor, den Stack zyklisch zu verwenden, muß allerdings den Heap nach Referenzen von markierten auf unmarkierte Objekte in Kauf nehmen, wenn der Stack leerläuft. [Schorr and Waite, 1967] entwickelten den „Pointer-Reversal“-Algorithmus , der einen Graphen mit konstantem Platz markieren kann. Hierbei geht aber leider die Einfachheit der ursprünglichen Mark-Phase verloren.

Weiterhin tritt bei der Verwendung von Objekten unterschiedlicher Größe eine Fragmentierung des Heaps ein, die zu einem wesentlich größeren Heap führen kann, als eigentlich vom Programm benötigt wird, da ungenutzte Lücken im Heap entstehen (siehe Abb. 2.4-(iii)).

Der „Mark-Compact“-Algorithmus

Um das Problem der Heap-Fragmentierung zu beheben, wurde eine Reihe von „Mark-Compact“-Algorithmen entwickelt [Cohen and Nicolau, 1983]. Diese

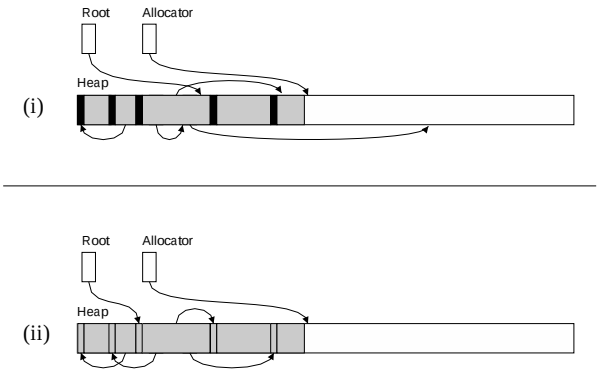

Abbildung 2.5: Kompaktierungsphase (i) und Zeiger-Anpassungsphase (ii) des „Mark-Compact“-Algorithmus

führen die gleiche Mark-Phase wie die „Mark-Sweep“-Algorithmen aus. Danach wird der Heap kompaktiert, d.h. die Objekte werden derart zusammengeschoben, daß die vorhandenen Lücken geschlossen werden (siehe Abb. 2.5-(i)).

Hierzu müssen die beteiligten Objekte natürlich im Speicher verschoben werden, was bedeutet, daß alle betroffenen Referenzen von den Wurzeln und innerhalb des Heaps angepaßt werden müssen. Dies geschieht in der Zeiger-Anpassungsphase („Pointer Update Phase“)[4] (siehe Abb. 2.5-(ii)).

Der Algorithmus läßt sich also in drei Phasen unterteilen:

1. Markierung: Markieren der lebendingen Objekte. Wie bei „Mark-Sweep“ gilt $K(MarkPhase) = O(|live|)$.

2. Kompaktierung: Schliessen der Lücken im Heap durch Verschieben der einzelnen Objekte. Da der gesamte Heap durchlaufen wird, gilt die gleiche Komplexität, wie bei „Sweep“: $K(CompactPhase) = O(heapsize)$.

3. Zeiger-Anpassung: Anpassen der Referenzen aus der Wurzelmenge und dem Heap selbst und gleichzeitiges Zurcksetzen der Markierungsbits. Da hierbei nur die Zeiger in lebendigen Objekten angepaßt werden, gilt $K(UpdatePhase) = O(|live|)$.

[4]Da der Algorithmus Zeiger verändert, kann er nicht für die Implementierung einer konservativen Garbage Collection verwendet werden. Dies gilt für alle kompaktierenden Algorithmen, da sie zur Kompaktierung Objekte verschieben müssen.

Ein kompaktierter Heap hat neben der fehlenden Fragmentierung noch weitere Vorteile gegenüber einer Freispeicherliste: Die Allokation wird deutlich schneller, da lediglich ein Zeiger hochgezählt werden muß, anstatt die Liste zu durchsuchen.

Ein weiterer Vorteil eines kompaktierten Speichers betrifft die Referenzlokalität [Hennessy and Patterson, 1996]. Darunter wird das Maß verstanden, in dem Zugriffe auf benachbarte Speicherzellen auch zeitlich dicht beieinander liegen. Eine hohe Referenzlokalität führt zu einer höheren Trefferquote im Cache und damit zu weniger Speicherzugriffen.

Objekte, die in zeitlicher Nähe erzeugt wurden, werden oftmals auch im weiteren Verlauf nahe nacheinander referenziert. Da der „Mark-Compact"-Algorithmus nacheinander allozierte Objekte nebeneinander ablegt und diese auch bei einer Garbage Collection nicht trennen wird, beeinflußt er die Lokalität des Programmes positiv.

Der „Mark-Compact"-Algorithmus vermeidet also Fragmentierung bei schnellerer Allokation und verbesserter Lokalität – dafür muß der Algorithmus gegenüber mark-sweep eine dritte Phase mit der Komplexität $O(|live|)$ durchlaufen. Setzt man voraus, daß die Mark-Phasen der beiden Algorithmen, sowie Compact und Sweep jeweils die gleiche Zeit beanspruchen, so ist also „Mark-Compact" um diesen zusätzliche Durchlauf langsamer als „Mark-Sweep". Dieser Durchlauf kann vermieden werden, wenn nicht direkte Referenzen sondern *handles* verwendet werden. Diese würden den Programmablauf aber zugunsten der Garbage Collection verlangsamen und ist daher nicht zu empfehlen.

Der „Copying Collection"-Algorithmus

Die bisher beschriebenen Algorithmen arbeiten als „Müllsammler", da sie nach der Mark-Phase den Speicher durchlaufen und den ungenutzten Speicher „einsammeln". Der Algorithmus „Copying Collection" verfolgt dabei den umgekehrten Ansatz: er evakuiert die lebendigen Objekte in einen anderen Datenbereich und gibt den Ausgangsbereich auf einmal frei. Damit ist er in seiner Komplexität lediglich von der Menge *live* abhängig. Die hier dargestellte Variante des Algorithmus geht auf [Cheney, 1970] zurück.

Kopierende Garbage Collection unterteilt den zur Verfügung stehenden Speicher in zwei gleich große Halbräume („semi spaces"): diese heißen „from-space" und „to-space". Dabei wird vom Programm immer nur der Quellbereich verwendet – alle Objekte befinden sich hier und auch neue Objekte

werden hier abgelegt (siehe Abb. 2.6-(i)). Findet eine Garbage Collection statt, so werden die lebendigen Objekte in den freien Zielraum kopiert. Danach werden die Rollen der beiden Bereiche getauscht.

Dabei wird folgendermaßen vorgegangen: Zunächst werden zwei Zeiger `scan` und `free` auf den Anfang von „to-space“ initialisiert. `scan` markiert dabei, bis wohin „to-space“ fertig bearbeitet wurde, während `free` anzeigt, wohin das nächste evakuierte Objekt angelegt werden soll.

Danach werden alle Objekte, die von den Wurzeln direkt referenziert werden, nach `free` kopiert und der Zeiger entsprechend erhöht (Abb. 2.6-(ii)). An der ursprünglichen Adresse der Objekte werden Vorwärtsreferenzen (sogenannte *forwarder*) hinterlassen, die auf die neue Adresse des kopierten Objektes verweisen. Das Originalobjekt wird also durch die Vorwärtsreferenz überschrieben.

Jetzt wird der Zeiger `scan` Objekt für Objekt vorbewegt. Jedes Objekt A, das überstrichen wird, wird auf die enthaltenen Zeiger überprüft. Für einen Zeiger ergeben sich prinzipiell zwei Möglichkeiten:

1. Er zeigt auf ein Objekt B in „from-space“, das noch nicht kopiert wurde. In diesem Fall wird B evakuiert, d.h. es wird nach „to-space“ kopiert und der Freispeicherzeiger `free` wird entsprechend erhöht. An der alten Adresse von B wird eine Vorwärtsreferenz hinterlassen. In A wird die neue Adresse von B eingetragen (Abb. 2.6-(iii) und Abb. 2.6-(iv)).

2. Er zeigt auf eine Vorwärtsreferenz, die ihrerseits auf ein Objekt B in „to-space“ zeigt. In diesem Fall wird lediglich die neue Adresse von B in A eingetragen (Abb. 2.6-(iv) und Abb. 2.6-(v)).

Der Zeiger `scan` wird auf diese Weise solange erhöht, bis der Zeiger `free` erreicht wird und damit alle Zeiger innerhalb nach „to-space“ zeigen (Abb. 2.6-(vi)).

Als letzter Schritt des Algorithmus werden die Rollen der beiden Halbräume getauscht. Die Freispeichergrenze wird auf `free` initialisiert und das Programm fortgesetzt (Abb. 2.6-(vii)).

Der Algorithmus von Cheney hat eine Vielzahl von Vorteilen: Zum einen ist er in seiner Komplexität nur von $|live|$ abhängig, nicht aber von $heapsize$. [Appel, 1987] hat in einem einfachen Rechenbeispiel gezeigt, daß die Kosten der Garbage Collection bei Verwendung von „Copying Collection“ durch die Erhöhung des verwendeten Hauptspeichers beliebig verringert werden können.

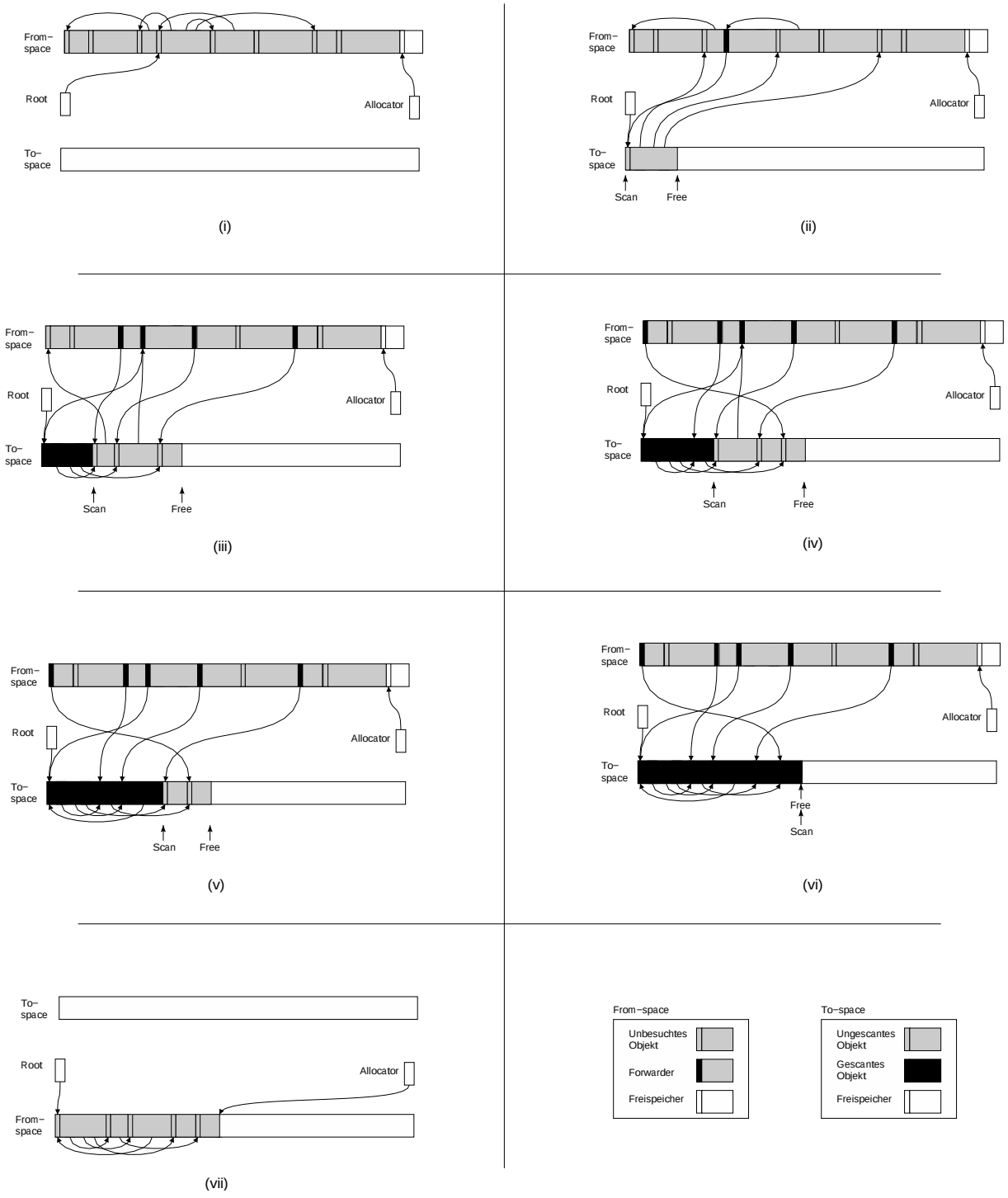

Abbildung 2.6: Ablauf einer „Copying Collection“

Ein weiterer Vorteil besteht darin, daß die Speicherzuteilung sehr einfach implementiert werden kann: wie bei „Mark-Compact“ muß lediglich ein Zeiger erhöht werden, wenn ein neues Objekt anzulegen ist. Positiv ist auch die Tatsache zu bewerten, daß kein Markierungs-Stack benötigt wird, denn der Algorithmus führt eine Breitensuche über den Speicher aus, wobei „to-space“ als Zwischenspeicher verwendet wird.

Diesen Vorteilen steht der große Speicherverbrauch der Methode gegenüber: der doppelte Platz muß zur Verfügung stehen. Außerdem verursacht das wiederholte Kopieren von langlebigen oder großen Objekten einen erheblichen Aufwand.

Nichtsdestotrotz bildet der „Copying Collection“-Algorithmus wegen der genannten Vorteile und seiner Einfachheit die Basis für zahlreiche ausgefeilte Implementierungen von Garbage Collection [Jones and Lins, 1996].

2.2.3 Zusammenfassung

In den vorigen Abschnitten wurden einige Anforderungen, die an die Speicherverwaltung einer modernen virtuellen Maschine gestellt werden dargestellt.

Dabei wurde gezeigt, daß eine dynamische Speicherverwaltung mit Garbage Collection die Softwareentwicklung in einer objektorientierten oder funktionalen Sprache deutlich produktiver gestalten kann. Aus diesem Grunde verfügen fast alle Sprachen dieser Klasse über Garbage Collection.

Der angewendete Algorithmus sollte Speicher aus einem linearen Bereich erlauben, da so die Speicherzuteilung erheblich beschleunigt wird. Auf diesen Punkt wird im Abschnitt 4.1 näher eingegangen.

Um eine Fragmentierung des Speichers zu vermeiden, sollte der Algorithmus eine Speicher-Kompaktierung durchführen. Die Voraussetzung hierfür ist, daß es sich um einen exakten Algorithmus handelt. Auf welche Weise Exaktheit erzielt wird (durch Markierung oder Referenztabellen), ist unter Berücksichtigung der Zielsprache zu entscheiden. Dieses Thema wird in Abschnitt 4.2 für den Fall der Sprache Java näher behandelt.

„Copying Collection“ vereinigt viele positive Eigenschaften, weist aber insofern einen Schwachpunkt auf, daß langlebige Objekte wiederholt kopiert werden. Der „Mark-Sweep“-Algorithmus hingegen bewegt keine Objekte, leidet daher aber unter Speicher-Fragmentierung. Bei beiden Algorithmen kann Garbage Collection störend lange Unterbrechungen des Programmflusses ver-

ursachen. All diese Probleme versucht die generationale Garbage Collection (s. Abschnitt 4.3) zu beheben, indem der Heap nach Alter der Objekte in mehrere Segmente aufgeteilt wird, die dann separat durch unterschiedliche Algorithmen bearbeitet werden können.

Eine neue Klasse von Problemen tut sich auf, wenn die virtuelle Maschine in einer Multithreading- bzw. Multiprozessor-Umgebung abläuft und dabei von mehreren Prozessoren profitieren soll. Beispielsweise muß die nebenläufige Speicherzuteilung möglich sein, da die Allokation sonst einen zentralen Flaschenhals darstellen könnte (s. Abschnitt 4.1).

Weiterhin ist eine Eigenschaft der vorgestellten Algorithmen in einer Multiprozessor-Umgebung infrage zu stellen: sie arbeiten sequentiell, d.h., daß *alle* Threads angehalten werden müssen, bevor die Garbage Collection in einem eigenen Thread ausgeführt werden kann. Auf einem Multiprozessor-System ist in dieser Zeit nur ein Prozessor ausgelastet, während alle anderen ungenutzt bleiben. Nebenläufige Algorithmen zur Garbage Collection (Abschnitt 4.4) versuchen, dieses Manko zu beheben, indem sie nebenläufig mit den Anwenderthreads abläufen.

Die effiziente Implementierung der Speicherverwaltung stellt eine komplexe Aufgabe dar, die nur unter enger Kooperation mit den anderen beiden Kernkomponenten, dem dynamischen Compiler und dem Thread-System, gelöst werden kann. Kapitel 4 stellt den Stand der Technik bei der Implementierung der Speicherverwaltung für virtuelle Maschinen dar.

2.3 Nebenläufigkeit

Nebenläufige Programmverarbeitung ist in vielen unterschiedlichen Inkarnationen in Programmiersprachen vorhanden. In einem nebenläufigen Programm werden mehrere Kontrollflüsse erzeugt, die *potentiell* gleichzeitig ausgeführt werden können. Findet die Ausführung tatsächlich gleichzeitig statt, so spricht man von *Parallelität* [Butenhof, 1997]. Während in deklarativen Programmiersprachen die Nebenläufigkeit einzelner Programmteile oftmals abgeleitet werden kann, z.B. durch die Konfluenzeigenschaft referentiell transparenter funktionaler Sprachen [Peyton Jones, 1989], geschieht dies in imperativen Sprachen explizit durch den Programmierer.

Die erzeugten Kontrollflüsse sind oftmals nur partiell unabhängig voneinander, so daß sie an definierten Punkten untereinander synchronisiert werden müssen. Dies ist z.B. notwendig, wenn eine Aktivität das Ergebnis einer anderen zur Weiterverarbeitung benötigt. Um nebenläufige Programme zu

ermöglichen, muß eine Programmiersprache daher mindestens folgende zwei Konstrukte anbieten: Definition nebenläufiger Aktivitäten und deren Synchronisierung.

Grundsätzlich werden zwei Argumente für den Einsatz von Nebenläufigkeit angeführt [Butenhof, 1997]:

- Viele Probleme lassen sich einfacher modellieren, wenn sie als mehr oder weniger unabhängige Aktivitäten verstanden und durch entsprechende Sprachkonstrukte umgesetzt werden können. Jede Aktivität kann dann isoliert entworfen und implementiert werden[5]. Nebenläufigkeit wird in diesem Kontext also als Abstraktionskonstrukt verstanden, das den Softwareentwicklungsprozess vereinfacht. Ob Parallelverarbeitung stattfindet, ist hierbei von untergeordneter Bedeutung.

- Rechner enthalten mehrere ausführende Entitäten, in der Regel einen oder mehrere Prozessoren (CPU) sowie Ein-Ausgabe-Geräte. Will man diese Ressourcen gleichzeitig nutzen, so muß man ihnen entsprechend mehrere Anweisungsströme zuführen. Diese können dann parallel ausgeführt werden und damit zu einer Reduktion der Ausführungszeit führen. Das Ziel der nebenläufigen Programmdefinitionen ist hier also Parallelität zur schnelleren Programmausführung, indem man das Programm der vorhandenen Hardwarekonfiguration anpasst.

Wir werden sehen, daß für beide Modelle eine mehr oder weniger „kanonische“ Implementierung existiert. Diese beiden „Auslegungen“ wirken in unterschiedlichen Systemebenen und weisen daher unterschiedliche Charakteristika bezüglich ihrer Performanz und Skalierbarkeit auf. Eine der entscheidenden Kennzahlen ist hierbei die Zahl der unterstützten nebenläufigen Aktivitäten. Diese ist im ersten Fall durch die Problemgröße definiert, während im zweiten Fall die Hardwarekonfiguration die Grenze zieht.

Im Folgenden wird gezeigt, welche Abstraktionen zur nebenläufigen Programmierung in Java in Form von Threads, Monitoren und einem Speichermodell zur Verfügung stehen. Die Besonderheiten des Java-Modells und ihre Implikationen für die Sprachimplementierung werden dargelegt. Interaktionspunkte mit anderen Komponenten des Laufzeitsystems werden aufgezeigt. Damit wird der Grundstein gelegt für das Verständnis der Techniken in Kapitel 5, das konkrete Detailfragen und Implementierungen diskutiert.

[5]Es darf allerdings nicht übersehen werden, daß die notwendige Synchronisierung die Komplexität des Entwurfs wiederum erhöht.

```
1  new Thread() {       // anonyme Subklasse von Thread
2    public void run() {
3      do_asynchronous_work();
4    }
5  }.start();           // Ausführung starten
```

Abbildung 2.7: Thread-Beispiel

2.3.1 Threads und Synchronisierung in Java

Threads sind in Java das Mittel zum Ausdruck von Nebenläufigkeit. Ein Thread entspricht einem Exemplar der Systemklasse `java.lang.Thread` oder einer Subklasse. Für jeden Thread führt die virtuelle Maschine nebenläufig die Methode `run` dieser Klasse aus. Ist die Ausführung der Methode beendet, terminiert der Thread. Threads werden in Java also durch die Sprachmittel Objekterzeugung und Methodenaufruf dargestellt. Nebenläufigkeit durch Threads findet sich in vielen Programmiersprachen [Nelson, 1991, Stoutamire and Omohundro, 1996] oder Bibliotheken [POSIX, 1996, Rogue Wave, 2000] und ist somit ein verbreitetes Programmiermodell.

Abbildung 2.7 zeigt ein Codefragment, das ein neues Thread-Objekt erzeugt und startet.

Zur Synchronisierung lehnt sich die Java-Spezifikation an das Modell des „Monitors“ nach [Hoare, 1974] an. Ein Monitor erfüllt zwei Aufgaben: er erlaubt gegenseitigen Ausschluß und damit geschützten Zugang zu einer Ressource. Weiterhin fungiert ein Monitor als Ereignisvariable[6], das heißt, durch ihn können Ereignisse zwischen Threads ausgetauscht werden. Während diese Ereignisse selbst keine Information tragen, findet Informationsfluß über den globalen Objektspeicher statt. Dieses Modell belegt die konzeptuelle Verwandtschaft des Java-Modells zu Multiprozessor-Architekturen mit gemeinsamen Speicher und steht im Kontrast zu Architekturen wie PVM [Sunderam, 1989] und MPI [Brunck, 1994], die auf der Technik des *message passing* basieren.

Gegenseitiger Ausschluß wird in Java durch das Schlüsselwort `synchronized` in der Methodensignatur oder durch einen „synchronized block“ der Form `synchronized(expr) { statements }` realisiert. In beiden Fällen wird der

[6]Nach Hoares Definition kann ein Monitor mehrere Ereignisvariablen enthalten, in Java genau eine.

Monitor des betroffenen Objekts erworben, das heißt entweder des Empfängerobjekts im Falle einer „synchronized method“ oder des Ergebnisses des Synchronisierungsausdrucks eines „synchronized block“.

Aus obiger Definition folgt, daß Monitore von einem Java-Programm nur in Blockstruktur erworben werden; je zwei Monitore können nur in umgekehrter Erwerbungsreihenfolge wieder freigegeben werden. Auf Bytecode-Ebene ist dies allerdings nicht der Fall: Monitore werden durch Bytecode-Instruktionen `monitorenter` und `monitorexit` erworben und freigegeben; dabei ist keine Schachtelungsreihenfolge vorgegeben. Compiler anderer Programmiersprachen können daher legalen Bytecode erzeugen, der andere Freigabereihenfolgen benutzt.

Java Monitore sind *rekursiv*; sie können von einem Thread beliebig oft erworben werden. Monitore sind *blockierend*; ein synchronisierender Thread blockiert so lange, bis der Monitor erworben werden konnte. Es gibt keine Möglichkeit, die Verfügbarkeit eines Monitors zu erfragen.

Um die Signalvariable eines Monitors zu nutzen, muß dieser bereits erworben sein. Durch Warten auf der Signalvariablen wird der Monitor freigegeben und der Thread suspendiert. Nun haben andere Threads die Chance, den Monitor zu erwerben, den Objektzustand zu verändern und dies über die Signalvariable zu signalisieren. Ein oder mehrere wartende Threads werden daraufhin fortgesetzt und versuchen, den Monitor wieder zu erwerben, sobald dieser vom signalisierenden Thread freigegeben wurde[7]. Gelingt dies, kehrt der Thread aus dem Warteaufruf zurück und kann fortfahren.

Warten und Signalisieren wird durch Aufruf der Methoden `wait` und `notify[All]` der Klasse `java.lang.Object` realisiert.

Das Beispielprogramm in Abbildung 2.8 zeigt den Umgang mit Synchronisierungsoperationen in Java. Die Thread-Zustände sind am Rand markiert und entsprechen denen aus Abbildung 2.9. Dargestellt ist die Definition eines „worker threads“. Dieser entfernt wiederholt aus einer Menge von Arbeitsaufträgen einen Job und führt diesen aus. Ist kein Job vorhanden, so wartet der Thread auf der Ereignisvariable des Objekts `incoming`. Ein Auftraggeber signalisiert diese entsprechend.

Abbildung 2.9 erläutert die Zustände $sync_{t,m}$, die ein Thread t gegenüber einem Monitor m einnehmen kann. Die möglichen Übergänge sind:

u → **e** Erwerb eines freien Monitors

[7] Auch Signalisierung kann nur verwendet werden, wenn der Monitor erworben ist.

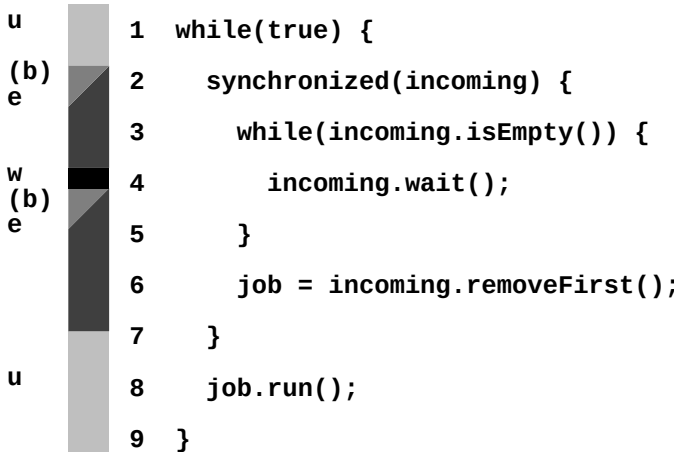

Abbildung 2.8: Monitor-Beispiel

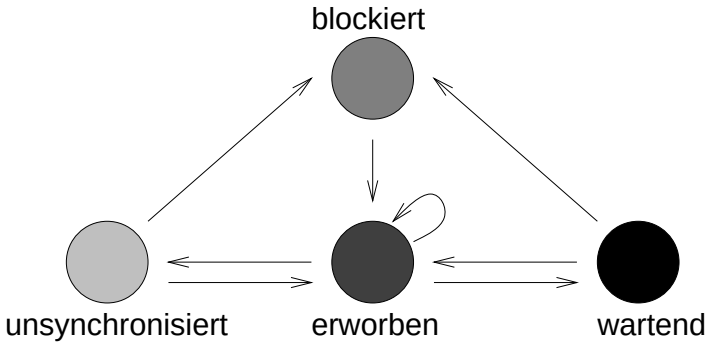

Abbildung 2.9: Zustandsdiagramm eines Threads und eines Monitors

u → **b** versuchter Erwerb eines bereits erworbenen Monitors

b → **e** Erwerb eines Monitors nach dessen Freigabe

e → **e** rekursiver Erwerb oder Freigabe eines bereits erworbenen Monitors

e → **w** Warten auf Signalvariable

w → **e** Wiedererwerb nach Signalisierung

w → **b** versuchter Wiedererwerb nach Signalisierung

Man beachte, daß diese Zustände nur in Bezug auf einen Monitor gelten. Ein Thread kann gleichzeitig beliebig viele verschiedene Monitore erworben haben ($m1 \neq m2 \wedge sync_{t,m1} = sync_{t,m2} = e$), blockieren oder warten kann er jedoch nur auf einem ($\neg\exists m1 \neq m2 : sync_{t,m1} = sync_{t,m2} = b$). Das atomare Erwerben mehrerer Monitore ist nicht vorgesehen.

Versuchen zwei Threads zur gleichen Zeit, den Monitor zu erwerben, zu muß ein Thread blockieren — ein solcher Konflikt wird als *contention* bezeichnet.

2.3.2 Das Speichermodell

Die nebenläufige Ausführung von Threads hat eine weitere maßgebliche Konsequenz: die im sequentiellen Fall weitgehend transparenten Lese- und Schreiboperationen auf dem globalen Speicher gewinnen an Komplexität, und ihre Semantik muß durch ein entsprechendes Modell, das *Speichermodell* definiert werden. Die folgenden Abschnitte erläutern den Begriff des Speichermodells im Allgemeinen und in Anwendung auf Java.

Zu den Basisoperationen der Von-Neumann-Architektur zählen das Lesen und Verändern des globalen Heap, also das Lesen und Schreiben von Speicherzellen. Das Verhalten dieser Operationen auf Uniprozessorarchitekturen ist einfaches und intuitiv: alle Speicheroperationen sind global — in Programmreihenfolge — geordnet und das Lesen eines Feldes liefert den Wert, der zuletzt geschrieben wurde.

Nach dieser Betrachtung sind Operationen auf disjunkten Speicherzellen unabhängig voneinander, d.h. es ist beispielsweise für zwei Schreiboperationen nicht definiert, welche zuerst ausgeführt wurde. Dadurch eröffnet sich eine Zahl von Optimierungsmöglichkeiten für den Prozessor, um insbesondere die hohe Latenz des Speicherzugriffs zu umgehen. Beispielsweise kann der Prozessor Schreiboperationen so umordnen, so daß sie die gleiche Cache-Zeile betreffen und en bloque in den Speicher geschrieben werden können.

Formalisierung

Moderne Prozessoren werden abstrakt in Form von *processor architectures* spezifiziert, um im folgenden als Prozessorfamilie mehrere Entwicklungszyklen zu durchlaufen. Hierzu zählen die SPARC-, Alpha- und PowerPC-Architekturen [Weaver and Germond, 1994, Sites, 1992, May et al., 1994]. Um sicherzustellen, daß die Prozessoren auch in Multiprozessor-Umgebungen ein vorhersagbares Verhalten aufweisen, muß das Speichermodell formuliert werden. Insbesondere durch die hohe Komplexität nebenläufiger Ausführung bietet sich eine formale Beschreibung an.

Beispielhaft wird hier grob eine Formalisierung mit Hilfe schematischer Auftragssysteme [Jessen and Valk, 1987] skizziert: ein auszuführendes Programm bzw. ein Fragment, das dem Prozessor bereits vorliegt, entspricht in diesem Modell einer Ausführungsfolge w eines schematischen Auftragssystems mit Lese- und Schreibaufträgen auf Registern und dem globalen Speicher[8]. Durch das Speichermodell können Präzedenzen auf den Aufträgen definiert sein, die über die Wahrung des Datenflusses, d.h. die Werteübertragungsrelation hinausgehen.

Das Prozessorverhalten läßt sich nun mit Hilfe des Modells spezifizieren: eine Ausführung w' durch den Prozessor wird dann als korrekt angesehen, wenn w' äquivalent zu w ist, d.h. für beliebige Eingaben das selbe Ergebnis liefert.

Gebräuchliche Speichermodelle für Multiprozessoren fordern keine zusätzlichen Präzedenzen auf den Aufträgen, d.h., unter Wahrung des Daten- und Kontrollflusses sind beliebige Umordnungen von Speicheroperationen möglich. Das hat zur Folge, daß die „von außen“ sichtbare Zustandsfolge der Speichervariablen von w' sich von der von w unterscheiden kann. Die Auswirkungen werden anhand eines Beispiels erläutert. Gegeben sei das folgende Codefragment:

```
; initially: x = 0, ok = false
store x, <some value>
store ok, true
```

Die ensprechende Ausführungsfolge lautet dann

$$w = l_0[], s_0[x, ok], l_1[], s_1[x], l_2[], s_2[ok], l_3[x, ok], s_3[]$$

($l0, s0, l3, s3$ stellen den *Initialisierungs-* bzw. *Ausgabeauftrag* dar). Das zugehörige Auftragssystem ist in Abbildung 2.10 abgebildet.

[8]Diese Definition ist vereinfachend: bei indirekten Speicherzugriffen stehen Operanden oder Ziel einer Operation erst zur Laufzeit fest.

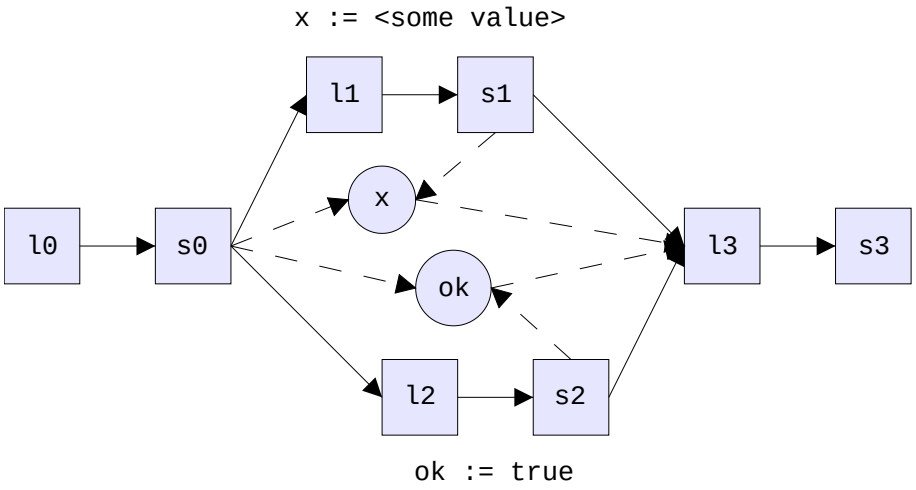

Abbildung 2.10: Auftragssystem

Die Ausführungsfolge

$$w' = l_0[], s_0[x, ok], l_2[], s_2[ok], l_1[], s_1[x], l_3[x, ok], s_3[]$$

ist äquivalent zu w. Aus diesem Grunde können beide Zuordnungen legal vertauscht werden. Es wäre somit für ein anderes Programm möglich, den inkonsistenten Zustand $x = 0, ok = true$ im Speicher zu beobachten.

Dieses Modell ist unvollständig und verdeutlicht lediglich den Charakter und Eigenschaften eines Speichermodells. Eine realistische Spezifikation findet sich in [Broy, 1995]; hier wird mit anderen Mitteln das Speichermodell der Alpha-Architektur formalisiert.

Sequential Consistency

Die Komplexität nebenläufiger Berechnungen auf einem gemeinsamen Speicher wurde früh erkannt. Lamport formulierte daher das Modell der „Sequential Consistency“: die beobachteten Veränderungen des Speichers müssen einer global geordneten Ausführung der Speicheroperationen entsprechen, wobei diese pro Prozessor in Programmordnung erfolgen [Lamport, 1979]. Viele der klassischen Synchronisierungsalgorithmen wie z.B. „Dekker's Algorithm“ beruhen auf „Sequential Consistency“.

Unglücklicherweise verhindert „Sequential Consistency“ viele der oben angesprochenen Optimierungen. Da nicht bekannt ist, welche Operationen von nebenläufigen Prozessen auf dem Speicher ausgeführt werden, können Programmoperationen im Gegensatz zum sequentiellen Fall nicht transparent

umgeordnet werden. Bezogen auf das formale Modell bedeutet „Sequential Consistency“ eine Präzedenz zwischen je zwei im Programm aufeinanderfolgenden Aufträgen. Der erwartete Effizienzgewinn durch den Einsatz mehrerer Prozessoren wird somit durch den Verlust an Optimierungen geschmälert oder gar zunichte gemacht. Aus diesem Grunde implementieren moderne Multiprozessorarchitekturen weitaus lockerere Speichermodelle ([Adve and Gharachorloo, 1995] geben eine sehr gute Einführung). Diese Speichermodelle garantieren nicht, wann genau ein Prozessor Auswirkungen von Speicheroperationen anderer Prozessoren wahrnehmen kann[9].

Um das Verhalten des Prozessors zu beeinflussen und die Sichtbarkeit von Schreiboperationen an bestimmten Programmpunkten zu erzwingen, bieten genannte Architekturen explizite Markierungsinstruktionen (sogenannte *memory barriers*). Diese geben sozusagen Grenzmarkierungen vor, innerhalb derer optimiert werden darf, und ermöglichen somit den konsistenten Informationstransfer über den gemeinsamen Speicher.

Im Modell entspricht eine *memory barrier* einem Auftrag, der von allen Variablen liest und auf alle schreibt. Mit Hilfe der Datenflußrelation stellt eine *memory barrier* also transitiv Präzedenzen zwischen den vorhergehenden und den folgenden Speicheroperationen her; im obigen Beispiel sich somit der „unerwünschte“ Zustand $x = 0, ok = true$ im Speicher verhindern[10].

Speichermodell und Synchronisierung

Das nebenläufige Lesen und Schreiben gemeinsamer Daten muß — unabhängig vom Speichermodell — synchronisiert werden. Es liegt daher nahe, das Speichermodell an Synchronisierungsoperationen zu knüpfen. Das heißt, nur wenn alle Zugriffe auf gemeinsame Objekte durch Synchronisierung geschützt werden, ist auch garantiert, daß die Threads den gleichen Speicherzustand „sehen“.

Diese Überlegungen liegen dem POSIX Thread-Standard [POSIX, 1996] zugrunde. Um Speicherkonsistenz zu garantieren, muß der Schreiber eine Sperre aufgeben und der Leser diese Sperre erwerben. Beide Operationen haben neben der Synchronisation somit den Charakter einer portablen *memory*

[9] Nota bene: diese Problematik liegt jenseits der von Cache-Kohärenz. Es geht darum, daß Speicheroperationen unter Umständen gar nicht in Programmordnung ausgeführt werden.

[10] Ein „Beobachter“ auf einem anderen Prozessor muß zumeist auch auf Leserseite eine solche Barrier-Instruktion verwenden, da ansonsten eine Leseoperation „zu früh“ ausgeführt werden könnte.

barrier. Somit ist garantiert, daß erstens alle Schreiboperationen tatsächlich stattgefunden haben, wenn die Sperre freigegeben wird, und zweitens, daß die Leseoperationen erst vollzogen werden, nachdem die Sperre erworben werden konnte.

Das Speichermodell einer Hochsprache

Das Speichermodell ist keinesfalls nur ein hardwarenahes Artefakt der Systemarchitektur, von dem eine Hochsprache abstrahieren kann. Um Objektmanipulationen möglichst einfach und effizient auf Speicheroperationen abzubilden, muß ein weniger intuitives Speichermodell auch in der Hochsprache in Kauf genommen werden. Dieses Modell ist in der Sprachdefinition zu verankern. Durch das Speichermodell eröffnen sich für den Compiler zusätzliche Optimierungsmöglichkeiten. Dieser kann dem Modell genügende Umordnungen vornehmen und infolge dessen z.B. redundante Speicherzugriffe eliminieren.

Die Speicheroperationen objektorientierter Sprachen sind Sondierung und Veränderung von Objektzuständen. Es gilt also zu definieren, welche Transitionen des Objektzustands von anderen Threads beobachtet werden können und wie diese zu beeinflussen sind.

Die Java Sprachdefinition [Gosling et al., 1996] formuliert daher in Textform ein Speichermodell für Java in den Begriffen *Objektfeld*, *Array*, *Thread* und *Monitor*, welches von der virtuellen Maschine entsprechend implementiert werden muß. In diesem Modell werden eine Reihe von Operationen auf dem globalen Speicher und einem thread-lokalen Arbeitsspeicher sowie die notwendigen Ordnungskriterien auf ihnen definiert. Das Java Speichermodell berücksichtigt also die Speicherhierarchie und sieht die Umordnung von Speicheroperationen vor. Trotzdem hat es sich in seiner jetzigen Form als unvollständig, schwer verständlich und nicht effizient implementierbar erwiesen. Insbesondere widerspricht es einigen gängigen Compileroptimierungen — und wird in diesen Punkten von den meisten optimierenden dynamischen Compilern verletzt. Als Reaktion wird das Speichermodell daher zur Zeit revidiert. Beispiele und Erläuterungen zu dieser Problematik finden sich in [Pugh, 2000b].

Synchronisierungsoperationen stehen in dem Ruf, ineffizient zu sein. Als Folge existieren in der Literatur eine Zahl von Idiomen, die vor allem auf die Vermeidung solcher Synchronisierungsoperationen hinzielen, z.B. für „konstante“ Objekte oder Einmal-Initialisierungen. Da auf diese Weise die impliziten *memory barriers* entfallen, sind weitreichende Optimierungen auf

diesen Konstrukten möglich. Es zeigt sich, daß die meisten dieser Idiome in Unkenntnis oder unter Mißachtung des Speichermodells formuliert sind und daher auf aggressiven Implementierungen nicht funktionieren werden. [Pugh, 2000a] analysiert "Double-Checked Locking", ein sehr weit verbreitetes und ebenfalls nicht sicheres Idiom.

Zusammenfassung

Gängige Compiler-Optimierungen sowie moderne Hardware-Plattformen weisen komplexe Modelle der Speichersichtbarkeit nebenläufiger Programme auf. Der Portabilitätsgedanke erfordert die Definition eines plattformunabhängigen Speichermodells für die Programmiersprache Java. Das aktuelle Speichermodell weist eklatante Schwächen auf, die die Erzeugung effizienten Maschinencodes durch den optimierenden Compiler erheblich behindern. Daher ist eine Revision des Speichermodells erforderlich.

Das Speichermodell erfordert „korrekte" Synchronisierung der Programme. Es ist daher weiterhin wichtig, die Synchronisierungsoperationen effizient zu implementieren.

2.3.3 Einsatz von Nebenläufigkeit in Java

Der Einsatzkontext von Java-Software hat sich seit Einführung der Plattform radikal verschoben: während zu Beginn Java-Programme als sichere, portable, ausführbare Inhalte im Kontext von Webseiten, sogenannte „Applets", positioniert waren, ist der Erfolg dieser Plattform auf ein anderes Umfeld zurückzuführen. Allgemein vorteilhafte Eigenschaften der Sprache wie starke Typisierung, automatische Speicherverwaltung, dynamisches Laden von Programmfragmenten, hohe Portabilität und andere haben Java zur Plattform der Wahl für Serversoftware gemacht.

Serverprogramme stellen per se nebenläufige Software dar, denn sie müssen mehrere Client-Programme gleichzeitig bedienen können. Für Server der oberen Leistungsklasse sind daher mehrere hundert nebenläufige Threads nicht ungewöhnlich. Gleichzeitig ist die effiziente Nutzung leistungsfähiger Hardware von eminenter Wichtigkeit: die verwendete Hardwareplattform stellen immer häufiger Multiprozessorsysteme mit bis zu 64 Prozessoren und gemeinsamen Speicher dar [Lenoski and Weber, 1995].

Obwohl die Java-Synchronisierungskonstrukte vom Mechanismus her keine Neuerung darstellen, so erfordern zwei Umstände, daß der Implementierung

dieser Konstrukte besondere Aufmerksamkeit gewidmet werden muß:

- Während gängige Thread-Schnittstellen Monitore als separate Objekte modellieren, die an den nötigen Stellen explizit zu erzeugen und einzusetzen sind, kann in Java *jedes* Objekt zur Synchronisierung verwendet werden.
- Große Teile der Java-Bibliotheken sind im eingeschränkten Sinne bereits thread-sicher implementiert. Das bedeutet konkret, daß Methoden, deren nebenläufige Ausführung ein Objekt in einen inkonsistenten Zustand bringen könnte, durch die Verwendung des Schlüsselwortes `synchronized` exklusiven Zugriff auf das Objekt erhalten.[11]

Hieraus ergeben sich folgende Konsequenzen: ein Java-Programm erzeugt eine enorme Anzahl von Monitoren, von denen — mit Ausnahme einiger Bibliotheksobjekte — nur ein kleiner Anteil tatsächlich zur Synchronisierung verwendet wird. [Agesen et al., 1999] und [Onodera and Kawachiya, 1999] nennen übereinstimmend eine Rate von 8–12% in einer Reihe von Benchmarks. [Dieckmann and Hölzle, 1998] messen eine durchschnittliche Objektgröße von 20 Bytes für eine Reihe von Java-Anwendungen. Würde für jedes Objekt nur ein Wort zur Synchronisierung reserviert, ergäbe sich bereits ein Anstieg der Heapgröße um 20% für zumeist ungenutzte Monitore. Es folgt, daß der benötigte Platz für eine Monitorimplementierung sinnvollerweise nur für die aktuell synchronisierten Objekte bereitgestellt werden darf.

Weiterhin wird die Mehrzahl der synchronisierten Objekte — vielfach aus Basisbibliotheken — tatsächlich nur von einem Thread benutzt. [Onodera and Kawachiya, 1999] ermitteln, daß weniger als 2% aller synchronisierten Objekte tatsächlich einen Konflikt auflösen. Aufgrund „großzügiger“ Synchronisierung werden zudem Monitore häufig rekursiv erworben. Die Optimierung solcher „präventiven“ Synchronisierungsoperationen hinsichtlich ihrer Geschwindigkeit verspricht daher einen großen Effizienzgewinn.

2.3.4 Interaktion mit anderen VM-Komponenten

Die nebenläufige Ausführung mehrerer Threads verkompliziert die Implementierung anderer Komponenten der virtuellen Maschine. Insbesondere die Speicherverwaltung und der Compiler sind hiervon betroffen.

[11] Dies ist keinesfalls hinreichend für Thread-Sicherheit: die Anwendungssemantik erfordert oftmals aufrufübergreifende Synchronisierungsklammern.

Die meisten Algorithmen für Garbage Collection erfordern, daß zur konsistenten Ermittlung der Wurzelreferenzen alle Threads an definierten Punkten, sogenannten „GC Points“, angehalten werden. Mit steigender Zahl von Threads wird es schwieriger, diesen Operation effizient zu implementieren. Mehrere diesbezügliche Techniken werden in Abschnitt 5.3 erläutert.

Bei der hohen Allokationsrate objektorientierter Sprachen ist es weiterhin nicht gangbar, für jede Allokationsoperation den globalen Heap durch komplexe Synchronisierungsmaßnahmen zu schützen. Die verwendeten Ausschlußverfahren dürfen nur einen sehr geringen Overhead pro Operation aufweisen.

Zusätzliche Komplexität für den Compiler besteht darin, daß der generierte Maschinencode ähnlich wie der Freispeicher nicht durch klassische Sperren geschützt werden kann. In einigen Fällen muß Code durch den Compiler nachträglich geändernt werden — beispielsweise, wenn Annahmen über den Empfängertyp falsch sind, oder sich durch Laden neuen Codes die Klassenhierarchie geändert hat. Da andere Threads diesen Code nebenläufig ausführen, also davon lesen, muß eine solche Änderung atomar vollzogen werden. Typischerweise wird Code derart erzeugt, daß nur ein einzelnes Instruktionswort überschrieben werden muß.

2.3.5 Zusammenfassung

Nebenläufigkeit spielt eine große Rolle in der Komplexität von Java-Implementierungen im Vergleich zu anderen Programmiersprachen. Obwohl Nebenläufigkeit kein neues Konzept ist, erheben moderne Java-Implementierungen als erste den Anspruch, gleichermaßen eine große Zahl von Threads zu unterstützen und vorhandene Ressourcen von Mehrprozessormaschinen auszunutzen.

Eine große Anzahl aktiver Thread erfordert effiziente Kontextwechsel und sparsamen Umgang mit Systemressourcen pro Thread. Auf der anderen Seite erfordern Multiprozessoren die Kommunikation mit dem Betriebssystem und die Belegung von Systemressourcen und werfen somit Skalierungsprobleme auf.

Synchronisierung in Java ist ubiquitär. Jedes Objekt kann potentiell zur Synchronisierung verwendet werden; nur bei einem Bruchteil findet dies tatsächlich statt. Es müssen daher platzeffiziente Kodierungsschemata für ungenutzte Monitore gefunden werden. Insbesondere durch Bibliotheksabstraktionen weisen Java-Programme zudem eine außerordentlich hohe Zahl

von Synchronisierungsoperationen auf, obwohl keine Konkurrenz besteht. Gerade diese Präventivoperationen müssen hocheffizient implementiert oder durch Programmanalyse gar eliminiert werden.

Die thread-sichere und trotzdem effiziente Implementierung anderer VM-Komponenten stellt weitere Herausforderungen an den Sprachimplementierer.

Kapitel 5 ist Implementierungstechniken zur Realisierung von Nebenläufigkeit in virtuellen Maschinen gewidmet.

Kapitel 3

Techniken zur Programmausführung

```
// He's dead, Jim.
_cfg = (PhaseCFG*)0xdeadbeef;
```

Sourcecode of the Hotspot VM's optimizing Compiler

Zur effizienten Ausführung von Java-Code wird ein optimierender Compiler benötigt – andererseits muß die Möglichkeit bestehen, zur Laufzeit Klassen hinzuzufügen. Dies führt zum Modell des dynamischen Compilers, der die ausgeführten Klassen erst zur Laufzeit übersetzt. Dieses Kapitel beschäftigt sich mit den Implementierungstechniken dynamischer Compiler. Dabei wird der Schwerpunkt auf jene Punkte gelegt, die einen dynamischen Compiler von einem statischen unterscheiden.

So ist es bei der Verwendung eines dynamischen Compilers nicht sinnvoll, alle Klassen gleichermaßen zu optimieren – hingegen muß eine Kosten-Nutzen-Rechnung ausgeführt werden, um zu entscheiden, welche Methode zur Verbesserung der Performanz zu kompilieren ist. Abschnitt 3.1 befaßt sich mit dieser Thematik.

Als Grundlage für die o.g. Kosten-Nutzen-Rechnung benötigt der Compiler Informationen über das Laufzeitverhalten des ausgeführten Programms. Diese Erhebung von Laufzeitprofilen wird in 3.2 geschildert.

Abschnitt 3.3 beschäftigt sich mit dem Problem der Deoptimierung, das durch das dynamische Laden von Klassen entsteht, da sich beim Hinzufügen einer Klasse die Voraussetzungen für die spezifische Optimierung einer Methode ändern könnnen.

Zuletzt werden in 3.4 globale Analyseverfahren beschrieben, die Optimierungen ausführen, die Methodengrenzen überschreiten. In 3.5 werden die Ergebnisse zusammengefaßt.

3.1 Analyse der Kosten-Nutzen-Rechnung eines dynamischen Compilers

Beim Einsatz von Optimierungstechniken muß der Nutzen (z.B. geringere Programmlaufzeit) gegen die eingesetzten Mittel aufgerechnet werden. Ein Compiler muß sich selbst „tragen“ [Franz, 1994]. Bei einem statischen Compiler wird implizit davon ausgegangen, daß das übersetzte Programm oft genug ausgeführt wird, um die eingesetzte Übersetzungszeit zu kompensieren. Allenfalls während der Programmentwicklung wird die Übersetzungszeit als störender Faktor angesehen und durch weniger zeitaufwendige Optimierungen oder den Einsatz von Interpretern soweit möglich minimiert.

Ein dynamischer Compiler arbeitet hingegen „in“ dem laufenden Programm und hat damit direkten Anteil an dessen Laufzeit. Der Einsatz aufwendiger Optimierungstechniken ist für einmalig ausgeführten Code mit allergrößter Wahrscheinlichkeit nicht nutzbringend, sondern nimmt im Gegenteil insgesamt mehr Zeit in Anspruch als er einspart. Methoden mit hohem Laufzeitanteil, sogenannte *hot spots*, versprechen die größten Vorteile.

Bei der Beurteilung, welche Programmteile lohnenswerte Ziele eines optimierenden Übersetzungsvorgangs sind, sind sowohl Charakteristika des Compilers als auch des zu übersetzenden Programms maßgeblich. In diesem Abschnitt wird ein Modell der anzustellenden Kosten-Nutzen-Rechnung entwickelt, anhand dessen anschließend einige Entwurfsentscheidungen für die Compiler-Komponente diskutieren werden.

3.1.1 Das Modell

Das folgende Modell betrachtet Laufzeit als maßgebliche Größe für Optimierungsentscheidungen. Es lehnt sich an das in [Arnold et al., 2000b] dargestellte Modell an, und ergänzt einige zusätzliche Betrachtungen.

Es seien zwei vereinfachende Annahmen gemacht: erstens ist die Einheit der Übersetzung eine Methode m ohne Betrachtung von *inlining*, zweitens wird sequentielle Verarbeitung angenommen, d.h. die Möglichkeit paralleler Übersetzung wird nicht in Betracht gezogen. Wegen dieser und folgender Vereinfachungen eignet sich das Modell nur für qualitative Aussagen.

Es werden drei Komponenten betrachtet:

- Erwartete Restlaufzeit bei Interpretation t_{int}: Wieviel Zeit wird das Programm erwartungsgemäß durch Interpretation der betrachteten Methode in Anspruch nehmen?
- Erwartete Restlaufzeit mit optimierender Übersetzung t_{opt}: Wieviel Zeit wird das Programm erwartungsgemäß in Ausführung der übersetzten Methode verbringen?
- Übersetzungszeit t_{comp}: Wie lange dauert die Übersetzung der Methode?

Die Übersetzung der Methode ist nur lohnenswert, wenn Übersetzungszeit plus erwartete Restlaufzeit der optimierten Methode kleiner sind als der Interpretationsaufwand:

$$t_{comp} + t_{opt} < t_{int} \tag{3.1}$$

Die drei Komponenten lassen sich weiter aufschlüsseln:

- Die Übersetzungszeit hängt vor allem von der Komplexität der verwendeten Optimierungsalgorithmen ab, und damit von der zu übersetzenden Methode. Relevante Größen sind hier die reine Methodengröße (Anzahl der Instruktionen), die Schleifen- und Sprungstruktur und damit die Struktur des Kontrollflußgraphen sowie die Zahl der verwendeten Variablen (z.B. für die Registerallokation). Da die beiden letzen Größen erst während eines Übersetzungsvorgangs offenbar werden, verwenden [Arnold et al., 2000b] ein lineares Modell der Methodengröße[1], welches durch entsprechende Messungen erarbeitet wurde. Es folgt:

  $$t_{comp} = c * |m| \tag{3.2}$$

  c drückt die Zeitkomplexität der Optimierungsalgorithmen aus.

[1] Dies ist setzt natürlich entsprechende Compiler-Algorithmen voraus.

- Die Laufzeit der optimierten Methode variiert ebenfalls mit der Struktur des Programmcodes. Es kann also nur eine ungenaue Vorhersage über die Effektivität der Optimierungen gemacht werden. Entsprechend der Vorlage wird hier ebenfalls von einem konstanten Faktor ausgegangen:

$$t_{opt} = \frac{t_{int}}{s} \tag{3.3}$$

  $s > 1$ drückt hier die Beschleunigung (*speedup*) der Methode aus und ist ebenfalls aus Erfahrungswerten abzuleiten.

- Man geht bei einem laufenden, nicht-interaktiven System von einer relativ stabilen Verteilung der Laufzeit auf einzelne Programmteile aus, so daß die Laufzeit einer Methode ein bestimmter Bruchteil der Gesamtrestlaufzeit ist.

$$t_{int} = p * t_{rest} \tag{3.4}$$

  Der Anteil p der betrachteten Methode an der Gesamtlaufzeit kann aufgrund der Stabilitätsannahme aus dem Laufzeitprofil des laufenden Programms gewonnen werden.

Setzt man obige Gleichungen ineinander ein, so ergibt sich, daß eine Übersetzung lohnenswert ist, wenn gilt:

$$c * |m| + \frac{p * t_{rest}}{s} < p * t_{rest} \tag{3.5}$$

resp.

$$\frac{|m|}{p * t_{rest}} < \frac{1}{c} * \left(1 - \frac{1}{s}\right) \tag{3.6}$$

Die einfließenden Größen noch einmal zusammengefasst:

$\|m\|$	Methodengröße
p	Anteil der interpretierten Methode an der Programmlaufzeit
s	Maß für die Effektivität des Compilers
c	Maß für die Zeitkomplexität des Compilers
t_{rest}	Erwartete Restlaufzeit des Programms

Das vorgestellte Modell kann leicht auf ein System mit mehreren, unterschiedlich stark optimierenden Compilern erweitert werden.

3.1.2 Interpretieren oder Übersetzen?

Das vorgestellte Modell kann als Entscheidungsalgorithmus in der virtuellen Maschine verwandt werden. Hierzu ist es nötig, die beteiligten Größen festzustellen. Wie dargelegt, sind s und c Charakteristika des Compileralgorithmus und werden typischerweise *offline* kalibriert. Die Methodengröße ist trivialerweise bekannt.

Zur Bestimmung von p muß ein Programmprofil erstellt werden. Die Profildaten können dazu regelmäßig zeitgesteuert, im Methodenprolog oder bei einem Kontextwechsel [Arnold et al., 2000b] erhoben werden. Dieser zusätzliche Verwaltungsaufwand ist so gering wie möglich zu halten, da er die Laufzeit negativ beeinflußt. Zu verwendeten Verfahren gibt der folgende Abschnitt 3.2 Auskunft.

Die Restlaufzeit t_{rest} kann nur heuristisch bestimmt werden. [Arnold et al., 2000b] schlagen vor anzunehmen, das Programm werde noch einmal so lange laufen wie bisher. Auf diese Weise können Negativentscheidungen später revidiert werden. Die HotSpot-VM in der Version 2.0 von Sun Microsystems bietet mit einer Client- und einer Server-Version die Option, zwischen kurz- und langlebigen Programmen zu entscheiden. Hiermit lassen sich die Übersetzungsentscheidungen der virtuellen Maschine beeinflussen.

In Formel 3.6 sind alle programmabhängigen Größen auf der linken Seite zusammengefasst. Die Übersetzungsentscheidung läßt sich daher qualitativ allein durch Betrachtung der linken Seite nachvollziehen. Zunächst wird deutlich, daß mit längerer Programmlaufzeit mehr Methoden übersetzt werden können, da dem einmaligen Übersetzungsvorgang eine vielfache Laufzeitersparnis gegenübersteht. Weiterhin weisen häufig ausgeführte Methoden sowie Methoden mit vielfach wiederholten Schleifen ein hohes Verhältnis von p zu $|m|$ auf und sind somit Übersetzungskandidaten.

3.1.3 Eigenschaften des Compilers

Während im vorigen Abschnitt die Übersetzungsentscheidung Gegenstand der Betrachtung war, behandelt dieser Abschnitt das Problem der Wahl des Optimierungsalgorithmus. Grundsätzlich kann festgestellt werden, daß die Parameter s und c positiv miteinander gekoppelt sind, d.h. effektivere Optimierungen lassen sich im Allgemeinen nur durch Einsatz von mehr Übersetzungszeit realisieren. Die Natur dieser Kopplung kann daher für unterschiedliche Algorithmen differierende Übersetzungsentscheidungen hervorrufen. Bei-

spielsweise kann auch ein erheblicher Mehraufwand in der Optimierung durch einen hohen Laufzeitanteil von „Kernmethoden“ gerechtfertigt sein. Umgekehrt wiegt ein marginaler Geschwindigkeitsvorteil unter Umständen nicht die erforderlichen zeitaufwendigen Programmanalysen auf.

Es gibt folglich keine optimale Wahl der angewandten Compiler-Algorithmen; statt dessen ist der geeignete Compiler vom Programmprofil abhängig. Dies hat zu zwei interessanten Entwicklungen geführt:

- Compiler-Algorithmen mit möglichst günstigem Kosten-Nutzen-Verhältnis kommen verstärkt zum Einsatz. Beispiele sind Algorithmen zur Registerallokation, die in linearer Zeit arbeiten [Poletto and Sarkar, 1999]. Dabei lassen sich mit signifikant schnelleren Algorithmen qualitativ vergleichbare Optimierungsergebnisse erzielen.

- Die binäre Entscheidung zwischen Interpreter und Compiler wird zugunsten von Systemen mit mehreren unterschiedlich stark optimierenden Compilern aufgegeben. Diese bieten die Möglichkeit durch die Verwendung von unterschiedlichen Optimierungsalgorithmen einzelne Programmteile auf der anscheinend geeignetsten Optimierungsstufe zu übersetzen. Je nach Laufzeitverhalten kann eine Methode über verschiedene Optimierungsstufen hin „befördert“ werden, wenn sich herausstellt, daß sie auch nach moderater Optimierung weiterhin einen dominierenden Anteil an der Laufzeit einnimmt.

  Systeme, die mehrere Compiler einsetzen, sind unter anderem in der Sun Research VM, der IBM Jalapeño JVM sowie der Intel Microprocessor Research Lab VM implementiert worden [Agesen and Detlefs, 1999b, Alpern et al., 2000, Cierniak et al., 2000]. Die Basis-Compiler arbeiten dabei extrem effizient bei akzeptabler Codequalität und eignen sich daher als vollständiger Ersatz für den Interpreter.

3.1.4 Diskussion

Dynamische Compiler arbeiten mit einem deutlich geringeren Zeitbudget als statische Compiler. Verzichtet man auf komplexe Optimierungsalgorithmen, ist die erzielte Leistung unbefriedigend. Setzt man hingegen großflächig klassische Optimierungen ein, so ist der Programmstart zu träge und die aufgewendetet Zeit wird möglicherweise nicht wieder eingebracht.

Optimierende dynamische Compiler arbeiten daher mit *effizienten* Algorith-

men, d.h. unter Berücksichtigung ihrer Laufzeit, *selektiv*, d.h. primär auf den *hot spots* des Programms und es werden *mehrere Compiler* unterschiedlicher Charakteristika eingesetzt.

Modelle für die Kosten-Nutzen-Rechnung dynamischer Compiler sind quantitativ nur wenig aussagekräftig; hier liegen allenfalls Heuristiken zur Entscheidungsfindung vor.

3.2 Erhebung von Laufzeitprofilen

Dynamische Optimierungen sind auf Informationen über das Laufzeitprofil angewiesen. Insbesondere drei Entscheidungen müssen unterstützt werden:

- Welche Methoden erfordern Optimierung?
- Welche Aufrufkanten sollten durch dedizierte Typprüfungen spezialisiert werden?
- Welche Aufrufkanten sollten integriert werden?

Welche Informationen müssen also erhoben werden, um genannte Entscheidungen treffen zu können, und auf welche Weise ist dies möglich?

Zunächst lassen sich maß- und stichprobenbasierte Methoden unterscheiden. Erstere erstellen ein genaues Profil, indem sie jede relevante Programmoperation im Profil festhalten. Dies kann durch Aufrufzähler und Zeitmessung jeder Methodenausführung geschehen. Die Erhebung von Stichproben (*sampling*), verfolgt einen statistischen Ansatz: eine Methode, die sich in $p\%$ aller Stichproben in Ausführung befindet, wird auch diesen Anteil an der Programmlaufzeit einnehmen[2]. Laut [Arnold et al., 2000c] erreichen stichprobenbasierte Methoden auch für kurzlebige Programme ausreichende Aussagekraft.

Gemäß Abschnitt 3.1 stellt der Anteil p einen guten Indikator für Übersetzungskandidaten dar; Aufrufzähler geben ein ungenaueres Bild, Zeitmessungen erscheinen dagegen relativ aufwendig. Zur Ermittlung der *hot spots* eines Programms bieten sich folglich Stichproben an.

Ein Typtest an einer Aufrufstelle eignet sich nur, wenn der Empfänger in der überwiegenden Zahl der Fälle den getesteten Typ aufweist. Um einen geeigneten Kandidaten zu ermitteln, muß also die Verteilung der Empfängertypen

[2]Dies ist natürlich unter Berücksichtigung von blockierten oder wartenden Threads zu sehen.

an einer Aufrufstelle erhoben werden. Auch dies ist mit direkter Messung möglich: an die Aufrufstelle wird ein Aufrufzähler eingebettet, der die Vorkommen einer kleine Anzahl von Empfängertypen zählt. Geschieht dies durch direkte Einbettung einer kaskadierten Testsequenz in den Code, so wird als Nebeneffekt bereits eine Beschleunigung des Aufrufs erzielt. Der auf diese Weise realisierte *polymorphic inline cache*(PIC) [Hölzle et al., 1991] dient also gleichzeitig der Optimierung als auch der Profilerstellung.

Die Erhebung von Typverteilungen auf statistischem Wege gestaltet sich komplizierter: beim mehrfacher Beobachtung desselben (Aufrufer, Ziel)-Paares ist nicht einfach entscheidbar, ob es sich um unterschiedliche Aufrufe handelt, oder ob dieselbe — langlebige — Aktivierung beobachtet wurde. Aus diesem Grund kann nur ein Instruktionszeiger, der im Prolog der Zielmethode erhoben wird, als Indiz für eine neuerliche Aktivierung betrachtet werden; der Prolog wird mit ausreichender Wahrscheinlichkeit innerhalb des Stichproben-Intervalls linear durchlaufen und kann somit nicht zweimal erhoben werden. Es besteht nunmehr allerdings die Gefahr, daß die kurzen Methodenprologe zu klein sind, um in ausreichendem Maße stichprobenrelevant getestet werden zu können. In diesem Zusammenhang erweist sich eine vom Programmpunkt abhängige Unterbrechungsstrategie wie die der Jalapeño-VM (siehe 5.3.1) von Vorteil. Stichproben werden hier nämlich jeweils an Unterbrechungspunkten (s. Abschnitt 2.3.4 und 5.3) entnommen. Da Threads ausschließlich in Methodenprologen und an Rückwärtssprüngen in Schleifen unterbrochen werden, erhöht dies die statistische Aussagekraft der Aufrufverteilungen in erheblichem Maße.

Nach Untersuchungen von [Arnold et al., 2000a] arbeiten Heuristiken zur Methodenintegration am erfolgreichsten, wenn das Aufrufprofil Aufrufe nach ihrem Aufrufer aufschlüsselt. An unterschiedlichen Aufruforten ist die Integration einer bestimmten Methode mehr oder weniger sinnvoll; die Profildaten helfen hier, Integrationskandidaten gegeneinander abzuwägen.

Mehrere virtuelle Maschinen arbeiten mit profilgesteuerten Optimierungen. Die Self-VM [Sun Microsystems, 2000b] enthält Methodenzähler und *polymorphic inline caches*, die vom optimierenden Compiler ausgewertet werden. Die Jalapeño-VM enthält eine Profildatenbank, die durch regelmäßige Stichproben an Unterbrechungspunkten gefüllt wird. Swift [Scales et al., 2000], der optimierende Compiler der virtuellen Java-Maschine für die Alpha-Architektur von Compaq, bedient sich der Profil-Infrastruktur DCPI [Anderson et al., 1997], die vom Betriebssystem und dem Prozessor zur Verfügung gestellt werden.

3.3 Deoptimierung

Der dominante Anwendungsfall eines Algorithmus eignet sich häufig besonders für Optimierungen. Dies ist unter Umständen mit höheren Kosten für die selten ausgeführten Spezialfälle verbunden. Dieser Ansatz lässt sich soweit verfolgen, daß ein Algorithmus allein für den *fast path* übersetzt und optimiert wird. Um inkorrekte Ergebnisse im Spezialfall zu verhindern, muß dieser erkannt werden und die bereits angewandten Optimierungen rückgängig gemacht werden. Diese Technik ist in mehreren Kontexten gebräuchlich:

- Fehlersuche: während durch Optimierungen wie *constant propagation* Variablen ganz eliminiert werden können, muß im Debugger die Änderung dieser Variable den erwarteten Effekt auslösen; sie ist nicht mehr konstant.
- Methodenaufrufe: existiert zum Zeitpunkt der Übersetzung nur eine Implementierung der gerufenen Methode im System, so kann diese statisch gebunden oder integriert werden (siehe auch 3.4.1). Wird zu einem späteren Zeitpunkt eine weitere Implementierung in das System gebracht, so muß die aufrufende Methode deoptimiert werden.
- Ausnahmebehandlung: wie erläutert, behindert die Vielzahl von möglichen ausnahmeauslösenden Operationen in Java in Verbindung mit präziser Ausnahmebehandlung die Optimierung in erheblichem Maße. Dieser Effekt läßt sich mildern, indem die entsprechenden Kontrollflußkanten bei der Optimierung zunächst ignoriert werden. Im — per definitionem seltenen — Ausnahmefall müssen die Optimierungen unter Umständen kompensiert werden, um die präzise Exception-Definition zu erfüllen.

Zwei Techniken im Zusammenhang mit Deoptimierung seien im folgenden dargestellt: *on-stack replacement* und Optimierung der Ausnahmebehandlung.

3.3.1 On-Stack Replacement

Methodenintegration ist ein wichtige Technik zur Optimierung von Methodenaufrufen. Dabei werden teilweise monomorphe Aufrufstellen integriert, die durch späteres dynamisches Laden neuen Programmcodes nicht länger eine eindeutige Zielmethode besitzen. Es muß also sichergestellt werden, daß

```
int dogs(Animal[] animals) {
  int dogs = 0;
  for(int i = 0; i < animals.length; i++) {
    Animal a = animals[i];
    if(a.isDog()) dogs++;
  }

  return dogs;
}
```

Abbildung 3.1: OSR-Beispiel

```
int dogs(Animal[] animals) {
  <null-check> animals;
  for(int i = 0; i < animals.length; i++) {
    Animal a = animals[i];
    <null-check> a;
  }

  return 0;
}
```

Abbildung 3.2: Optimierter Code

der optimierte Aufruf nicht mit einem Exemplar der neu geladenen Klasse als Empfänger ausgeführt wird.

Ein Teil der Lösung besteht darin, bereits zur Ladezeit die betroffene Methode neu zu übersetzen, bzw. als interpretiert zu markieren, und in ihrer Klasse neu zu installieren. Erst dann können Exemplare der neuen Klasse erzeugt werden. Dieses Vorgehen läßt allerdings ein Problem unberücksichtigt: es können weiterhin Aktivierungen der alten, optimierten Methode existieren. Diese könnten ein Exemplar der neuen Klasse erreichen und mit diesem den bereits ungültigen Code ausführen. Es gilt also, auch die Aktivierungen der ungültigen Methode zu ersetzen. Dazu muß eine Zuordnung der Stack- und Registerinhalte der optimierten Methode zu den Variablen der Quellmethode erfolgen. Aufgrund von Platzerwägungen sind solche Zuordnungsstrukturen nicht für jede Instruktion, sondern nur an dedizierten Punkten, typischerweise Methodenprologe und Rückwärtssprünge, zur Verfügung zu stellen.

Ein Beispiel erläutert die Problematik. Gegeben sei folgende Methode

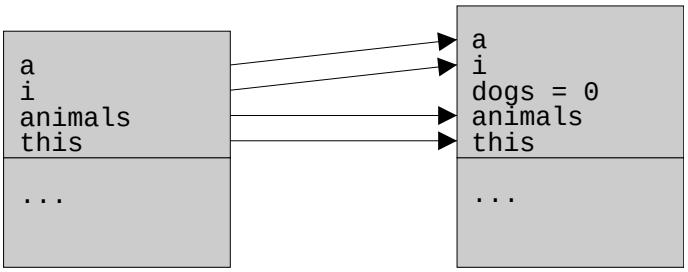

Abbildung 3.3: On-Stack Replacement

`dogs`(3.1), die aus einer Sammlung von Tieren die Zahl der Hunde ermittelt. Da zum Übersetzungszeitpunkt nur die Tier-Klasse `Cat` mit `Cat::isDog() = false` geladen ist, kann der Compiler wie in Abbildung 3.2 optimieren. Insbesondere die lokale Variable `dogs` ist im optimierten Rahmen nicht vorhanden, so daß eine Aktivierung von `dogs` im optimierten Fall um eine lokale Variable kleiner ist als ihre Ersatzaktivierung. Wird nun eine Klasse `Dog` hinzugeladen, und wird ein Exemplar in das Array `animals` gespeichert, während eine Aktivierung von `dogs` in einem anderen Thread ausgeführt wird, so muß dieser Thread suspendiert und sein Laufzeit-Stack entsprechend angepasst werden (Abbildung 3.3). Ein Rahmen einer optimierten Methode kann aufgrund von Methodenintegration dabei sogar mehreren unoptimierten Rahmen entsprechen.

On-stack replacement ist ein sehr komplexes Verfahren, da Laufzeit-Stacks manipuliert werden und eine Rückwärtsabbildung von optimiertem Code auf die Programmgrößen implementiert werden muß. Aus diesem Grunde schlagen [Agesen and Detlefs, 1999a] eine Einschränkung für ungeschütztes Inlining vor: kann bewiesen werden, daß der fragliche Empfänger des integrierten Aufrufs *vor Aktivierung* der aufrufenden Methode bereits existierte, so wird diese Aktivierung auch durch das Laden neuer Klassen nicht ungültig gemacht. Die Methode muß zwar neu übersetzt werden, laufende Aktivierungen können jedoch intakt bleiben. Im obigen Beispiel wird der Empfänger aus einer externen Struktur entnommen; in diesem Fall würde die vorgeschlagene Vereinfachung daher nicht zutreffen.

On-stack replacement wird in der SELF-VM unter anderem zum Debugging verwendet [Hölzle et al., 1992]. Die HotSpot-VM deoptimiert Methoden, deren Voraussetzungen zur Methodenintegration invalidiert wurden, durch OSR.

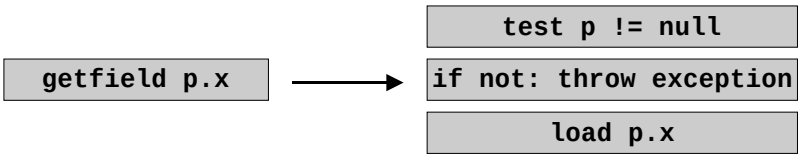

Abbildung 3.4: Laufzeittests

3.3.2 Ausnahmebehandlung

Das Umordnen von Programmstücken ist ein wichtiges Verfahren zur Optimierung. Durch die Forderung nach präziser Ausnahmebehandlung (s. Abschnitt 2.1.2) im Zusammenhang mit vielen laufzeitgeprüften Operationen verhindert die Java-Spezifikation diese Option. Akzeptiert man jedoch für den Ausnahmefall eine langsamere Ausführung, gewinnt man einen Teil der möglichen Optimierungen zurück. Der hier vorgestellte Algorithmus geht auf [Gupta et al., 2000] zurück.

Die Grundidee ist dabei die folgende: die betroffenen Operationen (beispielsweise `getfield p.x`) werden für die interne Compiler-Repräsentation aufgeschlüsselt. Sie bestehen aus drei Teilen (siehe Abbildung 3.4): ein Teil prüft, ob die Ausnahmebedingung vorliegt (`p == null`), einer besteht aus der Ausnahme selbst (`throw new NullPointerException()`) und einer aus dem eigentlichen Kern der Operation (`load p.x`)[3]. Diese Teilstücke können vom Optimierer daraufhin freier umgeordnet werden; nur Operationen, die global sichtbare Seiteneffekte hervorrufen, dürfen natürlich nicht vor vorhergehende Ausnahmeprüfungen bewegt werden.

Der so erzeugte Code weist zwei Abweichungen von der präzisen Ausnahmedefinition auf:

- Es kann der falsche Typ von Ausnahme ausgelöst werden
- Aktionen, die im Programm vor einer Ausnahmeoperation liegen, sind noch nicht ausgeführt

Es muß daher zusätzlicher Kompensations-Code erzeugt werden, der diese Abweichungen bereinigt. Dieser weist dem Grunde nach die gleiche Struktur wie der optimierte Code auf, es bestehen jedoch die folgenden Unterschiede:

[3]Im aktuellen Maschinencode fallen diese Teile teilweise wieder zusammen, z.B. wenn der Test implizit beim Speicherzugriff durch Hardware-Mechanismen implementiert ist.

- die Stellen, an denen Ausnahmen ausgelöst werden, setzen nur entsprechende Flags. Die eigentliche Ausnahme wird bei der Operation selbst ausgelöst.
- Operationen, die vor andere bewegt wurden, dürfen nur ausgeführt werden, wenn deren Ausnahme-Flag nicht gesetzt ist.

Wird eine Ausnahme ausgelöst, so findet ein Kontrolltransfer an die entsprechende Stelle im Kompensations-Code statt, der den Algorithmus auf korrekte Weise zuende führt.

Ein Beispiel verdeutlicht das Vorgehen für den Ausdruck `p.x = a[i]` (Abbildung 3.5): angenommen, der Null-Test der Referenz `p` und der Array-Index-Test sind vom Compiler vertauscht worden. Wird nun eine `NullPointerException` durch Test (2) ausgelöst, so muß der Kompensationscode noch den Array-Test nachvollziehen und u.U. die korrekte `ArrayIndexOutOfBoundsException` auslösen.

Die Generierung des Kompensationscode verdoppelt den Platzbedarf der Methode. Es wäre daher von Vorteil, diesen bei Bedarf vom dynamischen Compiler erzeugen zu lassen. Ausnahmen sollten daher nicht zur Kontrollflußsteuerung eingesetzt werden, sondern für den Ausnahmefall reserviert sein.

3.3.3 Zusammenfassung

Berücksichtigt man bei der Codeerzeugung alle möglichen Programmabläufe, so erzielt man unter Umständen nicht zufriedenstellende Programmlaufzeiten ein. Deoptimierung erlaubt es, den Algorithmus auf den häufigen Fall einzuschränken und dadurch gezielter zu optimieren. Tritt der Ausnahmefall ein, wird z.B. eine neue Klasse geladen oder tritt ein Laufzeitfehler auf, so kann dynamisch darauf reagiert werden — die damit verbundenen Kosten werden durch den zuvor erzielten Gewinn ausgeglichen.

3.4 Globale Analyseverfahren

Alle bis hierher geschilderten Compiler-Optimierungen arbeiten nur im Kontext der aktuell compilierten Methode und — im Falle der Methodenintegration – ihrer direkten Kinder. Es existieren aber auch Optimierungen, die mit den auf dieser Ebene gewonnenen Informationen nicht ausgeführt werden können.

```
try:
  trap if a == 0            ; (1) NullPointerException
  r0 = load a.length        ; Array-Länge
  trap if p == 0            ; (2) NullPointerException
  trap if r0 <= i           ; (3) ArrayIndexOutOfBoundsException
  i = i << 2                ; index in Bytes umrechnen
  r1 = load a, i            ; Array lesen
  store p.x r1              ; Feld schreiben
catch:                      ; Kompensationscode
  if(source == 1) ex1=true, goto ex1
  if(source == 2) ex2=true, goto ex2
  if(source == 3) ex3=true, goto ex3

  ; modifizierte Kopie von oben
  if(a==0) ex1=true       ; Test (1)
ex1:
  if(!ex1) {
    r0 = load a.length
  }
  if(p == 0) ex2=true     ; Test (2)
ex2:
  if(!ex1) {
    if(r0 <= i) ex3=true  ; Test (3)
ex3:
    if(!ex2) {
      i <<= 2
      if(ex2) throw new NullPointerException()
      if(ex3) throw new ArrayIndexOutOfBoundsException()
      r1 = load a, i
    }
  }
  if(ex1) throw new NullPointerException()
  store p.x, r1
```

Abbildung 3.5: Generierter Code für `p.x = a[i]`

Soll beispielsweise bestimmt werden, ob ein dynamischer Methodenaufruf einer Methode `m` in der Klasse `C` in einen statischen Methodenaufruf umgewandelt werden kann, so müssen zur Analyse alle Subklassen von `C` herangezogen werden. Oder soll bestimmt werden, ob ein in der Methode `m` erzeugtes Objekt auf dem Stack alloziert werden kann, da seine Lebenszeit durch die Ausführung von `m` begrenzt ist, so müssen alle von `m` potentiell gerufenen Methoden analysiert werden.

Viele fortgeschrittene Optimierungen erfordern daher ein globales Wissen, d.h., sie sind auf die Kenntnis des gesamten Klassenbaums oder zumindest eines Teilbaumes angewiesen. Diese sogenannten „whole program optimizations“ werden üblicherweise als zu aufwendig erachtet und daher nicht eingesetzt. Besoders in der dynamischen Umgebung einer Java-Maschine, wo erst zur Laufzeit kompiliert wird und wo durch Laden neuer Klassen einmal gemachte Annahmen invalidiert werden können, scheint eine globale Analyse zu teuer zu sein. Da man sich hier jedoch ein großes Potential bei der Optimierung objektorientierte Programme verspricht, findet eine aktive Forschung zur Entwicklung effizienterer Algorithmen zur globalen Analyse statt.

Im den folgenden Abschnitten sollen 2 Arten globaler Analyseverfahren vorgestellt werden: *class hierarchy analysis* (Abschnitt 3.4.1) analysiert die Klassenhierarchie mit dem Ziel, dynamische durch statische Methodenaufrufe zu ersetzen. Die in Abschnitt 3.4.2 dargestellte *escape analysis* prüft, ob ein erzeugtes Objekt der erzeugenden Methode bzw. dem erzeugenden Thread „entfliehen“ kann. Ist dies nicht der Fall, so kann das ensprechende Objekt auf dem Stack alloziert werden bzw. dieses Objekt betreffende Synchronisationsoperationen können unterlassen werden.

3.4.1 Class Hierarchy Analysis

Wie bereits in Abschnitt 2.1.2 beschrieben, stellt der dynamische Methodenaufruf ein großes Problem bei der Optimierung von Java-Programmen dar. Als Beispiel sei der folgende Aufruf auf der in Abbildung 3.6 dargestellten Klassenhierarchie angenommen. In Verlauf der Methode `y` in der Klasse `X` findet eine dynamischer Aufruf der Methode `m` an dem Empfängerobjekt `c` statt.

```
public class X extends Object{
  public void y(C c){
    ...
    c.m();
```

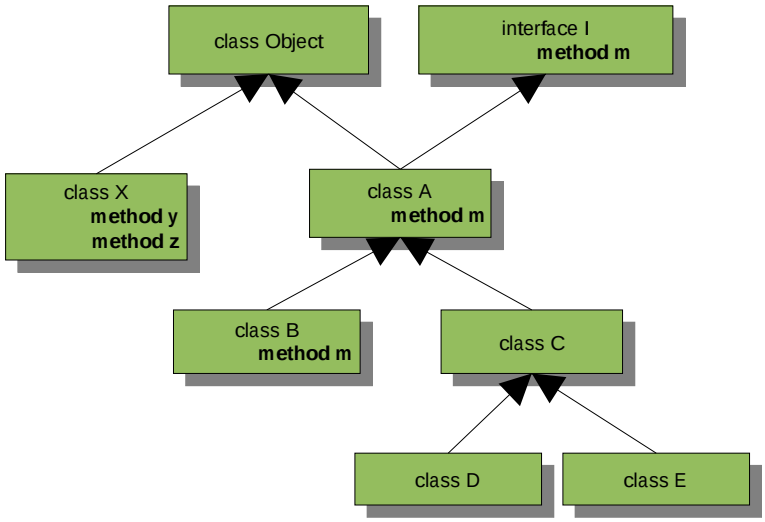

Abbildung 3.6: Beispiel einer Klassenhierarchie

```
    ...
  }
}
```

Ohne globale Analysen kann der Aufruf von `m` nur mit einer Sicherungsinstruktion (*guard*) in einen statischen Aufruf gewandelt werden, da einen Subklasse von `C` die aus `A` geerbte Implementierung überschreiben könnte. Es könnte also lediglich der folgende Umstellung des Codes vorgenommen werden.

```
public class X extends Object{
  public void y(C c){
    ...
    if(c.class == C)          // guard
      c.A::m();               // perform static call
    else                      // if class test fails
      c.class.dispatch(m);    // perform dynamic dispatch
                              // on receiver
    ...
  }
}
```

Die Auswirkung dieser Maßnahme auf die Performance hängt von der Aufrufhäufigkeit von `y` mit einer Instanz von `C` als Empfänger ab – vor einer

solchen Optimierung sollte also eine Erhebung von Laufzeitprofilen erfolgt sein.

Hat der Optimierer jedoch Kenntnis von **allen** Subklassen von `C`, so kann der dynamische Aufruf von `m` direkt in einen statischen Aufruf von `A::m` ohne *guard* gewandelt werden, da keine der geladenen Klassen Subklassen von `C` (`D` und `E`) diese Methode überschreibt. Für jeden Aufruf von `X::y` *muß* also `A::m` aufgerufen werden. Der Optimierer kann demnach den folgenden Code produzieren:

```
public class X extends Object{
  public void y(C c){
    ...
    c.A::m();                // perform static call w/o guard
    ...
  }
}
```

Dieser Optimierung kann jetzt eine Integration der Methode `A::m` in den Methodenkörper von `y` folgen.

Die im Beispiel dargestellte Optimierung ist nur aufgrund der statischen, klassenbasierten Typisierung Javas möglich. Erst durch die Information über die Klassen des Parameters `c` wird klar, welche Methode gerufen werden soll. Sowohl in dynamisch typisierten Systemen (z.B. Smalltalk), als auch in statisch typisierten Systemen mit struktureller Subtypisierung (z.B. Tycoon-2 [Ernst, 1998]) ist die Möglichkeit der Optimierung durch globale Klassenanalysen stärker beschränkt.

Optimierung aufgrund von globalen Analysen der Klassenhierarchie findet – zumindest in moderater Form – in praktisch jeder Java-Maschine mit optimierendem Compiler statt. [Dean, 1996] hat diese Form der Analyse bereits 1996 als *class hierarchy analysis* formalisiert und eine Implementierung für den Vortex-Compiler beschrieben.

3.4.2 Escape Analysis

Da Java die nebenläufige Programmausführung in mehreren Threads unterstützt, müssen Datenstrukturen, die von mehrern Threads nebenläufig verwendet werden können, in kritischen Abschnitten durch Synchronisation geschützt werden. Dies ist der Grund, weshalb beispielsweise die Java-Klasse

`java.util.Vector` mit Hilfe von Sychronisationen threadsicher implementiert wurde.

Jedoch ist nicht jede Verwendung von `java.util.Vector` eine Nebenläufige, wie aus dem in Abbildung 3.7 dargestellte Beispiel aus [Whaley and Rinard, 1999] ersichtlich wird. Es zeigt, wie durch globale Analyse thread-lokale Objekte identifiziert werden; dies sind Objekte, die nur von einem Thread erreichbar sind. Auf derartigen Objekten kann einen Synchronisation entfallen, da Konflikte durch nebenläufigen Zugriff ausgeschlossen sind.

```
 1:  public class Server extends Thread{
 2:    Serversocket Serversocket;
 3:    Int Duplicateconnections;
 4:
 5:    Server(Serversocket S){
 6:      Serversocket = S;
 7:      Duplicateconnections = 0;
 8:    }
 9:    public void run(){
10:      try {
11:        Vector connectorList = new Vector();
12:        while(true){
13:          Socket clientSocket = serverSocket.accept();
14:          new ServerHelper(clientSocket).start();
15:          InetAddress addr = clientSocket.getInetAddress();
16:          if(connectorList.indexOf(addr) < 0)
17:            connectorList.addElement(addr);
18:          else
19:            duplicateConnections++;
20:        }
21:      } catch (IOException e) {}
22:    }
23:  }
```

Abbildung 3.7: Beispiel eines einfache Servers

Im Beispiel wird ein einfacher Server implementiert, der eingehende Verbindungen auf einer `ServerSocket` annimmt. Für jede neue Verbindung wird in Zeile 14 ein neuer `ServerHelper`-Thread erzeugt und gestartet. Danach wird durch Suche in dem `Vector connectorList` geprüft, ob be-

reits eine Verbindung zu dem neuen Client besteht und ggf. die Variable `duplicateConnections` erhöht.

Durch Analyse der geschilderten Methode `Server::run` und der beiden Methoden `Vector::indexOf` und `Vector::addElement`, die mit dem Empfänger `connectorList` aufgerufen werden, kann der Optimierer beweisen, daß ein in `connectorList` gespeichertes Objekt dem erzeugenden Thread nicht entfliehen kann – es also nicht *thread escaping* ist. Dies bedeutet, daß unter keinen Umständen ein anderer Thread als der erzeugende eine Referenz auf das von `connectorList` referenzierte Objekt erhalten kann.

Das Beispiel illustriert noch eine weitere Eigenschaft: Normalerweise werden in Java nur Exemplare der Basisdatentypen (`int`, `long`, `boolean`, etc.) auf den Stack alloziert, während alle Objekte (d.h. Instanzen einer Java-Klasse) generell auf dem Heap angelegt werden. Dadurch können Refenzen auf Objekte, die innerhalb einer Methode angelegt wurden, in globalen Variablen abgelegt werden, wodurch sie weiterhin erreichbar und lebendig bleiben.

Wenn ein Objekt jedoch nicht den Aufruf der Methode überlebt, in der es erzeugt wurde, d.h. es von keiner Root im System erreichbar ist, nachdem die Ausführung der Methode beendet wurde, so könnte dieses Objekt auch im Stackframe des entsprechenden Methodenaufrufes angeleget werden.

Dies kann von Vorteil sein, da in der Regel selbst bei sogenannter „schneller“ Heap-Allokation (s. Abschnitt 4.1) eine reine Stack-Allokation performanter als die Heap-Allocation in einem von einem vom Garbage Collector kontrollierten Heap ist. Dies liegt zum einen darin begründet, daß bei einer Heap-Allokation, zumindest gelegentlich, eine Synchronisation über das Heap-Lock erfolgen muß; zum anderen muß seltener eine Garbage Collection ausgeführt werden, wenn weniger Objekte auf dem Heap angelegt werden. Dabei muß allerdings ein übermaßiges Wachsen des Stacks vermieden werden, weshalb eine Stack-Allokation innerhalb einer Schleife üblicherweise vermieden wird.

Wie oben kann auch hier durch Analyse der genannten Methoden `Server::run`, `Vector::indexOf` und `Vector::addElement` bewiesen werden, daß die Lebensdauer eines in `connectorList` gespeicherten Objektes durch die Ausführung der erzeugenden Methode begrenzt ist – es also nicht *method escaping* ist.

Der Optimierer könnte also die Methode `Server::run` folgendermaßen modifizieren:

1. Obwohl die Methoden `Vector::indexOf` und `Vector::addElement` als `synchronized` deklariert sind, kann jegliche Synchronization an dem

von `connectorList` referenzierten Objekt unterlassen werden.

2. Das in Zeile 11 erzeugte Objekt, das der Variablen `connectorList` zugewiesen wird, kann auf dem Stack im Rahmen des aktuellen Methodeaufrufes alloziert werden, da es mit Verlassen der Methode `Server::run` unerreichbar wird.

Nun stellt sich die Frage, wie eine Analyse vorzugehen hat, um zu beweisen, daß ein bestimmtes Objekt `o` von keinem anderen Threads als dem erzeugenden erreichbar wird. Prizipiell muß bewiesen werden, daß `o`:

- niemals mit dem Bytecode `putstatic` einer Klassenvariablen zugewiesen wird.
- niemals mit dem Bytecode `putfield` oder `aastore` einem Objektfeld zugewiesen wurde dessen Besitzer dem Thread entflieht.

Soll `o` auf den Stack alloziert werden so muß zusätzlich bewiesen werden, daß es nicht durch den Bytecode `areturn` den Bereich seiner erzeugenden Methode verläßt.

Es existieren mittlerweile eine große Zahl von Implementierungen einer *escape analysis* für Java [Blanchet, 1999, Bogda and Hölzle, 1999, Choi et al., 1999, Whaley and Rinard, 1999]. Sie alle führen eine intraprozedurale kontextinsensitive Datenflußanalyse [Muchnick, 1997] durch, deren Ergebnis in kompakter Form für eine interprozedurale Analyse verwendet wird. Dabei werden im Falle von Rekursionen in der Regel konservative Annahmen gemacht, um eine akzeptabele Performance zu erhalten. Alle genannten Implementierungen verwenden einen statischen Compiler und erreichen damit eine Performanceverbesserung des ausgeführten Java-Programmes von 7 bis 40 %.

Wie bereits angedeutet, stellt die Komplexität des Algorithmus bei allen globalen Analysen ein entscheidendes Problem dar. Bei Verwendung des ersten veröffentlichten Algorithmus zur *escape analysis* [Park and Goldberg, 1992] stieg der Zeitaufwand exponentiell mit der Größe des analysierten Programms.

Es kann davon ausgegangen werden, daß alle der oben genannte Algorithmen für Java in polynomieller Zeit arbeiten, Ergebnisse einer Komplexitätsanalyse und Performanzmessung des Algorithmus werden allerdings nur von [Blanchet, 1999] genannt. Der Zeitaufwand für die Analyse eines Programms steigt demnach maximal quadratisch mit seiner Größe – in der Praxis wurde

aber nur ein lineares Wachtum festgestellt. Dabei wurde die Kompilationszeit im Mittel um 34% erhöht. Diese Zahlen lassen eine Integration einer *escape analysis* in einen dynamischen Compiler möglich erscheinen.

3.4.3 Diskussion

Globale Analysen ermöglichen gute Optimierungen, selbst wenn im analysierten Programm verstärkt das Mittel der Datenabstraktion angewendet wird. Gerade in Java werden Interfaces häufig nur von *einer* Klasse im System implementiert, da viele Standards in Form von Java-Interfaces beschrieben werden. Diese werden dann von verschiedenen Herstellern konkret implementiert, wobei aber üblicherweise in einer VM nur eine Implementierung zum Einsatz kommt. Z.B. wird in einer Instanz eines Java-Programms in der Regel lediglich *eine* Implementierung des JDBC[4]-Interfaces geladen werden, da eine Anwendung üblicherweise nur mit *einer* Datenbank kommuniziert. Durch eine globale Analyse der Klassenhierarchie kann der Optimierer beweisen, daß alle Aufrufe, die typstatisch an das JDBC-Interface gerichtet sind, die einzige Implementierung aufrufen müssen. Solange ein Interface von nur einer Klasse implementiert wird, entsteht durch diese zusätzliche Abstraktion also keine Verschlechterung der Performanz.

Allerdings muß bei allen globalen Analysen die Komplexität des gewählten Algorithmus wegen der Größe der Eingangsdaten besonders kritisch betrachtet werden. Ein Algorithmus der auf realen Daten mit quadratischer Komplexität arbeitet, kann hier als prinzipiell unbrauchbar eingestuft werden. Besonders bei der Verwendung eines dynamischen Compilers, wo sich die aufgewendete Übersetzungszeit amortisieren muß, kommen nur Algorithmen in Frage, die, zumindest in der Praxis, ein lineares Verhalten zeigen. Algorithmen, die eine kontextsensitive Datenflußanalyse ausführen scheiden daher aufgrund ihrer zu hohen Komplexität für globale Analysen aus [Grove et al., 1997].

Ein anderer Punkt, der im Zusammenhang von globalen Analysen und dem in Java praktizierten dynamischen Laden von Klassen auftritt, ist die Deoptimierung. Wird im o.g. Beispiel eine neue Subklasse von `C` in das System geladen, so muß die gemachte Annahme „keine Subklasse von `C` überschreibt `m`“ überprüft werden. Sollte sie nicht mehr gegeben sein, so muß eine Deoptimierung (siehe Abschnitt 3.3) von `X::y` stattfinden.

[4]Java Database Connectivity

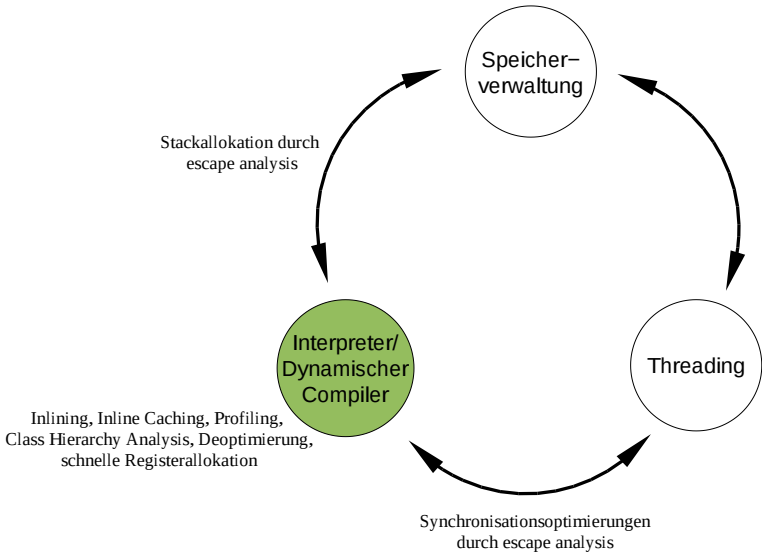

Abbildung 3.8: Die drei Hauptkomponenten aus Sicht des dynamischen Compilers

3.5 Zusammenfassung

Abbildung 3.8 zeigt die in Kapitel 2 eingeführte Grafik der 3 Hauptkomponente virtueller Maschinen, wobei hier einige Techniken und Aufgaben des dynamischen Compilers annotiert wurden.

Viele der erwähnten Techniken zählen bereits zum Standardrepertoire einer modernen virtuellen Maschine. *Inlining* und *inline caching* sind bereits seit langem bekannt und sind in vielen Implementierungen realisiert. Dynamische Profile werden prinzipiell von jeder Maschine erhoben, die über einen dynamischen Compiler verfügt – diese beschränken sich jedoch häufig auf einfache Aufrufzähler.

Dabei ist eine leistungsfähige dynamische Profilgewinnung eine Schlüsselfunktionalität, die es dem dynamischen Compiler erlaubt, effizienteren Code zu erzeugen, als der statische. So ist es der der IBM Production-VM erstmals gelungen, mit dynamischer Compiler-Technologie den 1. Platz der Volano-Benchmarks [Volano, 2000] zu belegen und damit TowerJ als statischen Compiler zu verdrängen.

Die *class hierarchy analysis* stellt eine weitere fortgeschritten Technik dar, die im Zusammenhang mit Optimierungen wie dem Umwandeln von dynamische in statische Aufrufe und der Methodenintegration ohne Sicherungsinstruktion die Möglichkeit der Deoptimierung erforderlich macht.

Wie gezeigt wurde, ist die Technik der Deoptimierung zwar prinzipiell verstanden, sie trägt jedoch erheblich zu der Komplexität eines dynamischen Compilers bei. [Agesen and Detlefs, 1999a] berichten, daß eine großer Teil der Komplexität des SELF-Systems auf Deoptimierung und Reoptimierung zurürckzuführen ist.

Eine Technik, der in diesem Kapitel nur wenig Aufmerksamkeit geschenkt wurde, ist die Interpretation von Code. Zwar verfügen auch viele moderne Maschinen noch über einen Interpreter, Wie in Anhang A ersichtlich wird, zeichnet sich jedoch ein deutlicher Trend zu Systemen mit mehreren Compilern ab. Dabei kommt ein schneller, wenig optimierender Compiler an Stelle eines Interpreters zu Einsatz. Der so erzeugte Code, kann schneller ausgeführt werden als interpretierter Code, so daß sich der erhöhte Zeitaufwand für die Kompilation sehr schnell amortisiert.

Zu den Bereichen, in denen in der nächsten Zeit noch erhebliche Fortschritte zu erwarten sind, zählen die globalen Analysen. Eine weitreichende *escape analysis* ist bisher in keinem Produktionssystem zu finden. Algorithmen, die eine schnelle *call graph construction* ermöglichen, werden erforscht.

Weiterhin steigt durch die Eliminierung des Interpreters die Nachfrage nach schnellen Compiler-Algorithmen. Ein Beispiel stellt die *linear register allocation* dar, welche die mit quadratischer Komplexität arbeitenden *graph coloring register allocation* ersetzt. Der erzeugte Code ist in der Ausführung nur wenig langsamer, der Algorithmus aber erheblich schneller als der des komplexeren Pendants. Hier sind noch weitere schnelle Techniken zu erwarten, die „suboptimalen“ Code generieren.

Kapitel 4

Implementierung der Speicherverwaltung

„Kick Butt; Have Fun; Eat Lunch;
Pick up the Garbage!“

Sun's Java Technology
Research Group

In folgenden Kapitel werden verschiedene Techniken vorgestellt, die bei der Implementierung der Speicherverwaltung von Relevanz sind. Dabei wird überprüft, welche der aktuell verwendeten Techniken als gut erforscht bezeichnet werden können, und welche noch erforscht werden bzw. welche Entwicklungen in den nächsten Jahren zu erwarten sind.

Dazu wird in Abschnitt 4.1 die Implementierung eines schnellen Allokationsverfahrens auf Basis kompaktierender Garbage Collection beschrieben. Die nötigen Schnittstellen zum Compiler und dem Thread-System aufgezeigt werden.

In Abschnitt 4.2 wird die Implementierung von Exaktheit in einer interpretierenden Java-Maschine beschrieben. Hier zeigen sich die Komplexität exakter Garbage Collection ohne Markierung und eine eklatante Schwäche in der aktuellen Java-Bytecode-Spezifikation. Weiterhin wird klar, welche Informationen der Compiler für exakte Garbage Collection bereitstellen muß.

Im folgenden Abschnitt 4.3 wird die Methode der generationalen Garbage

Collection beschrieben, die in praktisch jeder modernen virtuellen Maschine angewendet wird. Auch hier zeigt sich, daß die Garbage Collection eng mit dem Compiler zusammenarbeiten muß, um die Zuweisung von Zeigern zwischen den Generationen zu protokollieren. Daraufhin werden in den Abschnitten 4.4 und 4.5 Techniken zur Implementierung von inkrementellen, nebenläufigen sowie effizienten multiprozessorfähigen Collectoren dargestellt.

Abschnitt 4.6 dient der Zusammenfassung und Bewertung der geschilderten Techniken.

4.1 Allokation und Objektrepräsentation

Sowohl in objektorientierten als auch in funktionalen Sprachen werden Heapobjekte üblicherweise mit einer sehr hohen Frequenz angelegt. [Appel, 1989b] nennt für SML/NJ, eine Implementierung der funktionalen Sprache ML, eine durchschnittliche Rate von einem Speicherwort pro 30 Assemblerinstruktionen bzw. eine Allokation alle 90 Instruktionen — die Optimierung der Allokationsroutine erscheint also lohnend. Wird ein Algorithmus verwendet, der unterschiedliche Segmente verwendet (z.B. ein generationaler Algorithmus), so ist zwischen Allokation in der jüngsten und den älteren Generationen zu unterscheiden. Die o.g. Allokationsfrequenzen treten nur in der jüngsten Generation auf, weshalb nur diese hier betrachtet wird.

Ein wichtiges Entwurfsziel ist es, den häufig gewählten Pfad der Allokation so kompakt zu implementieren, daß der betreffende Code vom Compiler direkt im Code (*inline*) eingesetzt werden kann. Weiterhin sollten nur wenige Instruktionen ausgeführt werden müssen, der betreffende Code sollte also keine Schleifen enthalten. Es sind daher Algorithmen vorzuziehen, die den Speicher aus einem linearen Bereich beziehen, d.h. kompaktierende Algorithmen. Varianten, die bei variabler Objektgröße und hoher Allokationsfrequenz eine Freispeicherliste verwenden, sind nicht konkurrenzfähig, da diese nach einem Bereich passender Größe durchsucht werden muß[1]. Bei einem generationalen Algorithmus kann dies für ältere Generationen jedoch durchaus gerechtfertigt sein, da Speicher hier mit deutlich geringerer Frequenz angelegt wird.

Bei Verwendung eines Algorithmus, dem ein linearer Bereich zur Verfüng steht, kann die Allokation folgendermaßen dargestellt werden: der Speicher wird von „unten nach oben“ vergeben, d.h. `free` wird bei jeder Allokation

[1]Eine eigene Liste für jede verwendete Objektgröße führt zu einen erheblich höheren Speicher- und Synchronisierungsbedarf. Genaueres hierzu findet sich bei [Huelsbergen and Winterbottom, 1998]

```
                          ! %g1 = free, %g2 = Freispeichergrenze
                          ! %l0 = n, %l1 = Objektkopf
add   %g1,%i0,%g1         ! erhöhe free
cmp   %g2,%g1             ! Vergleich mit Freispeichergrenze
bg,pn slow_path           ! wenn fehlgeschlagen, rufe Garbage
                          ! Collection
nop                       ! delay slot
st    %l1,[%g1]           ! schreibe Metadaten in das neue Objekt
```

Abbildung 4.1: Einfache nicht thread-sichere Allokation durch Inkrementierung von `free`

inkrementiert. Das Objekt liegt aber in negativer Richtung zu seiner Adresse, d.h. das erste Wort eines Objektes *obj* liegt bei *obj*, das zweite bei $obj - wordsize$. Zur Allokation von n Bytes werden die folgenden Operationen ausgeführt:

1. Inkrementiere `free` um n
2. Vergleiche `free` mit der Freispeichergrenze
3. Falls diese erreicht wurde, rufe den Garbage Collector
4. Schreibe den Objektkopf nach `free`

Das Ergebnis der Allokation liegt nun in `free` vor. Der entsprechende Assembler-Code für einen SPARC-Prozessor besteht aus nur 5 Instruktionen und wird in Abbildung 4.1 dargestellt.

Um diesen Algorithmus thread-sicher zu machen, müßten im Prinzip die ersten beiden Operationen atomar ausgeführt werden. Dies ist so zwar nicht möglich, identisches Verhalten läßt sich jedoch mit Hilfe der SPARC-Operation `CAS` („Compare-And-Swap“) [Herlihy, 1991]) erreichen. Dabei ist zu bedenken, daß der Speicherbus für alle Prozessoren für die Dauer der `CAS`-Operation blockiert ist — die Operation stellt also eine Form der Synchronisierung dar. Da die Register prozessorlokal sind, kann `free` im Fall einer Multiprozessorumgebung nicht mehr in einem Register geführt werden. Stattdessen wird `free` im Hauptspeicher abgelegt und ein Zeiger darauf im Register gehalten. Das Ergebnis ist in Abbildung 4.2 dargestellt – zur Implementierung werden 9 Instruktionen benötigt. Diese Lösung entspricht prinzipiell dem Algorithmus, der in der Sun „HotSpot Performance Engine“ [Sun Microsystems, 1999] verwendet wird.

```
                              ! [%g1] = free, [%g2] = Freispeichergrenze
                              ! %l0 = n, %l1 = metadaten
:retry
   ld     [%g1],%l2           ! lade free nach %l2
   add    %l2,%l0,%l3         ! berechne neuen Wert für free, Ergebnis in %l3
   cas    [%g1],%l2,%l3       ! atomic compare-and-swap
   cmp    %l2,%l3             ! war cas erfolgreich?
   bne,pn retry
   cmp    [%g2],%l3           ! Vergleich mit Freispeichergrenze
   bl,pn  slow_path           ! wenn fehlgeschlagen, rufe Garbage
                              ! Collection
   nop                        ! delay slot
   st     %l1,[%l3]           ! schreibe Metadaten in das neue Objekt
```

Abbildung 4.2: Erweiterung von 4.1 zu einer thread-sicheren Allokation durch Verwendung von `CAS`

[Appel, 1989b] schlägt eine weitere Optimierung des Algorithmus vor, bei der der Vergleich mit der Freispeichergrenze entfällt. Stattdessen wird die Speicherseite oberhalb des Freispeichers durch Betriebssystemmechanismen schreibgeschützt. Durch das Schreiben in den Objektkopf wird eine Speicherschutzverletzung ausgelöst. Dies bedeutet, daß der freie Speicher erschöpft ist und eine Garbage Collection ausgeführt werden muß. Der Assemblercode für die Allokation verkürzt sich also auf 7 Instruktionen (s. Abb. 4.3). Die enthaltene Schleife wird nur im relativ seltenen Fall durchlaufen, wenn zwei Threads zeitgleich eine Allokation ausführen.

Voraussetzung für die Verwendung des Schutzmechanismus zur Allokation ist die Initialisierung des neue Speicherbereiches direkt in der Allokationsroutine. Ein späteres Auftreten des Fehlers würde die Behandlung erheblich verkomplizieren. Appel schlägt seine Methode zur Implementierung funktionaler Sprachen vor, die überwiegend wertbasiert arbeiten und diese Werte auch bei der Allokation initialisieren. Bei objektorientierten Sprachen findet die Initialisierung jedoch vorwiegend im Anwendungscode statt und kann daher nicht in der Allokationsroutine ausgeführt werden. In diesem Beispiel wird deshalb „rückwärts“ adressiert (s. Abb. 4.4), da die Objektmetadaten auf jeden Fall in den Objektkopf geschrieben werden müssen. Durch die Plazierung des Kopfbereichs hinter das Objekt wird die Schutzverletzung auch dann ausgelöst, wenn das neue Objekt nur teilweise in den geschützten Bereich hineinragt.

```
                              ! [%g1] = fsp, [%g2] = Freispeichergrenze
                              ! %l0 = n, %l1 = headerword
:retry
   ld     [%g1],%l2           ! lade aktuellen fsp nach %l2
   add    %l2,%l0,%l3         ! berechne neuen fsp, Ergebnis in %l3
   cas    [%g1],%l2,%l3       ! atomic compare-and-swap
   cmp    %l2,%l3             ! war das cas erfolgreich?
   bne,pn retry
   nop                        ! delay slot
   st     %l1,[%l3]           ! schreibe headerword in das neue Objekt
```

Abbildung 4.3: Bei der Verwendung von Speicherschutzmechanismen kann der Vergleich mit der Freispeichergrenze entfallen.

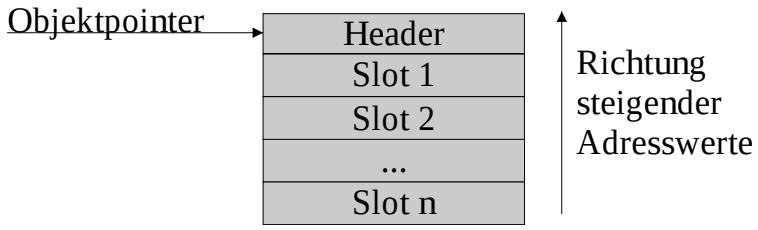

Abbildung 4.4: Mögliche Objekrepräsentation eines objektorientierten Systems bei der Verwendung von Speicherschutzmechanismen

Die Jalapeño Virtual Machine von IBM Research[Alpern et al., 2000] nutzt ähnlich wie [Appel, 1989b] den Speicherschutz des Betriebssystems. Hier wird er allerdings nicht bei der Allokation sondern zur effizienten Implementiertung des Null-Zeiger-Tests verwendet. Eine Java-VM muß vor der Dereferenzierung eines Objektverweises (also bei jedem `invoke` und jedem `getfield/putfield`) testen, ob der Verweis `null` ist. Ist dies der Fall, so muß die Maschine eine `NullPointerException` auslösen [Lindholm and Yellin, 1996].

Dieser Test wird in der für das Betriebssystem AIX implementierten Jalapeño Virtual Machine durch den Speicherschutz des Betriebssystems, also in der Regel durch die *memory management unit* im Prozessor ausgeführt. Um dies zu erreichen wird die in Abbildung 4.5 dargestellte Objektrepräsentation für Skalare und Arrays verwendet. Weiterhin wird die höchste Speicherseite im virtuellen Speicherbereich, beispielsweise der Bereich von `FFFF F000 - FFFF FFFF` im Falle eines 32 Bit Adressraumes und einer Seitengröße von 4K, gegen Lesezugriffe durch die VM geschützt.

Wird bei dem gewählten Layout eine Null-Referenz dereferenziert, und danach ein Feld des Objektes gelesen, so bedeutet dies effektiv einen Lesevorgang aus dem geschützten Speicherbereich. Dies löst eine Ausnahme aus, die die VM in eine `NullPointerException` umwandelt.

Da vor jedem Array-Zugriff eine Indexüberprüfung durchgeführt werden muß, ist es ausreichend, die Längeninformation des Arrayobjektes an einem negativen Offset zur Objektadresse abzulegen plaziert werden. Die Maschine muß also, um den Test ausführen zu können, vor jedem Zugriff die Längeninformation lesen und erreicht somit auf gleiche Weise den Null-Test.

Der in Abb. 4.2 dargestellte Allokations-Algorithmus ist zwar thread-sicher, verwendet jedoch die SPARC-Operation `CAS`. Da diese aufgrund der notwendigen Sychronisation zwischen den vorhandenen Prozessoren relativ teuer ist, verwenden die Sun Research VM [White and Garthwaite, 1998] und die IBM Virtual Machine [Gu et al., 2000] einen zweistufigen Mechanimus: jeder Thread legt zunächst einen lokalen Speicherbereich („Local Allocation Buffer“, LAB) an, aus dem er dann Speicher ohne jegliche Synchronisation vergeben kann. Die Allokation im LAB kann also nach dem Algorithmus aus Abb. 4.1 erfolgen. Ist dieser Bereich erschöpft, so wird unter Verwendung von `CAS` oder eines anderen Synchronisationsmechanismus ein neuer LAB erworben. Da unter Umständen sehr viele Threads gleichzeitig aktiv sein können (in Serversystemen können dies mehrere 100 sein), dürfen die LAB bei einer großen Anzahl Threads nicht zu groß sein. Die Größe der neu erzeugten LAB wird daher in der Sun Research VM abhängig von der Anzahl der Threads

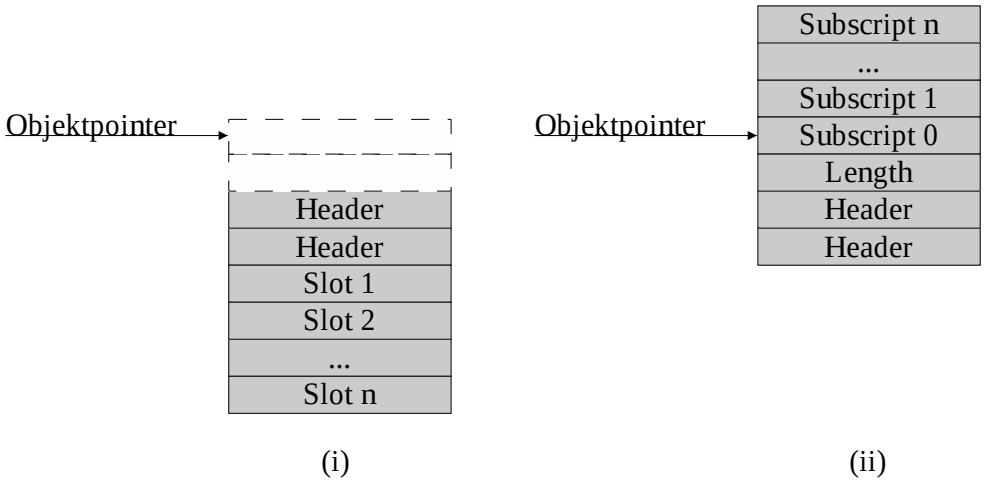

Abbildung 4.5: Objektrepräsentation für (i) Objekte und (ii) Arrays in IBMs Jalapeño Virtual Machine

dynamisch angepaßt.

[Shivers et al., 1999] schlagen als eine weitere Optimierung die „Static Allocation Arena“ vor. Hierbei wird am Anfang jedes vom Compiler erzeugten *basic block* bereits der gesamte benötigte Speicher angelegt. An den ursprünglichen Allokationspunkten werden die einzelnen Objekte lediglich initialisiert. Bei dieser Methode fällt also pro *basic block* nur eine Allokation an.

Eine effiziente Implementierung des Allokations-Algorithmus zeigt sich also die enge Verzahnung mit dem Compiler aufgrund der Plazierung des Codes *inline*. Weiterhin wird offenbar, daß ein Multiprozessorsystem als zu unterstützende Zielplatform eine effiziente Implementierung erheblich erschwert.

4.2 Exakte Garbage Collection für Java

Wie bereits in Abschnitt 2.2.2 beschrieben, gibt es prinzipiell zwei Wege, um die Exaktheit der Garbage Collection zu garantieren: Markierung und Referenztabellen.

Implementierungen von Sprachen, die statisch nicht zwischen Referenz und Skalar unterscheiden, sind für Exaktheit auf die Verwendung von Markierungen angewiesen – Beispiele für solche Sprachen sind Smalltalk, Beta, Self und TL-2. Alle uns bekannten Implementierungen dieser Sprachen verwenden

Markierungen, um Exaktheit zu erreichen.

Statisch getypte, monomorphe Sprachen können ohne Markierungen implementiert werden, wenn der Compiler dem Laufzeitsystem die entsprechenden Informationen zur Verfügung stellt. Handelt es sich um eine rein funktionale Sprache, so können sogar polymorphe Sprachen ohne Markierungen implementiert werden, wie [Appel, 1989a] gezeigt hat.

Java stellt eine statisch getypte, nicht funktionale und bezüglich der Unterscheidung von Skalaren und Referenzen monomorphe Sprache dar, die zur Laufzeit über die notwendigen Informationen verfügt, um exakte Garbage Collection ohne Markierungen realisieren zu können. Dies liegt darin begründet, daß der in [Lindholm and Yellin, 1996] spezifizierte Bytecode Operationen mit Skalaren und Referenzen ausreichend unterscheidet. Außerdem ist die Breite von Zahlenwerten durch die Sprachdefinition [Gosling et al., 1996] vorgegeben – so muß z.B. ein `int` in Java immer eine Breite von 32 Bit besitzen und vorzeichenbehaftet sein.

Diese beiden Gründe legen nahe, die Speicherverwaltung einer Java VM ohne Markierungen zu implementieren. Daraus folgt, daß zu jedem Zeitpunkt, an dem eine Garbage Collection stattfinden kann, Referenztabellen zur Verfügung stehen müssen, die die Typen der aktiven Stackinhalte und lokalen Variablen beschreiben[2].

Wird der Bytecode in Maschinencode übersetzt, so findet zur Übersetzungszeit eine Zuordung der abstrakten lokalen Variablen zum Laufzeitstack oder Registern statt. Diese Zuordung muß vom Compiler in Form von Stack- und Registertabellen abgelegt werden; diese können bei der Garbage Collection verwendet werden, ohne die Integrität der Maschine zu gefährden. Wird der Bytecode interpretiert, so müssen die Tabellen auf eine andere Weise erstellt werden.

Unabhängig vom Ausführungsmodell sollte vermieden werden, für jede Instruktion eine eigene Referenztabelle zu erzeugen, da diese leicht mehr Platz einnehmen können, als der Code selbst. Daher werden vorausberechnete diese in der Regel nur für bestimmte Punkte – sogenannte „GC Points“ oder „Safepoints“ – ermittelt. Die hiermit verbundenen Probleme und deren Lösung werden in Abschnitt 5.3 näher beschrieben.

Eine bemerkenswerte Ausnahme stellt die Java-Implementierung der Intel Microprocessor Research Labs dar [Sichnoth et al., 1999]: Durch geschickte Komprimierung der Referenztabellen wird der Platzbedarf so stark ein-

[2] Aus Sicht des Laufzeitsystems könnte man auch umgekehrt formulieren, daß eine Garbage Collection nur an den Stellen stattfinden darf, für die eine solche Tabelle existiert.

geschränkt, daß für jede generierte Maschineninstruktion eine Tabelle zur Verfügung gestellt werden kann.

Im folgenden werden zwei Methoden beschrieben, um Referenztabellen für interpretierten Java-Bytecode zu erstellen: Das parallele Führen eines Markierung-Stack und die abstrakte Interpretation von Bytecode.

4.2.1 Paralleles Führen eines Markierungs-Stack

Das parallele Führen eines Tagstacks stellt die einfachste Methode dar, Maps zu erhalten. Bei dieser Methode wird für jeden Thread, der in der Maschine ausgeführt wird, ein Markierungs-Stack geführt, der für jede Zelle des Stacks ein Bit enthält. Enthält diese Bit den Wert 1 so handelt es sich bei der entsprechenden Stackzelle um eine Referenz; enthält es den Wert 0, so handelt es sich um einen Skalar. Mit jeder Manipulation des Stacks werden die Markierungen entsprechend angepaßt.

Der Nachteil dieser Methode ist der konstante Mehraufwand, der bei jeder Stackoperation anfällt, um den Markierungs-Stack anzupassen. Demgegenüber steht der geringe Speicherverbrauch: Im Gegensatz zu vorausberechneten Tabellen, von denen in der Regel für jede Methode mehrere vorliegen müssen, wird hier nur ein Stack pro Thread benötigt — aus diesem Grunde wurde diese Technik zur Implementierung der Spotless JVM [Mathiske and Schneider, 2000] verwendet.

4.2.2 Abstrakte Interpretation von Bytecode

Der Java-Bytecode besitzt neben seiner Typisierung eine weiter Eigenschaft, welche die Erstellung von Stackmaps stark vereinfacht: Der Typ einer Variablen ist nicht vom Programmablauf (*control path*) sondern lediglich von der Programmposition (*control point*) abhängig, d.h. er enthält keine *control path dependency*. Im Kontext von Java wird diese Eigenschaft als „Gosling Property“ nach James Gosling bezeichnet. Diese ermöglicht es einem bestimmten Programmpunkt innerhalb einer Methode fest eine Stackmap zuzuordnen, unabhängig davon, über welchen Pfad der betreffende Programmpunkt erreicht wurde.

Die Variablen `i` und `s` in Abbildung 4.6 werden durch den Compiler (hier `javac`) der gleichen Stackzelle 0 zugeordnet. Trotzdem ist der Typ jeder Stackzelle allein vom der aktuellen Bytecode-Position abhängig. Fände zum Beispiel an Instruktion 12 eine Garbage Collection statt, so müßte Zelle 0 als

Skalar behandelt werden, an Instruktion 25 müßte sie als Referenz behandelt werden und an Instruktion 32 gälte sie als ununerreichbar.

Kontrollfluß-unabhängige Eigenschaften von Programmvariablen lassen sich mit Mitteln der Datenflußanalyse berechnen. Dazu wird der Wertebereich der betrachteten Variablen wird auf die Knoten eines Gitters (*lattice*) approximiert. Durch iterative abstrakte Interpretation des Programms lassen sich dann die betrachteten Werte gewinnen [Aho et al., 1986, Muchnick, 1997].

Diese Standardtechnik wird von [Agesen and Detlefs, 1997] und in erweiterter Form in [Agesen et al., 1998] eingesetzt, um die Typen lokaler Variablen in Java zu ermitteln. Das eingesetzte Typgitter ist in Abbildung 4.7 dargestellt. Die folgende Tabelle erklärt die Bedeutung der einzelnen Typen:

top	Der Zelle wurden unterschiedliche Typen zugeordnet. Es ist also ein Konflike aufgetreten.
pc	Die Zelle wurde zum Zwischenspeichern einer Rücksprungadresse in einem `jsr`-Call verwendet.
val	Die Zelle enthält einen Skalar.
ref	Die Zelle enthält eine Referenz.
uninit	Die Zelle ist uninitialisiert.
bottom	Die Zelle wurde noch nicht verwendet.

Für gültigen Java-Bytecode, der die „Gosling Property“ beachtet, ergibt die Flußanalyse für alle Variablen einen Zustand $\neq$ *top*, d.h. es treten keine Typkonflikte auf. Leider gibt es jedoch einen Java-Bytecode, der die „Gosling Property“ nicht einhält: `jsr`. Er wird verwendet, um den `finally`-Block des `try { ... } finally { ... }`-Konstruktes von unterschiedlichen Punkten innerhalb des `try`-Blocks als Subroutine auszuführen. Innerhalb dieser Subroutine der Zustand des Stacks jedoch mehrdeutig sein. Zur Lösung dieses „`jsr`-Problems“ schlagen [Agesen and Detlefs, 1997] ein Umschreiben des Bytecodes vor, indem Variablen mit einem Konflikt in eine Referenz- und eine Skalar-Variable aufgespalten werden. Andere virtuelle Maschinen expandieren die Subroutine einfach an alle Aufrufstellen und lösen so den Konflikt auf. Nach [Freund, 1998] ist der zusätzliche Platzbedarf vernachlässigbar.

4.3 Generationale Garbage Collection

Gerade bei interaktiven Systemen kann sich Garbage Collection mit den vorgestellten Algorithmen als störende Unterbrechung erweisen. Generationalen

```
public static void foo(){
  if("a string".equals("a string")){    0 ldc #2 <String "a string">
                                        2 ldc #2 <String "a string">
                                        4 invokevirtual #9 <Method boolean equals(java.lang.Object)>
                                        7 ifeq 22
      int i = 0;                        10 iconst_0
                                        11 istore_0
      System.out.println(i);            12 getstatic #9 <Field java.io.PrintStream out>
                                        15 iload_0
                                        16 invokevirtual #10 <Method void println(int)>
                                        19 goto 32
  } else {
      String s = "a string";            22 ldc #2 <String "a string">
                                        24 astore_0
      System.out.println(s);            25 getstatic #9 <Field java.io.PrintStream out>
                                        28 aload_0
                                        29 invokevirtual #11 <Method void println(java.lang.String)>
  }
  System.out.println("a string");       32 getstatic #9 <Field java.io.PrintStream out>
                                        35 ldc #1 <String "a string">
                                        37 invokevirtual #11 <Method void println(java.lang.String)>
  return;                               40 return
}
```

Abbildung 4.6: Eine Java-Methode und der von `javac` erzeugte Bytecode.

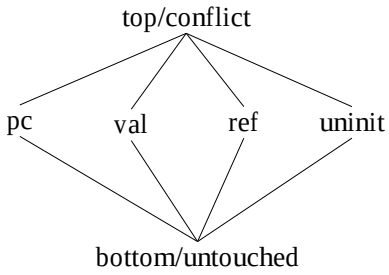

Abbildung 4.7: Typgraph der abstrakten Interpretation von Java-Bytecode nach Agesen

Garbage Collection [Lieberman and Hewitt, 1983, Moon, 1984, Ungar, 1984] ist als Antwort auf dieses Problem entwickelt worden. Sie beruht auf der Beobachtung, daß die meisten erzeugten Objekte in typischen Programmläufen nach relativ kurzer Zeit unerreichbar werden (die Altershypothese: „most objects die young“). Daher lohnt es, die Garbage Collection vor allem auf junge Objekte zu konzentrieren.

Zu diesem Zweck wird der Heap in mehrere Segmente unterteilt, in denen die Objekte nach Alter[3] gruppiert werden. Jedes einzelne Segment wird als Generation bezeichnet und enthält Objekte, die grob der gleichen Altersklasse angehören. Neue Objekte werden in der jüngsten Generation angelegt und, nachdem sie ein gewisses Alter erreicht haben, in die nächste Generation „befördert“.

Um nun eine Generation G_n unabhängig von den anderen bereinigen zu können, müssen die Referenzen von Objekten aus allen anderen Generationen $G_{m \neq n}$ mit in die Wurzelmenge der zu bereinigenden Generation einbezogen werden. Die Information über Zeiger zwischen Generationen (*intergenerational pointers*) muß also bei der Garbage Collection mit geringem Aufwand bereitgestellt werden, d.h. sie muß zur Verfügung stehen, ohne die anderen Generationen zu durchsuchen. Um dieser Anforderung zu genügen, müssen die Operationen, die einen Zeiger in ein Objekt schreiben während des Programmablaufes protokolliert werden. Dies geschieht in der Regel durch das Errichten einer „Write Barrier“ — ein Programmstück in der virtuellen Maschine, das bei jeder Schreiboperation ausgeführt wird und dann die

[3]Das Alter eines Objektes wird im für generationale Algorithmen zumeist als die Anzahl der GC-Zyklen, die ein Objekt überlebt hat, definiert, da sich dieser Wert einfach im Objektkopf speichern und aktualisieren läßt. Meßverfahren interpretieren im Sinne einer besseren Vergleichbarkeit das Alter eines Objekts als die Menge des inzwischen angelegten Speichers.

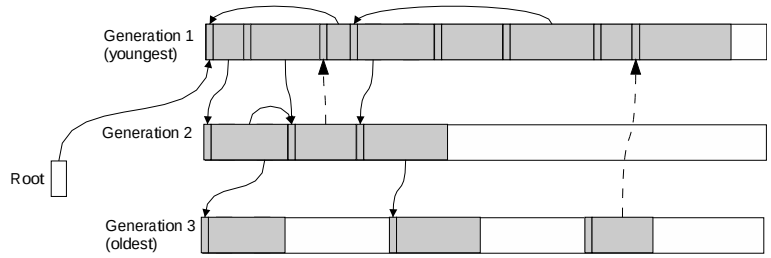

Abbildung 4.8: Generational Collection: Die Referenzen aus der alten Generation müssen berücksichtigt werden.

entsprechende Protokollierung durchführt.

Um den Aufwand möglichst gering zu halten, wird eine weitere Eigenschaft typischer Programme ausgenutzt: Referenzen verweisen überwiegend von jüngeren auf ältere Objekte. Diese Eigenschaft wird ausgenutzt, indem zusammen mit einer Generation G_n auch alle jüngeren Generationen $G_{m<n}$ bereinigt werden. Damit benötigt der Garbage Collector lediglich die Pointer von „alt“ nach „neu“, d.h., daß während des Programmablaufs nur über die selteneren Referenzen von älteren auf jüngere Generationen Buch geführt werden muß. In Abbildung 4.8 sind diese gestrichelt dargestellt.

Ein großer Vorteil dieser Vorgehensweise ist die Tatsache, daß Zeigeroperationen in der jüngsten Generation überhaupt nicht mehr protokolliert werden müssen, da diese ohnehin in jede Garbage Collection mit einbezogen wird. Da die meisten schreibenden Zeigeroperationen auf der jüngsten Generation zu erwarten sind, wird hierdurch ein Großteil der Protokollierung gespart.

Für die Protokollierung der Referenzen von älteren auf jüngere Generationen werden zwei Methoden unterschieden:

„Remembered Sets“ [Ungar, 1984]: Hierbei wird für jede Generation die Menge von Objekten älterer Generationen zusammengefaßt, die in diese Generation verweisen (s. Abb. 4.9). Diese Menge (das sogenannte „Remembered Set“) stellt einen Teil der Wurzelmenge der entsprechenden Generation dar und wird somit bei der Garbage Collection berücksichtigt.

Das „Remembered Set“ einer Generation muß bei jedem Speichern eines Zeigers in diese Generation in ein Objekt einer älteren Generation angepaßt werden. Dies wird durch eine „Write Barrier“ erreicht, deren Aufgabe ist,

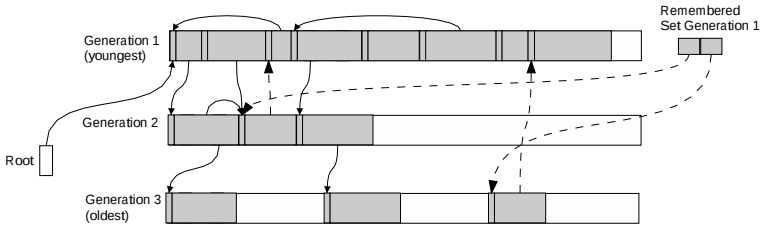

Abbildung 4.9: Remenbered Set.

1. zu prüfen, daß nicht in ein Objekt der jüngsten Generation geschrieben werden soll. Diese Operation braucht wie beschrieben nicht protokolliert zu werden.
2. zu prüfen, daß der zu schreibende Zeiger in eine jüngere Generation verweist.
3. zu prüfen, daß das betroffene Objekt nicht bereits im „Remembered Set“ der fraglichen Generation enthalten ist.
4. das „Remembered Set“ der betroffenen Generation zu aktualisieren, wenn obige Bedingungen zutreffen.

Diese Operationen lassen sich mit 17 MIPS-Assembler-Instruktionen implementieren, was zuviel ist, um die Barriere bei jeder Speicherinstruktion vom Compiler *inline* plazieren zu lassen. Daher schlagen [Hosking et al., 1992] die Verwendung eines sogenannten „Sequential Store Buffer“ vor. Hierbei handelt es sich lediglich um ein Array von Zeiger-Paaren (das Zielobjekt und der gespeicherte Zeiger), zu dem bei jeder Speicheroperation ohne vorherigen Test ein neues Element hinzugefügt wird. Läuft der Puffer über, wird er geleert, indem die entsprechenden Tests ausgeführt und die betroffenen „Remembered Sets“ aktualisiert werden.

Kann der Überlauf des „Sequential Store Buffer“ ähnlich wie bei der Allokation (s. Abschnitt 4.1) durch Verwendung einer geschützten Speicherseite von der Hardware erkannt werden, kann eine reduzierte Assembler-Sequenz von nur drei MIPS-Instruktionen vom Compiler *inline* plaziert werden. Dabei ist zu bedenken, daß die Prüfungen weiterhin ausgeführt werden müssen, sie werden aber nicht mehr *inline* ausgeführt, sondern konzentriert bei einem Überlauf bzw. vor einer Garbage Collection.

„Card Marking“ [Chambers, 1992, Hölzle, 1993]: Bei dieser Methode wird jede Generation in eine Menge von Karten der Größe *card_size*

```
st %dest, [%source + offset]              ; do the store
add %source, offset, %temp                ; compute address of
                                          ; modified word
sra %temp, log_base_2(card_size), %temp ; compute card index
stb %g0, [%card_base + %temp]             ; mark card by zeroing
```

Abbildung 4.10: Die von [Chambers, 1992] verwendete „Write Barrier“

unterteilt, wobei *card_size* eine Potenz von 2 sein muß. Weiterhin wird ein Bit-Array *map* der Länge $\frac{heap\_size}{card\_size}$ angelegt und jedes Element auf 1 initalisiert. Jedes Bit entspricht also einer Karte. Wird nun eine Speicheroperation in der Karte i ausgeführt, so wird durch die „Write Barrier“ ohne weitere Prüfungen das entsprechende Bit auf 0 gesetzt.

Bei der nächsten Garbage Collection wird jedes Objekt in einer so markierten Karte als Teil der Wurzelmenge angesehen, jedoch nur, wenn es tatsächlich in eine jüngere Generation verweist.

Die von [Chambers, 1992] im SELF-System verwendete „Write Barrier“ kann mit nur 3 SPARC-Instruktionen implementiert werden (s. Abb. 4.10), wobei hier aus Effizienzgründen ein Byte-Array anstatt eines Bit-Arrays verwendet wird.

Da die Kartengröße eine Potenz von 2 ist kann die Division zum Ermitteln des Indices effizient durch die Schiebe-Operation `sra` implementiert werden.

Beide geschilderten „Write Barriers“ verlagern einen Teil der zu leistenden Arbeit auf die Garbage Collection, um den Code möglichst kurz zu halten. „Card Marking“ hat gegenüber „Remembered Set“ den Nachteil, daß bei der Garbage Collection immer eine ganze Karte durchsucht werden muß, auch wenn möglicherweise nur ein enthaltenes Objekt einen Zeiger in die jüngere Generation enthält. Weiterhin besteht bei Sprachen ohne Markierung das Problem, den Anfang des ersten Objektes auf einer bestimmten Karte zu finden – beides bedeutet einen weiteren Mehraufwand für die Garbage Collection. Hingegen hängt die Arbeit, die dem Collector durch Pointer-Stores entsteht nicht von der Anzahl der ausgeführten Speicheroperationen, sondern von der Anzahl der markierten Karten ab. Bei wiederholtem Speichern in ein bestimmtes Objekt steigt der Aufwand bei der Verwendung eines „Sequential Store Buffers“ also stetig an, während er bei der Verwendung von „Card Marking“ konstant bleibt.

Generationale Garbage Collection hat sich zu einem De-Facto-Standard bei der Implementierung virtueller Maschinen entwickelt. So verwenden alle für einen produktiven Einsatz vorgesehenen Maschinen in Anhang A einen generationalen Algorithmus mit 2 Generationen. Dabei werden für die Bereinigung der jeweiligen Generationen unterschiedliche Algorithmen gewählt: Üblicherweise wird für die jüngere Generation ein kopierender Algorithmus verwendet, während für die ältere Generation ein Algorithmus verwendet wird, der Objekte mit einer geringeren Wahrscheinlichkeit bewegt (z.B Abwandelungen von „Mark-Sweep“). Die HotSpot VM [Meloan, 1999] verwendet hier einen inkrementellen Algorithmus (siehe Abschnitt 4.4), um auch Pausen für die ältere Generation möglichst kurz zu halten. Weiterhin findet die Allokation vielfach in einem festen Bereich der jüngsten Generation statt, aus dem die Objekte bei ihrer ersten Garbage Collection in den Hauptbereich der jüngsten Generation evakuiert werden. Die Verwendung dieses →*eden* oder →*nursery* genannten Bereichs erhöht die Lokalität bei der Initialisierung von Objekten und ermöglicht effizienteren Code für die Überlaufprüfung, da die Grenze ein konstanter Wert ist.

[Demers et al., 1990] haben eine Implementierung eines generationalen Algorithmus vorgestellt, die Objekte nicht bewegt und sich damit auch zum Erstellen von konservativen und „On-The-Fly“-Algorithmen (siehe Abschnitt 4.4) eignet. Dabei entfällt zwar der positive Effekt der Heap-Kompaktierung und der damit verbundenen erhöhten Lokalität und schnellen Allokation, ein generationaler Ansatz verspricht aber, allein aufgrund des verringerten Working-Sets des Collectors, eine Verkürzung der Programmpausen und der insgesamt aufgewendeten Zeit für Garbage Collection.

Generationale Garbage Collection erreicht bei Erfüllung der Altershypothese deutlich verkürzte Unterbrechungen durch Garbage Collection — [Ungar, 1984] gibt den Faktor 20, [Appel, 1989b] sogar Faktor 50 an. Dabei wird nicht nur die Zeit der meisten Unterbrechungen verringert, sondern auch die Zeit, die insgesamt zur Garbage Collection aufgewendet wird (2% statt 30% bei [Ungar, 1984]). Wird die Hypothese jedoch nicht erfüllt, so können häufige Bereinigungen des gesamten Speichers auftreten. [Ungar and Jackson, 1991] haben beobachtet, daß auch in normalen Programmabläufen übergangsweise Zustände eintreten können, in denen die Altershypothese nicht erfüllt ist. Dies hat zur Folge, daß häufiger alle Generationen im System müssen bereinigt werden, was sogar zu einer Verschlechterung der Performance gegenüber einem nicht-generationalen Algorithmus führen kann.

4.4 Inkrementelle und nebenläufige Garbage Collection

Die erwähnten Algorithmen für Garbage Collection werden üblicherweise aufgerufen, wenn die Anwendung nicht mehr über ausreichend freien Speicher verfügt. Hierauf wird eine Garbage Collection des gesamten Heaps beziehungsweise einer oder mehrerer Generationen ausgeführt.

Um die hierbei entstehenden Pausen möglichst weiter zu verringern, werden zwei verwandte Ansätze verfolgt:

Inkrementelle Algorithmen: Die Garbage Collection wird schrittweise mit der Programmausführung verzahnt. Dabei wird üblicherweise bei jeder Allokation ein kleiner Bereich des Heaps von der Garbage Collection bearbeitet. Anschließend wird die Kontrolle wieder an die Anwendung abgegeben. Beim der nächsten Allokation wird ein weiterer Bereich bearbeitet. Da diese kleineren Teile schneller bereinigt werden können als der gesamte Heap, sind auch die entstehenden Pausen kürzer.

Nebenläufige Collectoren: Die Garbage Collection läuft in einem eigenen Thread nebenläufig mit der Anwendung ab.

Bei inkrementellen Algorithmen handelt es sich also, wie bei allen bisher geschilderten Algorithmen um einen „Stop-the-World“-Algorithmus, da alle ablaufenden Threads angehalten werden müssen, und an einen konsistenten Punkt vorgerollt werden müssen (s. Abschnitt 5.3), bevor die Garbage Collection durchgeführt werden kann. Der Algorithmus muß allerdings „verkraften“, daß der Heap zwischen den Teil-Abläufen durch die Anwendung verändert wird – es muß also eine Form der Kommunikation und Synchronisation zwischen Anwendung und Garbage Collector stattfinden.

Bei nebenläufigen Algorithmen ist eine noch feinere Synchronisation zwischen Anwedung und Garbage Collector vonnöten. Da Garbage Collector und Anwendung keine Kontrolle darüber haben, wann sie unterbrochen werden, muß praktisch jede Operation den Heap in einem konsistenten Zustand hinterlassen (zumindest solange die nebenläufige Collection läuft). Weiterhin darf die Anwendung in der Ausführung nicht übermäßig behindert werden – Garbage Collection sollte bescheiden (*unobstrusive*) sein.

Im Bereich der inkrementellen und nebenläufigen Algorithmen wird die von [Dijkstra et al., 1978] formalisierte Abstraktion des „tricolor marking“ häufig

verwendet. Hierbei werden die bearbeiteten und noch zu bearbeitenden Objekte in drei Klassen (bzw. Farben) unterteilt.

schwarz: Ist ein Objekt schwarz gefärbt, so bedeute dies, daß sowohl das Objekt selbst, als auch alle seine direkten Kinder durch den Garbage Collector besucht und als erreichbar erkannt wurden. Schwarze Objekte sind abschließend bearbeitet und müssen nicht mehr besucht werden.

grau: Ein graues Objekt wurde bereits vom Garbage Collector besucht, es existieren aber möglicherweise Kinder, die noch nicht besucht wurden. Graue Objekte müssen also noch einmal besucht werden.

weiß: Diese Objekte wurden vom Garbage Collector (noch) nicht besucht. Ist ein Objekt am Ende einer Garbage Collection weiß, so ist es unerreichbar.

Diese Abstraktion läßt sich auf die klassischen Algorithmen anwenden. Bei „Copying Collection“ sind die schwarzen Objekte jene, die bereits evakuiert und deren Referenzen angepaßt wurden, also jene, die im „To-Space“ unterhalb des Pointers `scan` liegen (s. Abbildung 2.6-(iv)). Die Objekte, die in „From-Space“ liegen und noch nicht evakuiert wurden, sind weiß, während die Objekte in „To-Space“, die sich zwischen `scan` und `free` befinden, grau sind.

Im Fall eines „Mark-Sweep“-Algorithmus sind die markierten Objekte schwarz, die unmarkierten weiß und die Objekte, die sich auf dem Mark-Stack befinden, grau.

Aus der gegebenen Definition der Farbmengen läßt sich eine wichtige Invariante ableiten:

Ein schwarzes Objekt darf niemals direkt auf ein weißes Objekt verweisen.

Bei Verletzung dieser Farbinvarianten können fälschlicherweise lebendige Objekte collected werden. In Abbildung 4.11 ist eine Folge von Objektgraphen dargestellt, die dieses Problem illustriert. In (i) hat der Collector Objekt A vollständig besucht und befindet sich gerade in der Bearbeitung von B. Hier wird die Collection unterbrochen und der Mutator verändert den Objektgraphen zu (ii). Wenn der Collector wieder aufgerufen wird, wird er B schwarz färben, da es keine weißen Kinder mehr hat und dort die Bearbeitung beenden. Im Ergebnis wird C, obwohl es lebendig ist, collected werden, während B überlebt.

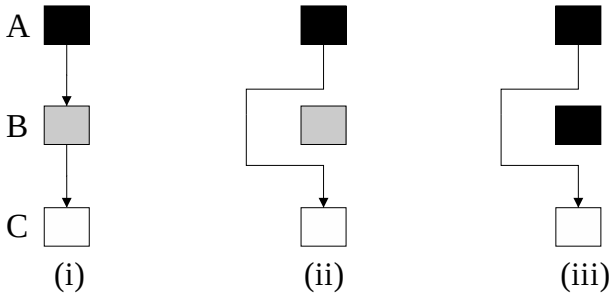

Abbildung 4.11: Tricolor Marking: Fehler bei Verletzung der Farbinvarianten.

Die Farbinvariante wird von den klassischen Algorithmen nicht verletzt, ein inkrementeller oder nebenläufiger Algorithmus muß jedoch gesondert Sorge dafür tragen, daß die Invariante eingehalten wird. Dies liegt darin begründet, daß der der Heap während der Garbage Collection vom der Anwendung verändert werden kann.

Um auf eine inkonsistente Veränderung des Heaps „hinter dem Rücken“ des Garbage Collectors zu reagieren, werden zwei unterschiedliche Ansätze verwendet, die die Einhaltung Farbinvarianten garantieren: man unterscheidet sie je nachdem, ob sie auf „Read Barriers“ oder „Write Barriers“ basieren.

„Read Barrier“-Methoden erhalten die Farbinvariante, indem die vermeiden, daß die Anwendung jemals weiße Objekte „sieht“. Dies geschieht, indem bei jedem Lesen einer Referenz aus dem Speicher geprüft wird, ob diese auf ein weißes Objekt verweist. Ist dies der Fall, so wird das Objekt durch eine Evakuierung grau gefärbt. [Baker, 1978] führt die „Read Barrier“ in Software aus und arbeitet dabei auf Objektebene – jeder lesende Zeigerzugriff wird getestet und das Objekt ggf. nach „To-Space“ bewegt. [Appel et al., 1988] verwenden den Speicherschutzmechanismus des Betriebssystems und arbeitet damit seitenorientiert: Alle Seiten, die weiße Objekte enthalten, werden geschützt. Die zugehörige Ausnahmeroutine evakuiert die gesamte Seite, wenn die Anwendung versucht, von einer geschützten Seite zu lesen.

„Write Barrier“-Methoden prüfen bei jeder Schreiboperation, ob die Farbinvariante verletzt wird und markieren ggf. ein beteiligtes Objekt grau, um direkte Schwarz-Weiß-Referenzen zu vermeiden. Der in [Dijkstra et al., 1978] vorgestellte Algorithmus der „On-The-Fly Garbage Collection“ arbeitet hier wie [Baker, 1978] auf der Granularität von Objekten: Versucht die Anwendung eine Referenz auf ein weißes Objekt in ein schwarzes Objekt zu schreiben, so wird ersteres durch die in Software realisierte „Write Barrier“ grau gefärbt. [Boehm et al., 1991] verwenden wie [Appel et al., 1988] den Spei-

cherschutzmechanismus des Betriebssystems und arbeiten somit seitenorientiert. Schreibende Zugriffe auf Seiten, die nur schwarze Objekte enthalten, werden abgefangen und alle Objekt der beschriebenen Seite werden als grau markiert.

„Read Barrier"-Algorithmen verursachen einen ungleich größeren Mehraufwand zur Laufzeit als eine Write Barrier, da lesende Operation deutlich häufiger auftreten als schreibende Allerdings treten bei Verwendung von „Write Barrier"-Methoden in Verbindung mit Algorithmen, die Objekte bewegen, zusätzliche Synchronisationsprobleme auf. Zur Implementierung eines nebenläufigen, bewegenden Algorithmus existiert daher augenblicklich keine Alternative zur „Read Barrier".

Die Ausnutzung der virtuellen Speicherverwaltung zur Implementierung einer Barrier ist prinzipiell einen elegante Lösung, da diese ohne Mehrkosten für den schnellen Ausführungspfad realisiert wird. Allerdings sind heutige Betriebssysteme nicht auf die intensive Verwendung von Ausnahmen im normalen Programmablauf optimiert, so daß die Verarbeitung einer Ausnahme üblicherweise mehrere 100 CPU-Zyklen in Anspruch nimmt.

Ein vergleichsweise neuer inkrementeller Algorithmus, der zunehmend bei der Implementierung von Java-VMs eingesetzt wird, ist der „Train"-Algorithmus [Hudson and Moss, 1992]. Er integriert einen „Write-Barrier"-basierten inkrementellen Algorithmus mit einem generationalen Garbage Collector.

Bei Verwendung des „Train"-Algorithmus wird lediglich die älteste Generation, der sogenannte „mature object space" (MOS), inkrementell bereinigt. Zu diesem Zweck wird er in kleine Blöcke fester Größe (sog. Wagen, *cars*) unterteilt, die wiederum aufgrund der Verweise untereinander zu Zügen (*trains*) zusammengefaßt werden. Jeder Wagen verfügt über ein eigenes „Remembered Set" für Verweise aus anderen Wagen und kann somit zusammen mit den jüngeren Generationen, aber unabhängig von anderen Wagen bereinigt werden.

Der „Train"-Algorithmus ist nicht zuletzt durch die gut beschriebene Implementierung von [Seligmann and Grarup, 1995] eine gut verstandene Methode zur Bereinigung der alten Generation. Sie wird daher zunehmend bei der Implementierung von Java-VMs eingesetzt, um die Pausen, die durch Garbage Collection des gesamten Speichers entstehen, zu reduzieren.

4.5 Garbage Collection auf Multiprozessoren

Eine Multiprozessorumgebung unterscheidet sich aus Sicht des Garbage Collectors von einer Uniprozessorumgebung und stellt somit besondere Anforderungen an die Implementierung des Collectors. Dabei sind die folgenden 3 Ausgangsprobleme zu beachten:

1. Die Ausnutzung der vorhandenen Prozessoren ist in einer Multiprozessorumgebung bei klassischen seriellen „Stop-the-World"-Algorithmen nicht optimal, da während der Garbage Collection nur ein Prozessor beschäftigt ist. Mit steigender Zahl der Prozessoren steigen also die relativen Kosten der Garbage Collection.

2. Atomare Synchronisationsoperationen werden mit wachsender Prozessorzahl teurer. Auf einer Uniprozessormaschine werden durch ein „Compare-And-Swap" (`CAS`) lediglich andere Geräte, die den Speicherbus verwenden, blockiert. Im Falle einer Mehrprozessormaschine wird der Speicherbus für einen kurzen Moment für alle Prozessoren blockiert. Die Verwendung derartiger globaler Sychronisationsoperationen sollte deshalb auf ein Minimum reduziert werden.

3. Jeder Prozessor verfügt über einen Satz lokaler Register. Diese könne von einem auf einem anderen Prozessor ablaufenden Algorithmus nicht eingesehen oder verändert werden. Weiterhin entspricht das Speichermodell in einem Multiprozessorsystem nicht dem der *sequential consistency* eines Uniprozessorsystems (siehe Abschnitt 2.3.2).

Abbildung 4.5 illustriert Ausgangsproblem 1 am Beispiel einer Umgebung mit vier Prozessoren. In der Teilabbildung (i) findet eine klassische „Stop-the-World" Collection statt – entsprechend wird während der Garbage Collection nur ein Prozessor verwendet, während die verbleibenden Prozessoren ungenutzt bleiben[4]. Das im vorherigen Abschnitt dargestellte inkrementelle Verfahren (Teilabbildung (ii)) verkürzt zwar die Pausen, die durch Garbage Collection entstehen, die Auslastung der Prozessoren verbessert sich jedoch nicht.

Um einen Garbage Collector zu implementieren, der die obigen Probleme berücksichtigt bieten sich prinzipiell zwei orthogonale Ansätze an[5]:

[4]Das hier zugrundeliegende Anwendungsmodell könnte beispielsweise das eines dedizierten Anwendungsservers sein, der mehrere Applikationen in einer virtuellen Maschine ausführt.

[5]Man beachte, daß die hier verwendete Terminologie für Garbage Collection gebräuchlich ist, jedoch nicht mit der aus Abschnit 2.3 korrespondiert. Während im Allgemeinen

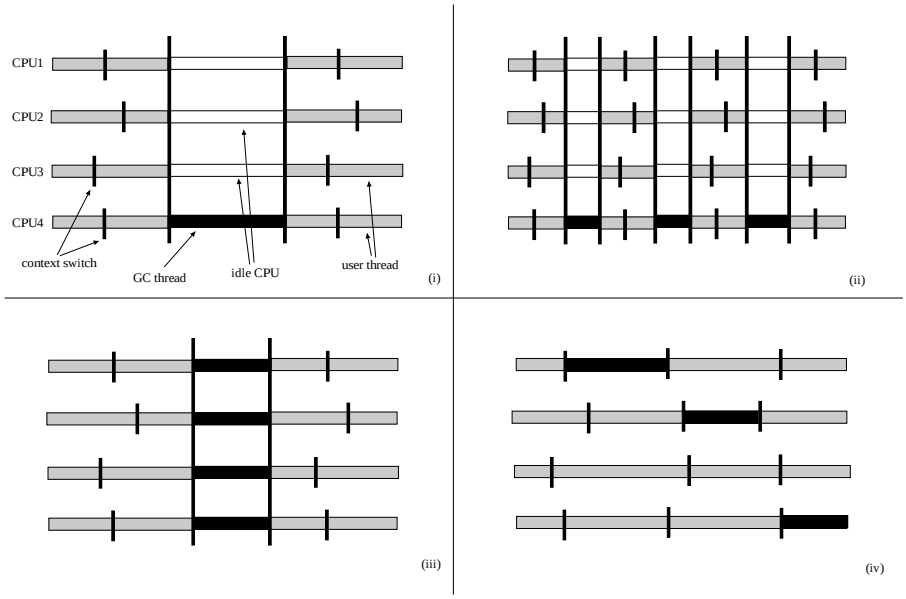

Abbildung 4.12: Garbage Collection in einer Multi-Prozessor-Umgebung: (i) klassische „Stop-the-World“ GC (ii) Inkrementelle GC (iii) Parallele GC (iv) Nebenläufige GC

Parallele Garbage Collection: Hierbei wird die Arbeit des Garbage Collectors selbst auf mehrere Prozessoren verteilt (siehe Abbildung 4.5-(iii)). Es wird also eine optimalere Ressourcenausnutzung während der Garbage Collection erreicht – es wird aber weiterhin um ein „Stop-the-World“-Ansatz verfolgt.

Nebenläufige, multiprozessorfähige Garbage Collection: Wie bei einem nebenläufigen Verfahren aus Abschnitt 4.4 laufen Anwendung und Garbage Collector in unterschiedlichen Threads ab. Wie in Abbildung 4.5-(iv) zu sehen, wird der Garbage Collector vom Scheduler wie ein Anwendungs-Thread unterbrochen und ggf. auf einer anderen CPU fortgesetzt. Es werden also zu keinem Zeitpunkt alle Threads angehalten.

Bei der parallelen Garbage Collection handelt es sich also um die Parallelisierung des sonst sequentiellen Vorgangs der Garbage Collection. Dabei ist zu

zwischen potentieller und tatsächlicher Gleichzeitigkeit unterschieden wird, verwendet man die Begriffe in diesem Kontext, um zwischen Nebenläufigkeit in sich bzw. mit der Anwendung zu differenzieren.

beachten, daß eine effiziente dynamische Lastverteilung zwischen den arbeitenden Threads vorgenommen werden muß. Diese Lastverteilung erfordert natürlich Kommunikation zwischen den beteiligten Threads, wobei die hierzu notwendige Synchronisierung aber wegen Ausgangsproblem 2 möglichst gering gehalten werden muß. Weiterhin muß zuverlässig und effizient erkannt werden, wann die Garbage Collection terminiert.

Die naive Implementierung eines parallelen „Mark-Sweep"-Algorithmus – also eines nicht-bewegenden Verfahrens – ist von vergleichsweise geringer Komplexität. Probleme treten hier überwiegend bei der effizienten Implementierung der Lastverteilung, der Enderkennung und der Verwaltung der Freispeicherliste auf.

[Endo et al., 1997] verwenden zur Lastverteilung sogenannte „Stealable Mark Queues" – jeder andere Prozessor kann Objekte aus der aus der Markierungsliste eines anderen Prozessors stehlen, wenn er keine Arbeit mehr hat.

Die Nachteile eines „Mark-Sweep"-Algorithmus sind bekannt, weshalb ein paralleles kompaktierendes Verfahren wünschenswert wäre. Dieses ist aufwendiger zu realisieren, da verhindert werden muß, daß ein Objekt von mehreren Prozessoren „gleichzeitig" bewegt wird.

[Kolodner and Petrank, 1999] versuchen, diese Probleme mit Hilfe einer parallel implementierten „Copying Collection" zu lösen. Da hier jedoch Objekte bewegt werden, zieht dies einen hohen Synchronisierunsaufwand nach sich. Dieser wird gemindert, indem Speicher in „To-Space" blockweise angelegt wird: eine synchronisierte Allokationsoperation wird daher nicht für jedes evakuierte Objekt ausgeführt. Unmittelbar bei der Evakuierung eines Objektes ist beim Eintragen des Vorwärtsverweises in „From-Space" jedoch `CAS` oder eine vergleichbare Operation notwendig.

Die Threads des Garbage Collectors kommunizieren über verschiedene Listen miteinander, um eine Lastverteilung zu erzielen. Eine Lastverteilung ist essentiell: [Endo et al., 1997] vermelden, daß eine naive Verteilung der Wurzelreferenzen auf 64 Prozessoren lediglich eine Beschleunigung der Collection um den Faktor 3 ergeben hat. Die beschriebene Lastverteilung über „Stealable Mark Queues" erreicht hingegen eine Beschleunigung um den Faktor 28.

Die vielversprechenden Algorithmen im Bereich der nebenläufigen Garbage Collection für Multiprozessorsysteme beruhen überwiegend auf Dijkstras „On-The-Fly"-Verfahren und gehören damit zur Klasse der „Mark-Sweep"-Algorithmen ohne Heap-Kompaktierung. Ein nebenläufiger und kompaktierender Algorithmus, der als bescheiden zu bezeichnen wäre, scheint schwer realisierbar, da im Prinzip **jede** lesende oder schreibende Operation der An-

wendung mit dem Garbage Collector synchronisiert werden müßte. Dies ist wegen Ausgangsproblem 2 inakzeptabel ist.

[Doligez and Leroy, 1993] gelingt es durch die Ausnutzung einer Spracheigeschaft von ML, einen generationalen und nebenläufigen Garbage Collector zu implementieren. Dabei erhält jeder Thread einen eigenen Heap, auf dem ausschließlich unveränderliche Objekte abgelegt werden – dieser Heap stellt die jüngste Generation dar. Weiterhin existiert ein globaler Heap, auf dem veränderliche Objekte sowie Kopien von Objekten, die eine Garbage Collection überstanden haben, abgelegt werden. Referenzen von der alten in einen junge Generation läßt der Algorithmus nicht zu – stattdessen wird das zugewiesene Objekt mitsamt seiner transitiven referentiellen Hülle in die alte Generation kopiert. Diese wird von einem nebenläufigen Garbage Collector bearbeitet. Der Algorithmus ist darauf angewiesen, daß der Compiler veränderbare und unveränderliche Objekte unterscheiden kann – dies ist in funktionalen Sprachen möglich, für die Implementierung einer objektorientierten Sprache eignet sich der Algorithmus jedoch nicht.

Der Algorithmus von [Doligez and Gonthier, 1994] stellt eine Verallgemeinerung des obigen Algorithmus dar, denn die Trennung von veränderlichen und unveränderlichen Objekten entfällt – damit aber auch die Generationen. Objekt-Allokation findet von vorneherein in dem globalen Speicher statt. Somit handelt es sich also um eine multiprozessorfähige Erweiterung von Dijkstras „On-the-Fly"-Algorithmus. Zu keinem Zeitpunkt werden alle Threads auf einmal angehalten – statt dessen synchronisiert der Garbage Collection sich mit jeweils mit einzelnen Threads. Sobald diese Synchronisation mit allen Threads durchgeführt worden ist, sprich wenn alle Wurzelobjekte markiert wurden, führt der Garbage Collector nebenläufig den „Mark-Sweep"-Algorithmus durch. In dieser Phase habe Anwendungs-Threads dem Garbage Collector durchgeführte Veränderungen am Speicher mitzuteilen. Dies kann jedoch ohne jegliche Synchronisation *inline* in einer „Write Barrier" ausgeführt werden. Außerhalb der Garbage Collection braucht die Anwendung keine zusätzlichen Operationen auszuführen.

[Domany et al., 2000] haben diesen Algorithmus für Java implementiert. Dabei wird die Technik aus [Demers et al., 1990] verwendet, um Generationen zu implementieren, ohne Objekte zu bewegen. Es handelt sich also um ein multiprozessorfähiges „On-the-Fly"-Verfahren mit Generationen und ohne Kompaktierung. Ihre Messungen beziehen sich allein auf den Geschwindigkeitsgewinn, der durch die Verwendung von Generationen erzielt wird (zwischen 4% Verlust und 25% Gewinn) – ein Vergleich mit einem herkömmlichen „Stop-the-World"-Algorithmus wurde jedoch nicht gemacht.

Die Ansätze der parallelen und nebenläufigen Collection sind prinzipiell kombinierbar, d.h. ein paralleler *und* nebenläufiger Collector ist durchaus denkbar. Da in üblichen Systemen die Garbage Collection einen Anteil von 2-5 % an der Gesamtlaufzeit des Systems einnimmt, scheint eine Parallelisierung einer nebenläufigen Collection in Systemen mit mehr als 20 Prozessoren wünschenswert. Ohne Parallelisierung könnte der Garbage Collector nicht mit der Anwendung schritthalten, und es würden Pausen entstehen.

Prinzipilell läßt sich feststellen, daß die Techniken der nebenläufigen Garbage Collection insbesondere aufgrund fehlender Kompaktierung noch nicht ausgereift genug sind, um an die Leistungsfähigkeit aktuell verwendeter generationaler Algorithmen heranzureichen. Hingegen scheint sich die Technik der parallelen Garbage Collection gut in einen herkömmlichen Kontext einbinden zu lassen und dürfte damit schneller in modernen multiprozessorfähigen virtuellen Maschinen zu finden sein.

4.6 Zusammenfassung

In diesem Kapitel wurde der aktuelle Stand der Technik zur Implementierung der Speicherverwaltung von Java-Maschinen dargestellt. Dabei wurden folgende Punkte herausgearbeitet:

- Kann der Garbage Collector eine Speicherkompaktierung zumindest auf den jüngsten Generation ausführen, so kann sehr effizienter Allokationscode eingesetzt werden, der gut vom Compiler *inline* plaziert werden kann. Zur Implementierung der Speicherverwaltung einer virtuellen Maschine, die aktuellen Performance-Anforderungen genügt, ist Kompaktierung also unerlässlich.

- Soll der Speicher kompaktiert werden, so muß der Algorithmus exakt sein, was zumindest für Sprachen ohne Markierung einen erheblichen Aufwand bedeutet – Java erschwert dies noch weiter durch Schwächen im Bytecode-Format. Dennoch muß dieser Aufwand im Sinne einer schnellen Allokation in Kauf genommen werden.

- Generationale Garbage Collection ist der de-facto Standard moderner virtueller Maschinen. Durch Konzentration auf junge Objekte gelingt es, nicht nur die Pausen zu verkürzen, sondern auch die insgesamt für Garbage Collection aufgewendete Zeit zu verkürzen.

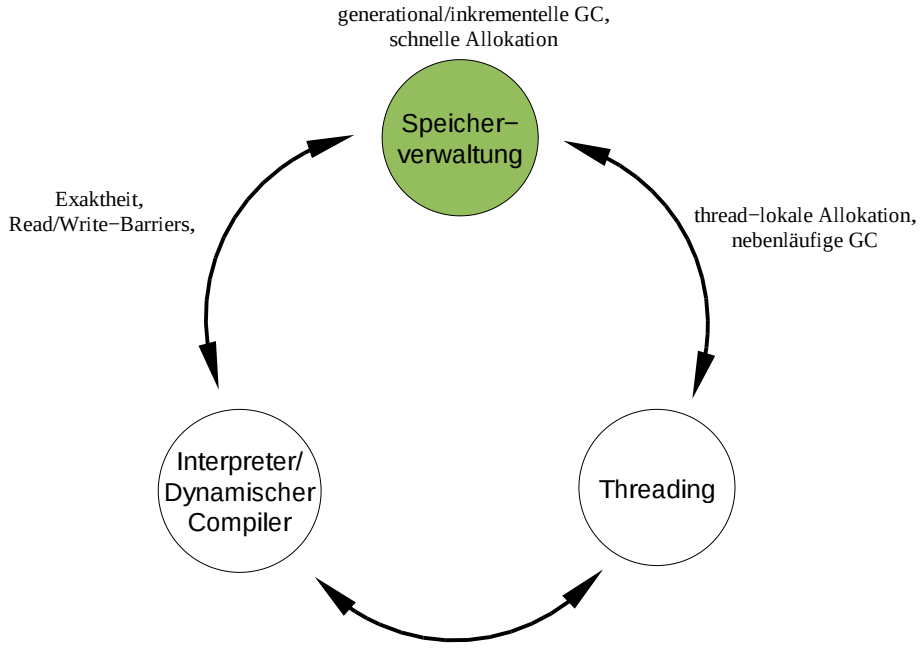

Abbildung 4.13: Die drei Hauptkomponenten aus Sicht der Speicherverwaltung

- Um Zeiger von älteren auf jüngere Generationen zu protokollieren, sind mehrere effiziente Verfahren bekannt, die alle auf „Write Barriers“ basieren. Beide produzieren eine sehr kurze Codesequenz, die der Compiler *inline* plazieren kann.

Weiterhin wurde die enge Integration zwischen der Speicherverwaltung und den beiden anderen Hauptkomponenten dynamischer Compiler und Thread-System offenbar: Sowohl bei der Allokation als auch beim Protokollieren von Referenzen zwischen Generationen muß die Speicherverwaltung mit dem Compiler zusammenarbeiten. Bei der Allokation und bei der Einleitung der Garbage Collection ist eine Kooperation mit dem Thread-System vonnöten.

Zur Beschleunigung der auch bei generationaler Garbage Collection notwendigen Bereinigen des gesamten Speichers bieten sich inkrementelle Techniken an. Die Sun HotSpot Maschine [Sun Microsystems, 1999] implementiert als erste Java-VM den „Train“-Algorithmus – inzwischen folgen verschieden andere Implementierungen [Appeal, 2000, Intel, 2000], so daß sich auch hier ein Standard herauszubilden scheint.

Als weiterer Punkt wurde die Implementierung von Garbage Collection für Multiprozessorumgebungen untersucht. Klassische Algorithmen nutzen die

Ressourcen hier nicht optimal aus, da sie sequentiell arbeiten und die Anwendung währenddessen ausschließen. Daher werden in diesem Bereich parallele Ansätze verfolgt, die die Arbeit des Garbage Collectors im Sinne einer Lastverteilung auf mehrere Prozessoren verteilt. Weiterhin existieren nebenläufige Verfahren, in denen der Garbage Collector nebenläufig mit der Anwendung ausgeführt wird. Diese Verfahren sind zwar schon lange bekannt, es existiern aber noch keine Implementierungen, die in ihrer Leistungsfähigkeit mit den gut erforschten Algorithmen der generationalen Garbage Collection vergleichbar wären.

Abschließend läßt sich festhalten, daß die meisten der in aktuellen Java-VMs verwendeten Techniken zur Implementierung der Speicherverwaltung gut erforscht sind. Lediglich im Bereich der Garbage Collection für Multiprozessorumgebungen sind in den nächsten 1–2 Jahren größere Fortschritte zu erwarten.

Kapitel 5

Implementierung von Nebenläufigkeit

„Better Living through Concurrency“

Dave Butenhof

Die effiziente Implementierung einer Java Virtual Machine für Server-Software erfordert die Ausnutzung mehrerer Prozessoren bei gleichzeitiger Unterstützung einer hohen Zahl von Threads. Diese zwei Entwurfsziele in Einklang zu bringen, verkompliziert das Laufzeitsystem in erheblichem Maße. Kernentscheidungen betreffen den Mechanismus und die Granularität von Thread-Wechseln, den Abbildungsmechanismus auf Systemkontexte, den Umgang mit blockierenden Systemaufrufen sowie Globaloperationen wie das Anhalten aller Threads („stop-the-world“). Es ist zu prüfen, inwieweit existierende Betriebssystembibliotheken den gestellten Ansprüchen gerecht werden.

Der Ubiquität von Monitoren und konfliktfreien Synchronisierungsoperationen muß Rechnung getragen werden. Insbesondere sind die Kosten für nicht verwendeter oder konfliktfreie Monitore ist so gering wie möglich zu halten. Auch hier sind die Eigenschaften der Systembibliotheken in Betracht zu ziehen.

Das Laufzeitsystem mit Speicherverwaltung und Codegenerierung muß unter Berücksichtigung nebenläufiger Ausführung entworfen und implementiert werden, d.h. jeder Zugriff auf Laufzeitstrukturen wie die Klassenhierarchie,

die Speicherorganisation und die Objektmetainformationen muß in threadsicherer Manier erfolgen.

Im Folgenden werden Thread-Implementierungen und Synchronisierungsalgorithmen separat behandelt. Auf eventuelle Abhängigkeiten wird entsprechend hingewiesen. Nach einer kurzen Einführung in die verwendete Terminologie wird zunächst eine grobe Übersicht gängiger Implementierungsklassen und ihrer Eigenschaften gegeben und anschließend detailliert auf einzelne Systeme eingegangen.

5.1 Terminologie

Im Kontext der Sprachimplementierung ist der Thread-Begriff vielfach überladen:

- Die Programmiersprache stellt eine Thread-Abstraktion zur Verfügung, z.B. Java-Threads. Diese werden im folgenden als Programmiersprachen-Threads bezeichnet.
- Das Betriebssystem bietet oftmals Schnittstellen und Bibliotheken zur Thread-Programmierung, z.B. POSIX Threads. Diese werden folgend als Betriebssystemthreads bezeichnet.
- Der Betriebssystemkern verwaltet Threads als Ausführungskontexte, die auf die Ressource „Prozessoren“ zu verteilen sind. Nachstehend wird der gängiger Terminologie folgend der Begriff des leichtgewichtigen Prozesses (*lightweight process*, LWP) verwendet.

Eine ähnliche Situation ergibt sich für Synchronisationsmechanismen. Konzepte gegenseitigen Ausschlusses finden sich auf allen drei Schichten. Auch hier wird zumeist ein entsprechender Präfix zur Disambiguierung verwendet, also Programmiersprachen-, Betriebssystem- und Kern-Sperren.

Die Nebenläufigkeitskonzepte der einzelnen Schichten können auf vielfältige Weise aufeinander abgebildet werden, um Nebenläufigkeit in einer Programmiersprache zu implementieren. Teilweise kann auch ganz auf die Verwendung von Funktionalität des Betriebssytems oder des Betriebssytemskerns verzichtetet werden. Die verschiedenen Ansätze werden im Folgenden dargelegt.

5.2 Threads

Aufgabe einer Thread-Implementierung ist die Bereitstellung mehrerer unabhängiger Ablaufstränge, die einen eigenen Aufruf-Stack und damit eigene lokale Variablen und Instruktionszeiger besitzen. Werden keine Threads einer niedrigeren Schicht genutzt oder werden Threads der oberen Schicht auf eine kleinere Anzahl von Threads einer unteren Schicht abgebildet, so muß ein Mechanismus zur Zuteilung und Entzug von Rechenzeit — ein sogenannter Scheduler — realisiert werden.

5.2.1 Threads auf Anwendungsebene

Werden Threads in der VM allein auf Anwendungsebene realisiert, ohne dabei auf Funktionalität des Betriebssystems zurückzugreifen, so spricht man von Implementierungen im *user space*. Erzeugung, Zuteilung von Rechenzeit und Synchronisation der Threads finden statt, ohne daß entsprechende Betriebssystemfunktionen verwendet werden. Für das Betriebssystem stellt sich die VM als ein nicht nebenläufiger Prozeß dar. Implementierungen im *user space* werden auch als „n:1-Implementierungen“ bezeichnet, da hier alle Programmiersprachen-Threads auf einen Betriebssystem-Thread abgebildet werden.

Dies kann aus verschiedenen Gründen geschehen:

Verfügbarkeit Eine nebenläufige Programmiersprache soll auf einem System implementiert werden, das die Ausführung mehrerer Threads nicht unterstützt. So existieren Java-Implementierungen für PalmOS und Windows 3.1, obwohl keine dieser beide Plattformen über ein Thread-Konzept verfügt.

Effizienz Kontextwechsel zwischen zwei Threads erfordern keinen Aufruf des Betriebssystem-Schedulers. Dieser Übergang vom *user space* in den *kernel space* ist aufwendig und belastet die Effizienz des Systems erheblich. [Anderson et al., 1991] nennen ein bis zwei Größenordnungen Zeitdifferenz zwischen Kontextwechseln auf Anwendungsebene und auf Systemkernebene.

Flexibilität Durch eine eigene Implementierung kann die VM Thread-Funktionalitäten realisieren, die vom Betriebssystem nicht zur Verfügung gestellt werden. Beispielsweise spezifiziert der Smalltalk

Sprachstandard präzise einen prioritätsbasierten Scheduler, der von der VM entsprechend umgesetzt werden muß.

Portabilität Je weniger Betriebssystemschnittstellen in einer Implementierung genutzt werden, desto einfacher kann diese auf ein anderes System übertragen werden. Insbesondere die Standardisierung von Thread-Schnittstellen ist erst vor kurzem entscheidend vorangeschritten — Implementierungen des POSIX Standards 1003.1c–1996 („POSIX Threads“) [POSIX, 1996] sind erst seit ca. 1998 ausreichend auf UNIX-Systemen verfügbar. Die Java-Referenzimplementierung Classic VM[Sun Microsystems, 2000a], die Sun seinen Lizenznehmern zur Verfügung stellt, enthält daher eine Thread-Implementierung im *user space* („green threads“).

Einfache Implementierung In einer Thread-Implementierung im *user space* wird der Scheduler nur explizit aufgerufen oder läßt sich in der Regel ausschalten. Teile der Maschinenimplementierung brauchen daher nicht thread-sicher implementiert zu werden — ein nicht zu unterschätzender Vorteil.

Implementierungen um *user space* lassen sich weiter nach ihrem Unterbrechungsmechanismus klassifizieren (*preemption*):

Freiwillige Prozessoraufgabe (*voluntary yield*). Der Thread behält solange die Kontrolle, bis der Anwendungscode einen expliziten Aufruf an den Scheduler tätigt. Dieser kann dann den Thread-Zustand sichern und einen Thread-Wechsel vornehmen.

Diese Technik erfordert natürlich entsprechend strukturierte Anwendungsprogramme. Für eine virtuelle Maschine ist dies jedoch unproblematisch, denn der „Anwendungscode“ ist in diesem Fall der Interpreter oder der vom Compiler generierte Code und kann damit automatisch entsprechend instrumentiert werden.

Nur beim Aufruf externen Codes besteht die Gefahr, daß ein Thread über Gebühr lange den Prozessor belegt oder gar durch Warten auf ein externes Ereignis die Gesamtheit der Threads blockiert. Steht eine Systemthread-Schnittstelle zur Verfügung, so können für diese Fälle neue Betriebssytem-Threads oder LWP gestartet werden. Dieser Ansatz führt hin zu Two-Level-Implementierungen, die wir in 5.2.3 beschreiben.

Es reicht aus, zwei Operationen zu möglichen Unterbrechungspunkten zu machen:

- Methodenaufrufe bzw. Methodenprologe
- Rückwärtssprünge in Schleifen

Codeblöcke zwischen zwei solchen Punkten werden linear durchlaufen und sind daher im Allgemeinen ausreichend kurz, um häufige Thread-Wechsel zu ermöglichen.

Zeitgesteuerte Unterbrechung Durch einen Betriebssystemmechanismus (z.B. UNIX Signale) können regelmäßig Unterbrechungen des Benutzerprogramms verursacht und die Kontrolle an den Scheduler übergeben werden. Dieser Mechanismus funktionert im Gegensatz zur freiwilligen Prozessoraufgabe in der Regel auch dann, wenn gerade Code einer externen Bibliothek ausgeführt wird.

Die wichtigesten Vorteile von Thread-Implementierungen auf Anwendungsebene sind schnelle Kontextwechsel und ihre Portabilität. Da sie mit relativ geringen Kosten realisiert werden können, eignen sie sich hervorragend für Systeme mit einer hohen Zahl von Threads. Gewichtige Nachteile stellen die fehlende Nutzung mehrer Prozessoren sowie der aufwendige Umgang mit blockierenden Systemaufrufen dar.

5.2.2 Threads als leichtgewichtige Prozesse

Die Vorteile nebenläufiger Programmierung werden nicht erst in Implementierungen moderner Programmiersprachen genutzt. Das Thread-Konzept der Ausführung mehrerer nebenläufiger Aktivitäten auf einem gemeinsamen Adreßraum wird von vielen Betriebssystemen seit langem bereits im Kern unterstützt und Anwendungsprogrammen durch entsprechende Systemschnittstellen angeboten.

Es liegt daher nahe, jeden Programmiersprachen-Thread mit Hilfe jeweils eines LWP zu implementieren, eine sogenannte „1:1-Implementierung“. Mehrere Gründe sprechen dafür:

Einfache Implementierung Die Implementierung eines Schedulers kann entfallen, da dieser vom Betriebssystem bereitgestellt wird.

Multiprozessorfähigkeit Die Ressource „Prozessor“ wird vom Betriebssystem verwaltet. Nur durch Nutzung mehrerer LWP kann dem Betriebssystem die Nebenläufigkeit der Applikation kommuniziert und die parallele Ausführung auf mehreren Prozessoren veranlaßt werden. Eine

Implementierung im *user space* verhält sich dem Betriebssystem gegenüber wie eine nicht nebenläufige Anwendung und kann daher nicht parallel ausgeführt werden.

Blockierende Systemaufrufe Vor allem Ein-Ausgaberoutinen des Betriebssystems können mit Blockiersemantik versehen sein. Man spricht auch von synchroner Ein- und Ausgabe. Das bedeutet, ein I/O-Aufruf kehrt so lange nicht zurück, bis die Operation beendet wurde. Das Betriebssystem sorgt dafür, daß hierbei nur der LWP blockiert, der die I/O-Operation durchführt. Eine Implementierung im *user space* muß statt dessen blockierende I/O-Operationen mit Hilfe nichtblockierender Betriebssystemfunktionen simulieren, da ansonsten der gesamte Prozeß mit allen Threads blockieren würde. Dies ist aufwendig zu implementieren und geht zur Lasten der Effizienz.

Parallelität kann nur durch die Verwendung von LWP erreicht werden, da allein das Betriebssystem Kontrolle über Prozessorvergabe und blockierende I/O-Operationen besitzt. Demgegenüber stehen hohe Kosten für Transitionen vom *user space* in den *kernel space*, die die VM-Performanz belasten, sowie Limitationen der Skalierbarkeit.

5.2.3 Zweistufige Thread-Implementierungen

Sowohl das n:1- als auch das 1:1-Modell entsprechen jeweils einer Sichtweise von Nebenläufigkeit als Abstraktions- oder Parallelisierungsmechanismus. Ihre Nachteile verhindern den Einsatz im jeweils anderen Kontext:

- Implementierungen im *user space* nutzen nicht die Ressourcen einer Mehrprozessormaschine. I/O-Operationen müssen mit Hilfe ineffizienter Hilffunktionen abgewickelt werden.
- Implementierungen mit LWP skalieren schlecht, da ein zunehmender Anteil der Rechenzeit in Kontextwechseln zwischen *user* und *kernel space* verbracht wird.

Zweistufige Implementierungen versprechen Abhilfe durch eine Kombination beider Modelle. Diese Implementierungen nutzen eine auf [Anderson et al., 1991] basierende zweistufige Architektur, in der sowohl Threads auf Anwendungsebene als auch LWP zum Einsatz kommen. Viele

UNIX-Derivate (z.B. AIX, IRIX, Solaris und Tru64 Unix) setzen diese Architektur für ihre Implementierung der POSIX Thread-Schnittstelle um, da sie flexibel über verschiedene Anwendungsprofile skaliert [Vetter et al., 1999, Sun Microsystems, 2000c, Compaq, 1999]. Interessanterweise scheinen erst Java-Implementierungen diese Bibliotheken an ihre Grenzen bringen — eine Zahl von Betriebssystemupdates der Solaris-Thread-Bibliothek, die für den fehlerfreien Betrieb der JVM erforderlich sind, ist ein deutlicher Beleg für diese These [Sun Microsystems, 2000d].

Schlüssel der Architektur ist die Abbildung von vielen im *user space* implementierten Threads auf eine kleine Anzahl von LWP. Maximale Parallelität wird bereits erreicht, wenn die Zahl der LWP gewählt wird als

$$k = p + i \tag{5.1}$$

(k: Zahl der LWP, p: Zahl der Prozessoren, i: Zahl der zu diesem Zeitpunkt aktiven I/O-Operationen). Gleichzeitig ist sichergestellt, daß Kontextwechsel zwischen im *user space* implementierten Threads keinen Aufruf des Betriebssystemkerns erfordern und daher sehr leichtgewichtig sind.

Um diese Zahl zu ermitteln, müssen der Kern-Scheduler und Anwendungs-Scheduler miteinander kommunizieren, insbesondere muß dem Anwendungs-Scheduler das Blockieren eines LWP mitgeteilt werden, damit dieser einen neuen LWP anfordern kann (siehe Abbildung 5.1: Im Idealfall sind genau p (hier 4) LWP im Zustand „Running“). Bietet der Betriebssystemkern keine entsprechende Kommunikationsschnittstelle, so kann vereinfachend eine fixe Zahl von LWP genutzt werden, die in etwa dem erwarteten Wert von k entspricht.

Für Implementierungen virtueller Maschinen, die zu einer hohen Zahl von Threads skalieren und dabei mehrere Prozessoren nutzen sollen, sollte folglich die Wahl auf eine zweistufige Architektur fallen. VM-Implementierungen, die Threads mit Hilfe einer solchen Thread-Bibliothek des Betriebssystems realisieren, profitieren automatisch von genannten Vorteilen. Dazu zählen beispielsweise die Solaris-Versionen der Sun Classic VM [Sun Microsystems, 2000a] mit *native threads* Option, der Sun HotSpot-VM [Sun Microsystems, 1999], und der Sun Research VM [Sun Microsystems, 2000e] sowie das IBM JDK auf AIX [IBM, 2000]. Linux-JVMs, die Threads direkt durch LinuxThreads [Leroy, 1997] realisieren, gehören nicht dazu, da der Linux-Kernel kein Two-Level-Scheduling unterstützt [Bryant and Hartner, 2000]. Gleiches gilt für Java-Maschinen, deren Threads direkt auf die LWPs von Windows NT abgebildet werden.

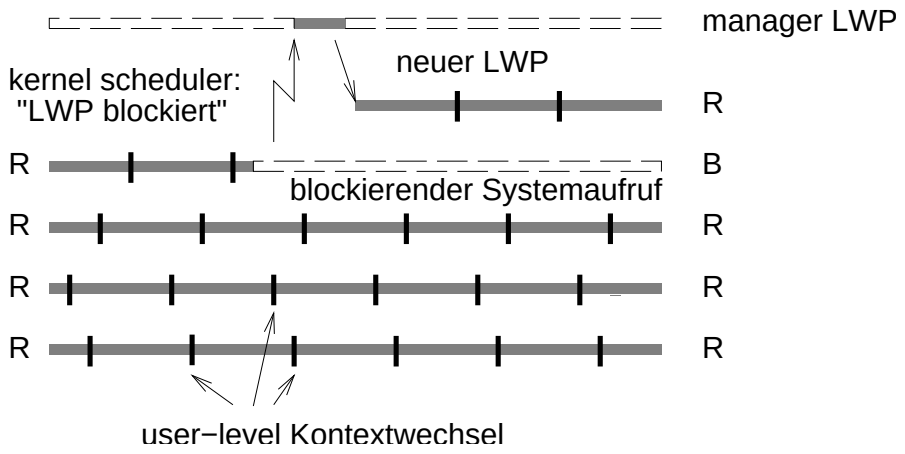

Abbildung 5.1: Blockieren eines LWP (LWP-Zustände: R=Running, B=Blocked)

Insbesondere für die letzten beiden Betriebssysteme existieren mehrere Java-Implementierungen mit VM-basierten Anwendungs-Schedulern. Hierzu zählen JRockit von Appeal [Appeal, 2000] sowie das Laufzeitsystem des NaturalBridge BulletTrain Compilers [NaturalBridge, 2000]. Die IBM Jalapeño VM implementiert einen eigenen Scheduler auf AIX Pthreads (siehe 5.3.1).

5.2.4 Blockierende Aufrufe in separaten Threads

VisualWorks, die Smalltalk-Implementierung von Cincom (ehemals ParcPlace, ObjectShare) [Cincom, 2000] steht in der Tradition von Thread-Implementierungen, die allein im *user space* realisiert sind. Alle Prozesse — so der Ausdruck für Smalltalk-Threads — werden vom Smalltalk-Scheduler auf einem LWP, ohne Interaktion mit dem Betriebssystem verwaltet. Das Ergebnis ist eine hochperformante virtuelle Maschine, die ohne Synchronisation implementiert werden kann.

Aufrufe von Systemroutinen stellen ein gewichtiges Problem dar: blockiert der Aufruf im Kernel, ist die gesamte virtuelle Maschine suspendiert. Dies führt bestenfalls zu schlechter Laufzeitperformanz, schlimmstenfalls zu Deadlock. Dies ist für serverseitige Anwendungen nicht akzeptabel. Smalltalk-Code soll jedoch weiterhin nur von einem LWP ausgeführt werden, um die effiziente Maschinenarchitektur beibehalten zu können.

Diese Überlegungen führen zu folgendem Entwurf [Miranda, 1997]: dem Smalltalk-VM-LWP wird ein Pool von LWPs zur Seite gestellt, die exklu-

siv zur Ausführung externer Bibliotheksaufrufe genutzt werden. Die Bibliotheksdeklarationen einer externen Routine ist dazu mit dem Schlüsselwort `_threaded` zu markieren und bedeutet der VM, entsprechende Aufrufe auf einem eigenen Betriebssytemkontext auszuführen.

Ruft nun ein Smalltalk-Prozeß eine solche Routine auf, so wird der Aufruf dem LWP-Pool übergeben und der Prozeß in der VM als suspendiert markiert. Die VM kann nun andere Prozesse ausführen. Nach erfolgter Ausführung des externen Aufrufs oder bei einem Rückruf in die VM wird der rufende Prozeß wieder als lauffähig markiert und nimmt am Scheduling der VM teil. Es findet hier also keine Kommunikation mit dem Scheduler des Betriebssystems statt, sondern die Aufrufe werden bereits präventiv auf einen anderen LWP umgeleitet.

Man beachte, daß diese Thread-Implementierung für den Fall $p = 1$, d.h. einer Uniprozessormaschine, dem Idealmodell dieses Abschnitts entspricht. Dabei wird die Besonderheit ausgenutzt, daß die I/O-Prozessoren nicht auf VM-Strukturen zugreifen und somit keine Synchronisierung erforderlich ist.

5.3 Stopping The World

Für die Speicherverwaltung stellt das Thread-System einen wichtigen Dienst zur Verfügung: um einen konsistenten Satz von Wurzelreferenzen innerhalb der Thread-Stacks sicherstellen zu können, müssen alle Threads angehalten werden. Im Falle eines exakten Collectors kommt erschwerend hinzu, daß sich alle Threads an einem konsistenten Punkt mit entsprechenden Referenztabellen befinden müssen.

Da im Allgemeinen nicht an jeder Instruktion eine entsprechende Referenztabelle (2.2.2) zur Verfügung steht, müssen die Threads bis zu einem der nächsten konsistenten Punkte fortfahren. Erst wenn alle Threads an so einem Punkt angehalten haben, kann der Garbage Collector seine Arbeit beginnen. Um die durch eine Garbage Collection verursachten Pausen möglichst klein zu halten, ist es von großer Bedeutung, diese Transition auch für eine große Zahl von Threads effizient vollziehen zu können.

Es lassen sich zwei grundsätzliche Strategien unterscheiden: *Polling* und Codemodifikation. *Polling* erfordert, daß Threads regelmäßig an *GC points* die Aufforderung zur Garbage Collection erfragen und bei positivem Ergebnis die Transition vornehmen. Dies erfordert entsprechenden Code an allen potentiellen Haltepunkten. Zur Platzoptimierung kann man hier z.B. auf ein *polling* an Aufrufstellen verzichten und dieses statt dessen in Methodenpro-

logen ausführen. Da ein *polling* an jedem Methodenaufruf und jeder Schleife ausgeführt wird, ist dieser Mechanismus sehr sorgfältig zu optimieren. Die effizienteste Realisierung (Verzweigungen, Unterbrechungen o.ä.) hängt stark von der verwendeten Architektur und dem Betriebssystem ab.

Polling ist im Interpretermodus eine zu vernachlässigende Größe; bei übersetzten Methoden ergibt sich ein anderes Bild. Die Laufzeiteffizienz läßt sich hier durch Codemodifikation signifikant verbessern. Hierzu wird folgendermaßen vorgegangen: um eine Garbage Collection einzuleiten, werden alle Threads suspendiert. Daraufhin wird der Maschinencode der gerade ausgeführten Methoden an designierten Haltepunkten dahingehend modifiziert, daß der Thread eine unbedingte Transition zur Garbage Collection vornimmt. Es ist dabei erforderlich, alle möglichen Kontrollpfade abzudecken. Da ein Methodenaufruf in der Regel mehrere potentielle Ziele hat, ist es sinnvoller, die aufrufende statt die aufgerufenen Methoden zu modifizieren. Die inkonsistenten Threads werden nun wieder aufgeweckt und laufen bis zu einem solchen Punkt – man sagt auch, sie werden vorgerollt. Diese Technik ähnelt dem Vorgehen von Debuggern beim Setzen von Breakpoints. Eine detaillierte Darstellung der Implementierung in der Sun Research VM findet sich in [Agesen, 1998].

5.3.1 Unterbrechung nur an konsistenten Punkten

In einer zeitgesteuerten zweistufigen Thread-Bibliothek hat das ausgeführte Programm keinen Einfluß auf die Unterbrechungspunkte. Dies hat zur Folge, daß sich p Threads in Ausführung befinden, i Threads konsistent in Systemaufrufen blockiert sind, und $n-p-i$ Threads wahrscheinlich in inkonsistenten Regionen unterbrochen und suspendiert sind (Abbildung 5.2). Die Transition zur Garbage Collection bedeutet daher, daß $n - i$ Threads bis zu einem *GC point* vorgerollt werden müssen[1].

Stellt man statt dessen sicher, daß Threads ohnehin nur an konsistenten Punkten unterbrochen werden, reicht es aus, $p \ll n - i$ Threads bis zur nächsten Unterbrechung fortlaufen zu lassen (Abbildung 5.3). Eine entsprechende Unterbrechungsstrategie muß durch einen im *user space* implementierten Scheduler in der VM realisiert werden; es ist daher nicht möglich, auf die generischen Betriebssystembibliotheken zurückzugreifen. Die Autoren der Jalapeño Virtual Machine von IBM Research argumentieren, auf diese Weise die Transition zur Garbage Collection signifikant zu beschleuni-

[1] [Agesen, 1999] deutet eine potentielle deutliche Reduktion dieser Zahl von inkonsistenten Threads an, vertieft dieses jedoch nicht weiter.

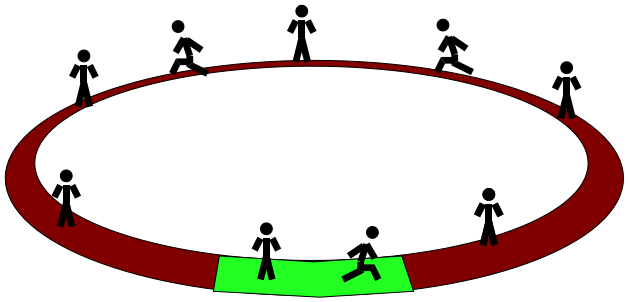

Abbildung 5.2: Konsistente und inkonsistente Threads bei beliebiger Unterbrechungsstrategie: $n - i \gg p$ Threads sind inkonsistent.

gen [Alpern et al., 2000].

Allerdings muß man folgendes beachten: Die Implementierung des Schedulers wird hierdurch komplexer. Der Jalapeño Scheduler kann sich auf *polling* eines hardwareunterstützten, zeitgesteuerten Prozessorbits abstützen. Fehlt eine solche Unterstützung, liegt der Verdacht nahe, daß der Aufwand hier von der Speicherverwaltung in den Scheduler verlagert wird; es folgt also kein Performancegewinn, dafür aber eine gleichmäßigere Verteilung des Verwaltungsaufwands. Entsprechende Studien sind uns nicht bekannt.

5.4 Synchronisierung

Implementierer des Thread-Systems einer virtuellen Maschine für Java sehen sich insbesondere mit zwei Problemen konfrontiert:

1. Jedes Objekt kann zur Synchronisation verwendet werden.
2. Die Verwendung thread-sicher implementierter Standardbibliotheken führt zu einer hohen Rate von Synchronisationsoperationen.

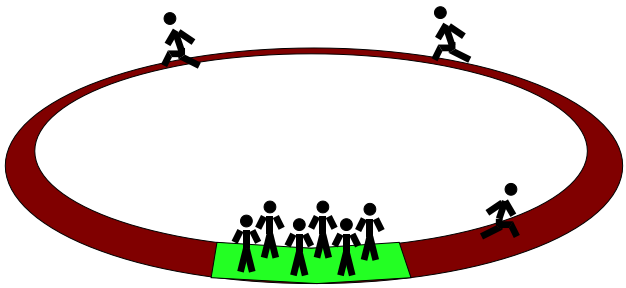

Abbildung 5.3: Konsistente und inkonsistente Threads bei Unterbrechung nur an konsistenten Punkten: nur p Threads sind inkonsistent.

Der Entwickler muß daher sorgfältig die Platz- und Zeiteffizienz der Monitorimplementierung gegeneinander abwägen.

In diesem Abschnitt wird dargestellt, welche Implementierungsmodelle für Monitore existieren. Unter anderem demonstrieren wir, wie sich die Synchronisationsprimitive „Monitor“ wiederum auf einfachere Synchronisationsoperationen — bis hin zu atomaren Prozessorinstruktionen — abbilden lässt. Am Ende des Abschnitts wird detailliert auf einzelne Implementierungen eingegangen.

5.4.1 Entwurfsdimensionen

In diesem Abschnitt werden einige ausgewählte Entwurfsdimensionen für die Monitorimplementierung und ihre Auswirkung auf die Systemperformanz diskutiert.

Sperrstrategie Ist eine Sperre bereits erworben, so kann ein weiterer Erwerber entweder blockieren oder die Sperre iterativ testen (*spin locking, busy wait*). Letztere Option sollte nur genutzt werden, wenn dies weniger Prozessorzeit benötigt als der sonst erforderliche mehrfache Kontextwechsel. Da

Monitore in Java über unbestimmte Zeit gehalten werden können, sollte eine blockierende Strategie gewählt werden.

Allokationszeitpunkt Java-Objekte können mit Monitoren ausgestattet werden, indem diese direkt in die Objektstruktur eingebettet oder bei der Objektallokation separat angelegt und aus dem Objekt referenziert werden. Wie bereits argumentiert, ist dieses Implementierungsmodell aufgrund der Allokation überwiegend ungenutzter Monitore nicht akzeptabel. Monitore müssen daher dann angelegt werden, wenn sie erstmals benötigt werden.

Zuordnungsmechanismus Da eine Einbettung in das Objekt aus Platzgründen entfällt, müssen Monitorstrukturen ihren Objekten zugeordnet werden. Dies kann entweder durch direkte Referenzen aus dem Objektkopf oder durch indirekte Zuordnung mittels einer Tabelle geschehen.

Hier wird Zeit- gegen Platzeffizienz abgewogen: eine direkte Referenz ermöglicht Zugriff auf den Monitor in konstanter und kurzer Zeit, während die Monitorreferenz als ein Wort pro Objekt zu Buche schlägt. Um den Platzbedarf zu mindern, kann die Monitorreferenz an Stelle eines anderen Wortes im Objektkopf abgelegt werden. Der überschriebene Wert — typischerweise der Hashwert des Objektes — wird dann in der Monitorstruktur abgelegt. Diese Technik wird *header word displacement* genannt und in der Sun Research VM [Sun Microsystems, 2000e] sowie der Hotspot Performance Engine [Sun Microsystems, 1999] eingesetzt. Beide Monitorimplementierungen werden in den Abschnitten 5.4.4 respektive 5.4.5 näher diskutiert.

Eine Tabellenstruktur hat keinen Platzbedarf für nicht genutzte Monitore, allerdings kann kein Zugriff in konstanter Zeit garantiert werden. Insbesondere muß die Tabelle selbst thread-sicher gemacht werden; Zugriffskonflikte auf dieser zentralen Struktur sind wahrscheinlich, weshalb die Laufzeitperformanz kritisch zu betrachten ist. Die Sun Classic VM [Sun Microsystems, 2000a] und Cacao [Krall and Probst, 1998] implementieren den Monitorzugriff auf diese Weise (die Tabelle und ihre Sperre heißen hier *monitor cache* und *monitor cache lock*).

Variable Monitordarstellung Häufig wird ein Monitor nur von einem Thread rekursiv erworben, ohne daß ein Konflikt vorliegt oder die Ereignisvariable genutzt wird. In diesem Fall ist die Darstellung einer Warteliste überflüssig und der Monitor reduziert sich auf den Besitzeridentifikator und einen (kleinen) Rekursionszähler, welche gemeinsam in einem Maschi-

nenwort repräsentiert werden können. Statt einer separaten Struktur kann der Monitor also im Objektkopf abgelegt und damit schneller zugegriffen werden. Nur bei Bedarf (Konflikt oder hohe Rekursionstiefe) muß er in eine separate Struktur konvertiert werden. Diese Technik wurde erstmals von [Bacon et al., 1998] für Java-Monitore vorgeschlagen und mehrfach verfeinert [Alpern et al., 2000, Onodera and Kawachiya, 1999]. Dieses Modell wird detailliert in Abschnitt 5.4.5 beschrieben.

5.4.2 Feinstruktur von Monitoren

Obwohl Monitore in Java Sprachprimitive sind, mit deren Hilfe nebenläufige Berechnungen auf gemeinsamen Daten koordiniert werden, zeigt sich, daß sie selbst komplexe Datenstrukturen darstellen. Ein Monitor nach Java-Semantik enthält eine rekursive Sperrre und eine Ereignisvariable und muß demnach mindestens folgende Informationen vereinigen:

```
monitor:
  owner       ; Besitzer der Sperre (aktiv)
  depth       ; Erwerbungstiefe
  entering    ; Menge der erwerbenden Threads (suspendiert)
  waiting     ; Menge der wartenden Threads (suspendiert)
```

Weiterhin müssen — abhängig von der Thread-Implementierung — Mechanismen zum Suspendieren und Reaktivieren von Threads existieren.

Damit lässt sich eine abstrakte Implementierung für Monitore konstruieren: die Monitoroperationen `monitorenter` und `monitorexit` können dann wie in Abbildung 5.4 implementiert werden. Sie garantieren gegenseitigen Ausschluß im Zugriff auf ihr Java-Objekt.

Die gewählte Monitorimplementierung hängt von der gewünschten Monitorsemantik ab — z.B. ob und wie die Erwerbungsreihenfolge von der Ankunftsreihenfolge der Threads abhängen soll. Die Java-Spezifikation gibt hier keine besondere Semantik vor.

Unabhängig davon ist der hier dargestellte Algorithmus in einem zentralen Punkt unvollständig: die Monitorstruktur selbst ist nicht vor nebenläufigem Zugriff geschützt. Um einen atomaren Übergang des Monitorzustands zu garantieren, muß sie wiederum durch eine Sperre abgesichtert werden. [Agesen et al., 1999] sprechen von einer Metasperre (*meta-lock*).

```
monitorenter:
  if owner is null:
    owner := current thread
    depth := 1
  else if owner is current thread:
    depth++
  else:
    add current thread to entering
    suspend current thread

monitorexit:
  require owner be current thread
  if depth > 1:
    depth--
  else if entering is empty:
    owner := null
  else:
    owner := some element removed from entering
    depth := 1
    wake up owner
```

Abbildung 5.4: Pseudo-Implementierung der Monitoroperationen `monitorenter` und `monitorexit`

5.4.3 Metasperren

Die Notwendigkeit für eine Metasperre stellt keinen rekursiven Schluß dar, da Metasperren stark vereinfachte Eigenschaften aufweisen:

- Metasperren werden nicht rekursiv erworben. Damit braucht kein Zähler der Rekursionstiefe geführt zu werden.
- Ein Thread hält maximal eine Metasperre zur Zeit. Es ist also durch Normalisierung dieser 1:[0..1]-Beziehung möglich, Informationen aus der Monitorstruktur direkt dem besitzenden Thread, statt einer zusätzlichen Struktur zuzuordnen. Es wird sich später zeigen, wie sich diese Eigenschaft ausnutzen lässt.
- Metasperren werden nur über definierte, kurze Codestücke gehalten. Kann man sicherstellen, daß einem Thread, der eine Metasperre hält, nicht über einen längeren Zeitraum der Prozessor entzogen wird, so ist es vertretbar, eine Metasperre durch *spin-locking* zu realisieren, und es brauchen keine Wartelisten geführt zu werden. Dies hängt vor allem von der Unterbrechungsstrategie der Thread-Implementierung ab.

Als Folge lässt sich eine Metasperre auf ein Maschinenwort reduzieren, das mit Hilfe atomarer Maschineninstruktionen (z.B. `CAS` [SPARC, 1998]) erworben wird. Abbildung 5.5 stellt die vereinfachte Monitorimplementierung ergänzt um Metasperren dar.

Beispiele

Metasperren finden sich vielfach in Implementierungen des POSIX Thread Standards [POSIX, 1996]. Dieser definiert einen abstrakten Typ `pthread_mutex_t`, welcher ähnlich obiger Monitorstruktur implementiert werden kann. Ein Feld der Sperrstruktur fungiert dann als Metasperre, z.B. `spinlock` bei LinuxThreads [Leroy, 1997]. Virtuelle Maschinen, die Synchronisation mit Hilfe solcher Thread-Bibliotheken realisieren, verwenden demnach entsprechende Strukturen. Dies gilt beispielsweise für die Sun Classic VM [Sun Microsystems, 2000a] mit *native threads* Option.

Die Jalapeño Virtual Machine von IBM Research [Alpern et al., 2000] ist mit minimalen Betriebssystemabhängigkeiten implementiert. Auch die Synchronisationsoperationen sind im user-level mit Hilfe zweier Instruktionen der PowerPC-Architektur [May et al., 1994] implementiert. `lwarx` (*load reserve*)

```
acquire metalock:
  while cas(metalock, 0, 1) fails:
    spin

release metalock:
  metalock := 0

monitorenter:
  acquire metalock             ##
  if owner is null:
    owner := current thread
    depth := 1
    release metalock           ##
  else if owner is current thread:
    depth++
    release metalock           ##
  else:
    add current thread to entering
    release metalock           ##
    suspend current thread

monitorexit:
  acquire metalock             ##
  require owner be current thread
  if depth > 1:
    depth--
    release metalock           ##
  else if entering is empty:
    owner := null
    release metalock           ##
  else:
    owner := some element removed from entering
    depth := 1
    wake up owner
    release metalock           ##
```

Abbildung 5.5: Pseudo-Implementierung der Monitoroperationen `monitorenter` und `monitorexit` erweitert um Metasperren

lädt ein Speicherwort und vermerkt gleichzeitig eine — nicht verpflichtende — Reservierung der entsprechenden Cache-Line. `stwcx` (*store conditional*) speichert ein Wort, vorausgesetzt, die Reservierung der Cache-Line wurde eingehalten. Dieses Instruktionspaar lässt sich somit wie eine „Compare-and-Swap"-Instruktion verwenden.

Jalapeño benutzt zwei verschiedene Repräsentationen für Monitore - *thin* und *fat locks*. *Thin locks* stellen eine weitere Optimierung dar, die wir in Abschnitt 5.4.5 behandeln. Jalapeños *fat locks* hingegen entsprechen der oben dargestellten Monitor-Implementierung nahezu vollständig. Ein *fat lock* ist eine eigenständige Struktur, die durch ein Feld im Objektkopf referenziert wird. Sie enthält alle Felder der Beispielimplementierung `metalock, owner, depth, entering, waiting` sowie ein sechstes Feld, welches das zugeordnete Java-Objekt referenziert. *Fat locks* können nach Bedarf verschiedenen Java-Objekten zugeordnet werden. Über das zusätzliche sechste Feld wird das Zuordnungsprotokoll Objekt↔Monitor abgewickelt.

Auf Anwendungsebene realisierte Thread-Implementierungen, wie z.B. green threads, deaktivieren den Unterbrechungsmechanismus so lange, wie ein Thread eine Monitorstruktur manipuliert. Aus diesem Grund brauchen Monitorstrukturen nicht vor nebenläufigem Zugriff geschützt zu werden. Man kann dieses Vorgehen als degenerierte, globale Metasperre ansehen.

5.4.4 Eine fortgeschrittene Metasperrenimplementierung

[Agesen et al., 1999] streben eine möglichst platzeffiziente und trotzdem performante Monitorimplementierung an. Die Monitorstruktur beruht auf folgenden Vorgaben:

- Wenige Objekte werden als Monitore genutzt. Der Platzbedarf unbenutzter Monitore sollte also möglichst gering gehalten werden. Dazu wird das Hashfeld im Objektkopf multimodal genutzt; der aktuelle Modus wird durch zwei freie Markierungsbits kodiert. Im überwiegenden Fall eines ungenutzten Monitors entstehen daher Platzkosten von lediglich zwei Bit.
- Während eines Überganges des Monitorzustandes, geschützt durch die Metasperre, braucht die Monitorstruktur für andere Threads nicht verfügbar zu sein, da sie ohnehin nicht zugreifen dürfen. Die Metasperre kann also im gleichen Feld wie die Monitordaten angelegt werden.

- Es brauchen keine eigenen Metasperrstrukturen (mit gesicherten Monitor- oder Hashzuständen) angelegt zu werden, da ein Thread nur eine Metasperre zur Zeit halten kann. Statt dessen können diese direkt auf dem Thread-Stack repräsentiert werden.

- Threads sind durch die entsprechende Systembibliothek des Solaris Betriebssystems realisiert. Auch wenn Metasperren gewöhnlich nur extrem kurz (wenige 10 SPARC-Instruktionen) gehalten werden, können im Ausnahmefall durch Entscheidungen des Schedulers längere Phasen mit erworbenen Metasperren auftreten. Metasperren sollten daher nicht durch *spin locks* implementiert werden. Statt dessen blockieren Threads bei konkurrierendem Erwerb einer Metasperre, und eine Listenstruktur ermöglicht die Übergabe der Metasperre an einen Nachfolger (*handoff*).

Die Struktur der Metasperre

Besitzer und Bewerber einer Metasperre bilden eine verkettete Liste. Jeder Thread referenziert dabei den Thread, der sich zuvor in die Liste eingetragen hat, in einer lokalen Variable. Die Eintragung findet durch einen atomaren Tausch des bisherigen Wortes im Objektkopf mit der Referenz auf das Thread-Objekt statt (SPARC-V9-Instruktion `SWAP` [SPARC, 1998]).

Nach dem Tausch kann der Thread anhand der Markierungsbits des vorherigen Wertes entscheiden, ob er die Metasperre erwerben konnte, oder ob diese bereits den Verweis auf einen Vorgänger-Thread enthält und damit ein Konflikt beim Erwerb der Metasperre besteht.

Wurde die Metasperre erfolgreich erworben, so kann die Monitorstruktur manipuliert werden und in den Objektkopf zurückgeschrieben werden — vorausgesetzt, kein anderer Thread hat sich für die Metasperre angemeldet (siehe Abbildung 5.6). Der Besitzer kann eine etwaigen Konflikt anhand des aus dem Objektkopf getauschten Wertes erkennen: enthält er noch die Adresse des jetzigen Metasperrenbesitzers, so kann die Monitorstruktur geschrieben werden — diese Operationsfolge wird atomar durch die SPARC-V9-Instruktion `CAS` realisiert.

Schlägt `CAS` fehl, so hat sich bereits ein anderer Thread in die Bewerberliste eingetragen. Die Metasperre und die neue Monitorstruktur werden direkt an diesen weitergereicht (wie in Abbildung 5.7). Beide Threads vollziehen dazu ein Rendezvous, welches mit Hilfe von POSIX Synchronisierungsprimitiven realisiert wird. Dazu enthält jede Thread-Struktur je eine Betriebssystem-Sperre und -Ereignisvariable.

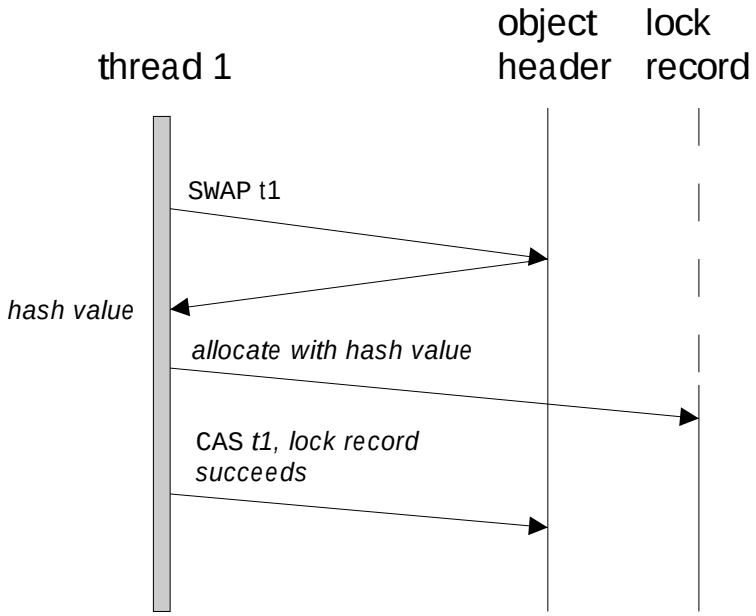

Abbildung 5.6: Metasperre nach [Agesen et al., 1999] — keine Konflikt

Diese Implementierung von Metasperren ist extrem speichereffizient, denn sie erfordert nur zwei freie Bits im Objektheader für den Fall eines nicht erworbenen Monitors. Trotzdem wird die Laufzeitperformanz nicht negativ beeinflußt: Erwerb und Freigabe eines Monitors erfolgen in wenigen SPARC-Instruktionen, darunter je zwei atomare Instruktionen — mit einer weiteren Optimierung können letztere auf je eine atomare Instruktion für den Fall ohne Monitor-Koflikt reduziert werden.

Wird ein Monitor erworben oder zum Warten verwendet, so müssen Sperrstrukturen alloziert werden, die jedoch sofort nach Benutzung wiederverwendet oder freigegeben werden können. Der dafür benötigte Platz ist vernachlässigbar (< 100k), wie Messungen von [Agesen et al., 1999] ergeben. Die Allokation findet in thread-lokalen Bereichen statt, so daß hier keine Synchronisation vonnöten ist.

5.4.5 Variable Monitordarstellungen

Wie diskutiert dominieren in Java zwei Monitorzustände, nämlich ein nicht erworbener Monitor sowie ein konfliktfrei erworbener Monitor. Es ist daher sinnvoll, zu untersuchen, ob für diese Zustände eine andere Monitorrepräsentation als für die restlichen von Vorteil sein könnte.

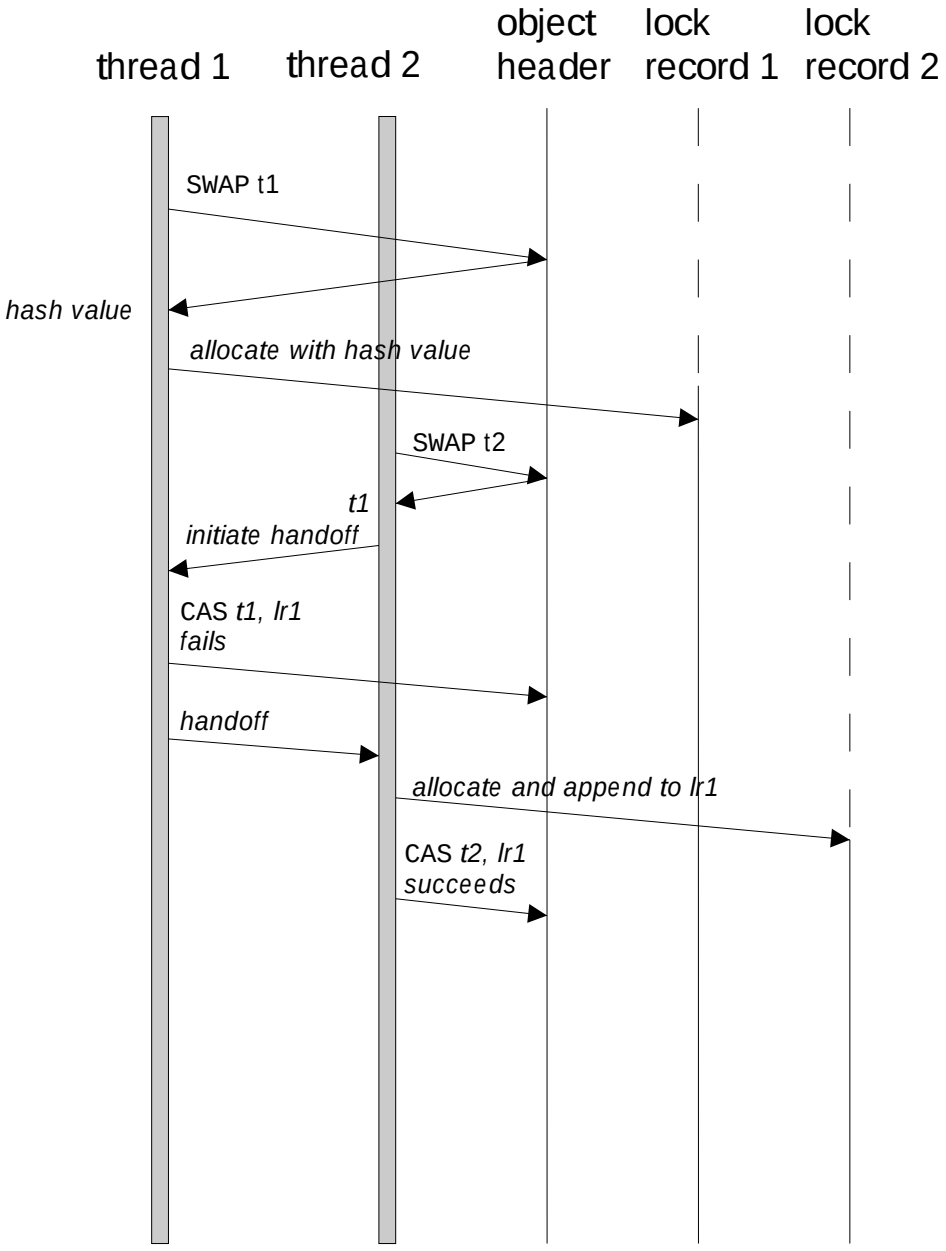

Abbildung 5.7: Metasperre nach [Agesen et al., 1999] — Konflikt beim Erwerb der Metasperre

Stack-Allokation von Monitoren

Die HotSpot-VM [Sun Microsystems, 1999] nutzt die Blockstruktur der Java-Synchronisationskonstrukte in folgender Hinsicht aus: alle Monitore, die in einer Aktivierung erworben werden, müssen auch in dieser Aktivierung freigegeben werden[2]. Besteht kein Konflikt, so können die Monitore also sehr leichtgewichtig auf dem Stack alloziert werden.

Hotspot repräsentiert solche leichtgewichtigen Sperren durch zwei Worte (*object id, displaced header*)[3]. Jeder rekursive Erwerb eines Monitors wird durch eine neue Sperrstruktur auf dem Laufzeitstack repräsentiert. In diesem Fall enthält das zweite Wort eine 0, die anzeigt, daß das Objekt rekursiv gesperrt wurde und auf einer niedrigeren Stackposition weitere Sperrstrukturen liegen.

Abbildung 5.8 zeigt einen in mehreren Aktivierungen rekursiv erworbenen Monitor: Es wurde gerade die Methode `get` von der Stelle (*) aufgerufen. Sowohl `lookup` als auch `get` synchronisieren auf die Hashtabelle. Daher existieren in beiden Aktivierungen Sperrstrukturen, wobei die erste den Hashwert des Objektes gesichert hat. Dieses trägt statt dessen nun einen Verweis auf die Sperrstruktur im Laufzeitstack.

Der Test auf rekursiven Monitorerwerb ist sehr einfach: verweist das entsprechende Headerfeld des Objekts in den eigenen Stack (`lock < sp && sp - lock < 64k`), so hat dieser Thread die Sperre bereits erworben. Die Monitorfreigabe erfolgt durch lineare Suche im aktuellen Ausführungsrahmen nach der obersten Sperrstruktur für dieses Objekt. Aufgrund der typischerweise sehr kleinen Anzahl von Monitoroperationen pro Aktivierung stellt dieses Vorgehen kein Laufzeitproblem dar.

Im Falle eines Konflikts muß der Monitor in eine Heap-Struktur mit Betriebssytem-Sperre und -Ereignisvariable umgewandelt werden — dieser Vorgang wird als Expansion oder *inflation* bezeichnet. Er wird durch einen nachfolgenden Bewerber ausgeführt, der sich mit weiteren Bewerbern über eine globale Sperre koordiniert. Die Expansion eines Monitor tritt jedoch relativ selten auf, so daß diese globale Sperre kein Effizienzproblem verursacht. Die Rückführung in den leichtgewichtigen Monitorzustand erfolgt während der Garbage Collection. Dies hat den Vorteil, daß alle Threads angehalten sind und keine aufwendige Synchronisation stattfinden muß.

[2]Die VM-Spezifikation [Lindholm and Yellin, 1996] diktiert zwar keine Blockstrukturierung der Synchronisierungskonstrukte, *erlaubt es einer VM jedoch, eine solche Einschränkung zu machen!*. Effektiv bedeutet dies, daß die Synchronisierungskonstrukte geschachtelt sein müssen, will man portablen Java-Bytecode erzeugen.

[3]Siehe 5.4.1 zu header word displacement.

```
class Cache {
  Hashtable table;

  Object lookup(Object key) {
    synchronized(table) {
      Object result = table.get(key); (*)
      ...
}

class Hashtable {
  synchronized Object get(Object key) ...
}
```

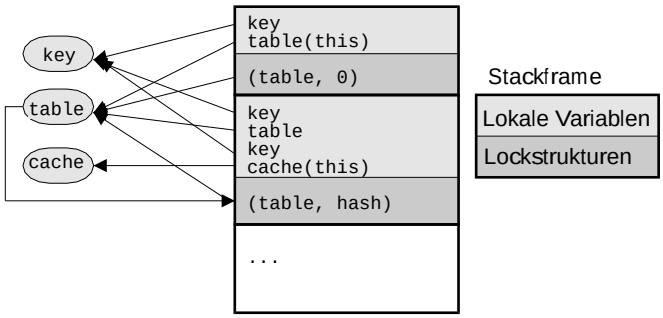

Abbildung 5.8: Hotspot: Aufrufstack mit rekursivem Lock

Die Monitorimplementierung der HotSpot-VM zeichnet sich durch hohe Laufzeitperformance aus, da die Monitorallokation auf dem Stack erfolgt. Durch Einsatz von *header word displacement* kann der Objektkopf auf zwei Worte beschränkt werden.

Thin Locks

[Bacon et al., 1998] schlagen eine andere Optimierung des Sperrprotokolls vor: ein Feld im Objektkopf wird bimodal genutzt; zum einen, um direkt einen Monitor zu kodieren, zum anderen, um einen Verweis auf eine separate Monitorstruktur zu halten. Nur der Besitzer des Monitors ist in der Lage, das Feld zu verändern.

Die beiden häufigsten — und einfachsten — Monitorzustände werden direkt in diesem Feld kodiert:

- ein unbenutzter Monitor — das Feld im Objektkopf enthält den Wert 0.
- ein erworbener Monitor ohne Erwerbungskonflikt, ohne Wartende und mit kleiner Rekursionstiefe — das Feld im Objektkopf enthält die Thread-Id des Besitzers und die Rekursionstiefe.

Der Vorteil von *thin locks* besteht darin, daß dominierende Fall eines Monitors ohne Erwerbungskonflikt, der wenige Male rekursiv erworben wurde, hier stark optimiert wird:

- Das erstmalige Erwerben des Monitors erfolgt durch ein *load-reserve/store-conditional* Paar der PowerPC-Architektur [May et al., 1994] (vgl. Jalapeño in Abschnitt 5.4.3), vorausgesetzt das gelesene Feld im Objektkopf enthält den Wert 0.
- Enthält das Feld einen Wert ungleich 0, ist aber der Besitzer der aktuelle Thread, so kann der Rekursionszähler ohne Einsatz atomarer Instruktionen erhöht werden.
- Die Monitorfreigabe kann durch einfaches Überschreiben des Feldes im Objektkopf erfolgen. Insbesondere braucht keine atomare Instruktion verwendet zu werden.

Im Falle eines Konfliktes muß das *thin lock* in eine Monitorstruktur konvertiert werden — es muß expandiert werden. Das von [Bacon et al., 1998] vorgeschlagene Expansionsprotokoll hat mehrere Schwachpunkte:

- Der nächste Erwerber ist zuständig für die Konversion. Dazu muß er aktiv auf dem Objekt warten *busy waiting*, bis der Monitor freigegeben wurde (das Feld im Objektkopf den Wert 0 enthält). Dies kann für unbestimmte Zeit nötig sein, und bedeutet damit eine Verschwendung von Prozessorzeit.
- Es gibt keinen Mechanismus, die Konversion rückgängig zu machen (*deflation*). Monitore, bei deren Erwerb nur einmal ein Konflikt aufgetreten ist, bleiben dann unnötigerweise im schwergewichtigen Zustand.

Das Modell von [Bacon et al., 1998] wurde in beiden Punkten signifikant von [Onodera and Kawachiya, 1999] verbessert. Das Expansionsprotokoll verzichtet auf *busy waiting*, und die Rückführung eines Monitors in den leichtgewichtigen Zustand wird unterstützt.

Das Expansionsprotokoll nach Onodera und Kawachiya

Im Falle eins Konfliktes muß eine Monitorstruktur für das betroffene Objekt alloziert werden. Dies geschieht über eine globale Tabelle; diese Struktur ist

eindeutig dem Objekt zugeordnet, d.h. mehrfache Anfragen für das Objekt liefern immer die gleiche Struktur. Die Lösungs beruht darauf, die Monitorstruktur selbst zur Koordination des Expansionsprotokolls zu verwenden. Sie wird also selbst in zwei Modi verwendet — zunächst als Koordinationspunkt für das Expansionsprotokoll. Ist dieses beendet, übernimmt sie die Rolle des Monitors selbst. In der Monitorstruktur werden Betriebssystemobjekte für die Synchronisierung genutzt.

Der Expansionsalgorithmus funktioniert folgendermaßen, entsprechend der 5 Phasen in Abbildung 5.9.

1. Ein Thread erwirbt den leichtgewichtigen Monitor, indem er `(Thread-Id, 1)` in das Monitorfeld des Objektkopfes schreibt.

2. Weitere Threads scheitern beim Erwerb. Über die globale Tabelle wird eine Monitorstruktur angelegt. Alle Threads erwerben nun die Sperre der Monitorstruktur und warten auf der Ereignisvariable der Monitorstruktur. Zuvor wird dem Monitorbesitzer der Wunsch nach Monitor-Expansion durch das Setzen eines freien Bits im Objektkopf außerhalb des Monitorfeldes (*FLC-Bit* nach Onodera und Kawachiya) kommuniziert.

3. Der Monitorbesitzer gibt den Monitor frei, indem er das Monitorfeld auf 0 setzt. Danach prüft er das FLC-Bit und stellt fest, daß das Expansionsprotokoll initiiert wurde. Daraufhin erwirbt er die Sperre der Monitorstruktur, die er aus der globalen Tabelle erfragt, und signalisiert die Ereignisvariable.

4. Ein wartender Thread wacht auf und erwirbt wieder den Mutex der Monitorstruktur. Er stellt fest, daß der Monitor freigegeben ist und installiert den Verweis auf die Monitorstruktur im Objektkopf. Er benachrichtigt alle anderen auf Expansion wartenden Threads; diese können allerdings nicht sofort auch den Mutex erwerben. Der erste Thread kann nun in die kritische Region eintreten.

5. Bei der Monitorfreigabe gibt der Thread die Sperre der Monitorstruktur frei, und ein anderer Thread kann diesen erwerben.

Treten keine Konflikte beim Monitorerwerb auf, so ist dieses Sperrprotokoll extrem effizient. Insbesondere verzichtet der Algorithmus auf atomare Instruktionen bei der Monitorfreigabe im gemeinen Fall. Daraus resultiert allerdings eine race condition zwischen freigebendem Thread (Phase 3)

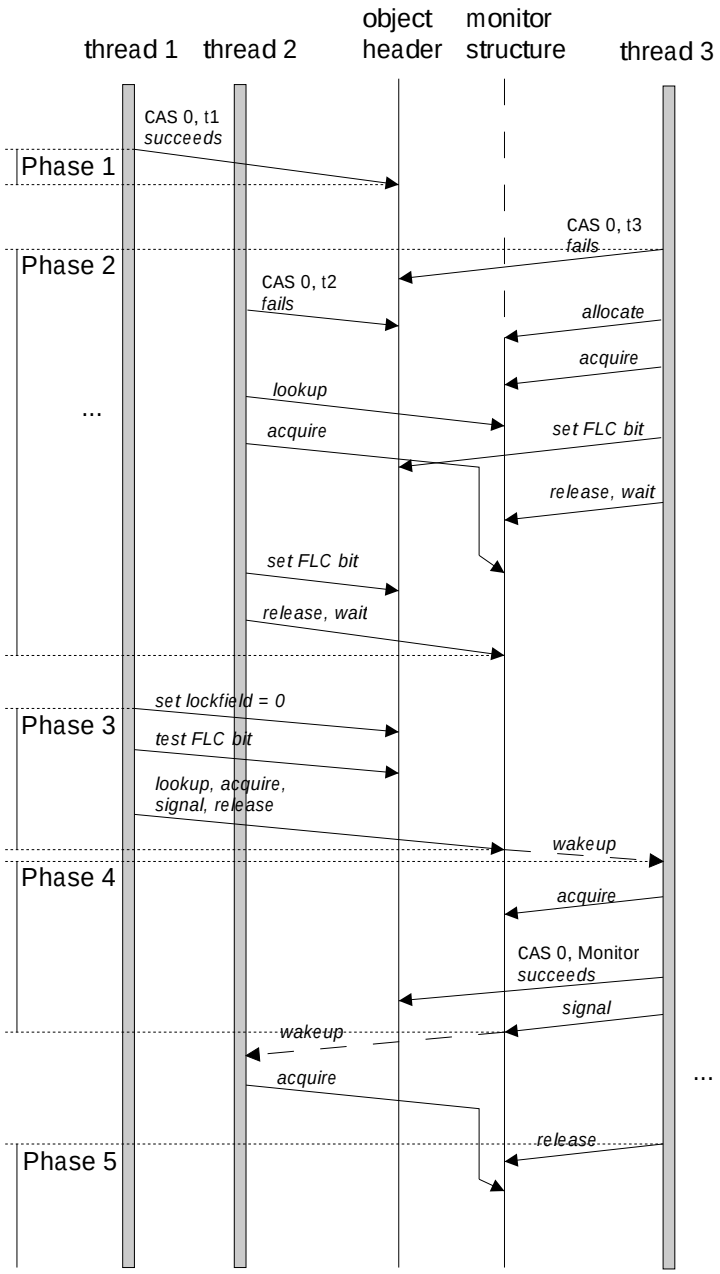

Abbildung 5.9: Expansion eines Monitors nach [Onodera and Kawachiya, 1999]

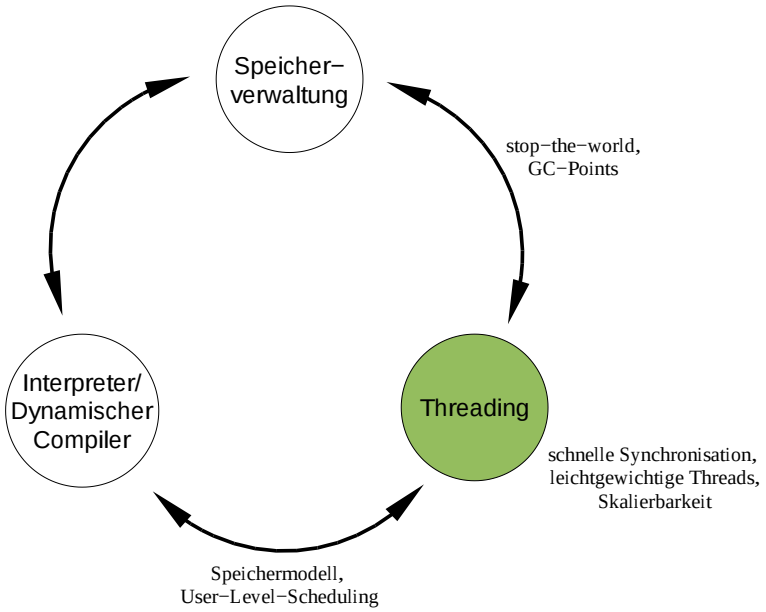

Abbildung 5.10: Die drei Hauptkomponenten aus Sicht des Threading-Subsystems

und erwerbendem Thread (Phase 2), die dazu führen kann, daß der freigebende Thread den Monitor zu erwerben sucht, *nachdem* der nachfolgende Thread ihn bereits erwerben konnte. Dadurch könnte der freigebende Thread unnötig blockiert werden, als sei er unfair vom Scheduler verzögert worden [Onodera, 2000].

Synchronisierung basierend auf *thin locks* wird von IBM in den aktuellen Java 2 Development Kits sowie in der Jalapeño VM eingesetzt.

5.5 Zusammenfassung

Durch klar definierte und einfache Konstrukte und deren enge Integration in den Sprachstandard ermuntert die Java-Plattform zu einem großzügigen Umgang mit nebenläufigen Konstrukten. Werden beispielsweise im Kontext des Pthread-Standards Programme mit einer Vielzahl von Threads oftmals als Indiz schlechten Softwaredesigns angesehen, so befindet sich diese Klasse von Software ausdrücklich im Zielkreis moderner Java-Implementierungen.

Notwendigerweise muß eine Abstraktion der wenigen konkreten Ausführungseinheiten der Maschine hin zu der Vielzahl oftmals kurzlebiger Threads statt-

finden. Dies geschieht am effizientesten durch eine mehrstufige Architektur mit einer ausdrucksstarken Schnittstelle zwischen Betriebssystemkern und Anwendungsprogramm.

Während einige Betriebssysteme eine solche Architektur bereits in Form von Bibliotheken realisieren, gilt dies für mehrere wichtige Plattformen nicht; die fehlende Scheduler-Komponente der Anwendungsschicht sollte hier durch die VM implementiert werden. Eine kommerzielle Java-Implementierung demonstriert eindrucksvoll die Skalierbarkeit dieses Ansatzes in einem threadlastigen Benchmark[4].

Die Entwicklung eines VM-eigenen auf Anwendungsebene implementierten Schedulers bringt weitere Vorteile gegenüber der Nutzung generischer Systembibliotheken mit sich. Kontextwechsel nur an dedizierten Punkten zu vollziehen, verringert Unterbrechungen durch Garbage Collection auf ein Minimum. Es bleibt offen, ob diese Vorteile die Komplexität einer eigenen Scheduler-Komponente rechtfertigen. Unter Umständen werden die Thread-Bibliotheken der Betriebssysteme entsprechend erweiterte Schnittstellen aufweisen.

Der Implementierung von Synchronisierungsoperationen gebührt besondere Aufmerksamkeit. Dem bedarfsgesteuerten Einsatz von Synchronisierungsobjekten wie „Pthread Mutex“ oder „Smalltalk Semaphoren“ steht die Ubiquität von Java Monitoren entgegen. Außerdem entstehen durch Bibliotheksabstraktionen viele unnötige Synchonisierungsoperationen.

Jedes Objekt mit einer vollständigen Monitorimplementierung auszustatten, verbietet sich aus Platzgründen. Eine externe Zuordnung durch Tabellen hat Effizienzprobleme. Eine einfache Monitorimplementierung auf Basis von Systemabstraktionen scheidet daher aus. Vielmehr sind platzeffiziente Kodierungsschemata innerhalb der Objektrepräsentation zu finden. Insbesondere das überwiegende Auftreten konfliktfreier Monitore mit maximal einem Erwerber bietet Raum für Optimierungen. Die bimodale Nutzung des Objektheaders etabliert sich hier als Standard-Algorithmus.

In Hinsicht auf die aufgezeigten Implementierungsoptionen stehen die Java Synchronisierungsoperationen zu Unrecht in der Kritik, ineffizient zu sein. Im Hinblick auf Optimierungen im Rahmen des Speichermodells wird deutlich, wie gefährlich der Versuch ist, an korrekter Synchronisierung „vorbeizuprogrammieren“.

Die Nutzung von Systemthreads für Nebenläufigkeit anstatt oder in Kombination mit einem im *user space* realisierten Scheduler hat eine bedeutsa-

[4]JRockit [Appeal, 2000]/Volano [Volano, 2000]

me Konsequenz: sie führt zu Nebenläufigkeit innerhalb des Laufzeitsystems. Folglich sind Dienste wie die Speicherverwaltung thread-sicher zu implementieren; trotzdem darf ein derart stark frequentierter Dienst nicht übermäßig synchronisieren. Auch die sichere Manipulation generierten Maschinencodes durch den Übersetzer ist eine Herausforderung im Kontext von Multiprozessoren und Instruktions-Caches. Diese Umstände stellen einen wichtigen Grund für die Komplexität moderner virtueller Maschinen dar. Angesichts der Tatsache, daß Uniprozessoren weiterhin eine wichtige Plattform für Java-Software darstellen, stellt sich die Frage, ob dort nicht alternative Architekturen wie z.B. die der VisualWorks Smalltalk VM zu erwägen sind.

Kapitel 6

Zusammenfassung und Ausblick

Durch die Relevanz von Java ist die Implementierung virtueller Maschinen wieder zu einem Gegenstand aktiver Forschungsaktivitäten geworden. Dabei werden bekannte Techniken erweitert und an einen neuen Kontext angepaßt als auch neue Konzepte entwickelt und erprobt.

Diese Arbeit gibt einen Überblick über die Architektur virtueller Maschinen, speziell im Hinblick auf Java. Die drei Hauptkomponenten „Programmausführung“, „Speicherverwaltung“ und „Thread-System“ werden auf ihre Aufgabe und die Techniken ihrer Implementierung hin analysiert. Dabei wird eine enge Verzahnung dieser Komponenten offenbar.

Der dynamische Compiler bedient sich der Speicherverwaltung bei der Allokation der von ihm benötigten Strukturen wie Klassen- und Methodenobjekten. Im Gegenzug erfüllt er wichtige Aufgaben wie die *Inline*-Expansion von Allokationsoperationen, „Read Barriers“ und „Write Barriers“ für generationale und nebenläufige Garbage Collection sowie die Generierung von Referenztabellen für exakte Garbage Collection.

Mit dem Thread-System kooperiert der Compiler, indem er Synchronisations- und Unterbrechungscode erzeugt. Außerdem identifiziert er die Programmpunkte, an denen sich der ausführende Thread in einem konsistenten Zustand befindet, d.h. wenn Referenztabellen zur Identifikation der Inhalte von Registern und Stack vorliegen.

Die Speicherverwaltung muß mit dem Thread-System zusammenarbeiten, um thread-lokale Allokationspuffer mit den Ausführungskontexten zu assoziieren. Zur Einleitung einer Garbage Collection muß das Thread-System alle Threads an konsistenten Punkten stoppen.

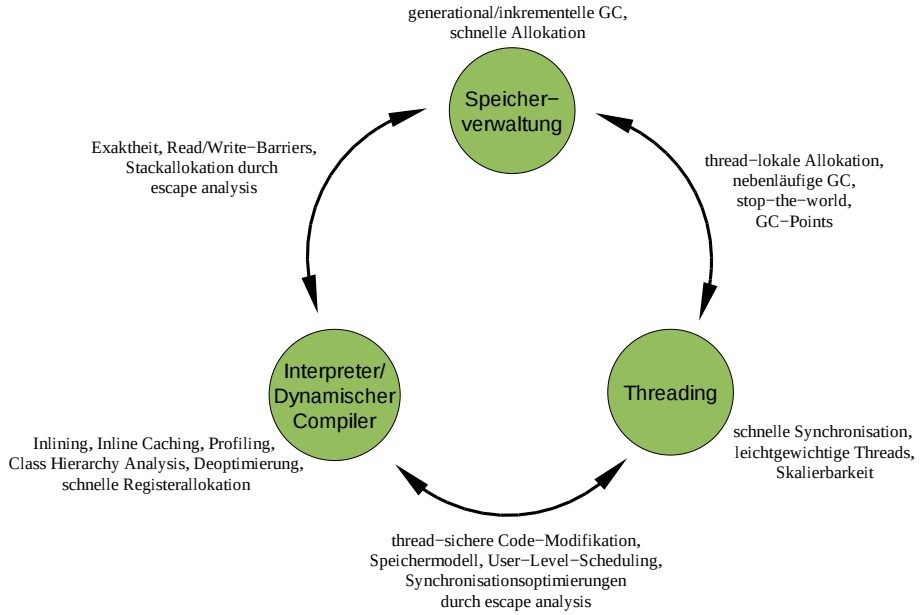

Abbildung 6.1: Die drei Hauptkomponenten einer virtuellen Maschine

Weiterhin wird in dieser Arbeit beschrieben, welche Techniken dem aktuellen Stand verfügbarer Implementierungen entsprechen. Diese stellen nach einfachen Interpretern und Maschinen mit nicht-adaptiven Compilern die dritte Generation in der Entwicklung dar. Es wird ersichtlich, durch welche Komponenten sie sich auszeichnen:

- Die Ausführung findet durch gemeinsamen Einsatz eines Interpreters und eines optimierenden Compilers oder durch ein Paar von schnellem und optimierendem Compiler statt. Die Optimierung wird durch dynamisch gewonnene Profildaten gesteuert.

  Dynamische Methodenaufrufe werden zum einen durch *inline caches* sowie durch statische Aufrufbindung in Verbindung mit Methodenintegration beschleunigt. Die hierzu erforderliche Empfängertypvorhersage findet aufgrund statischer Typinformationen sowie begrenzter globaler Analysen statt; um das volle Potential dieser Optimierungen ausschöpfen zu können, muß die dynamische Deoptimierung von Methoden vorgesehen sein.

  Optimierungen auf niedrigerer Ebene werden mit Hilfe klassischer Optimierungstechniken vollzogen, wobei zunehmend das Kosten/Nutzen-Verhältnis der verwendeten Compileralgorithmen in den Mittelpunkt rückt.

- Die Speicherbereinigung basiert auf wohlverstandenen Technik der generationalen Garbage Collection mit zwei Generationen. Dabei wird entweder in der jüngsten Generation ein separater Bereich („Eden“) zur Allokation reserviert, oder es kommen thread-lokale Allokationsbuffer zum Einsatz.

  Weiterhin wird verstärkt auf den „Train“-Algorithmus für die inkrementelle Bereinigung der alten Generation zurückgegriffen.

- Die Thread-Implementierungen setzen vorwiegend auf Bibliotheken des Betriebssystems auf, die die Ausführung auf mehreren Prozessoren erlauben. Einige Maschinen realisieren zweistufiges Scheduling statt dessen durch eigene Implementierungen, um höhere Skalierbarkeit zu erzielen.

  Die Synchronisationskonstrukte werden auf Basis atomarer Prozessorinstruktionen direkt in der VM implementiert, da die Verwendung von Systemkonstrukten aus Platzgründen nicht möglich ist.

Die Arbeit zeigt weiterhin auf, welche Bereiche Gegenstand aktiver Forschung sind und Fortschritte erwarten lassen. Hierzu zählen folgende Punkte:

- Die unoptimierte Ausführungsphase durch einen Interpreter wird durch einen schnellen, schwach optimierenden Compiler ersetzt werden. Der erzeugte Maschinencode läuft schneller ab als interpretierter Code bei sehr geringen Übersetzungskosten.

- Für die optimierenden Compiler werden zunehmend Algorithmen entwickelt werden, die schneller ablaufen und dabei nahezu gleich effizienten Code erzeugen wie die zur Zeit gebräuchlichen.

- Globale Analyseverfahren wie *escape analysis* werden zunehmend an Bedeutung gewinnen.

- Die adaptive Steuerung der dynamischen Compilerkomponente beruht zur Zeit auf einfachen Heuristiken. Die verwendeten Verfahren sind schwer vergleichbar und oftmals eng an die VM-Architektur gekoppelt. Auf diesem Gebiet ist eine weitere Formalisierung in Form von Standardalgorithmen und -darstellungen zu erwarten.

- Architekturen mit vielen Prozessoren stellen den klassischen „stop-the-world“-Ansatz der Garbage Collection infrage. Das Anhalten aller Threads auf einem System mit mehr als 8 Prozessoren wird zunehmend unwirtschaftlich; für diesen Einsatz sind daher nebenläufige

und parallele Algorithmen zur Speicherbereinigung notwendig. Zur Zeit existieren keine Algorithmen, die die Ansprüche der Praxistauglichkeit erfüllen können, jedoch befinden sich zur Zeit mehrere vielversprechende Ansätze in der Entwicklung.

In Bezug auf den Entwicklungsprozess virtueller Maschinen ist zu erwarten, daß die Schnittstellen der Komponenten klarer dokumentiert werden. Ansätze hierzu lassen sich beispielsweise im Garbage Collection Interface der Sun Research VM [White and Garthwaite, 1998] erkennen. Hier wird versucht, die Verantwortlichkeiten und Abhängigkeiten der Speicherverwaltung in Form einer Programmschnittstelle festzuhalten.

Gelingt es, durch ein Bootstrapping die virtuelle Maschine in der Sprache selbst zu implementieren, kann somit auch bei der Entwicklung des Laufzeitsystems von den Vorteilen der Sprache Java profitiert werden. Insbesondere ist es auf diese Weise möglich, die VM selbst mit Hilfe dynamischer Profile zu optimieren. Eine solche Implementierung existiert bereits mit der Jalapeño VM.

Abschließend lässt sich sagen, daß moderne virtuelle Maschinen in ihrer Komplexität in den Bereich sehr großer Softwaresysteme wie Datenbanksysteme [Lockemann and Schmidt, 1987] und Betriebssysteme [Tanenbaum, 1992] gerückt sind. Sie übernehmen dabei zunehmend Aufgaben des Betriebssystems als Ausführungsumgebung für Anwendungsprogramme. Durch eine enge Kooperation ihrer Komponenten werden Ausführungsgeschwindigkeiten erreicht, die statisch kompilierten Sprachen in nichts nachstehen und sie sogar übertreffen können.

Anhang A

Implementierungen virtueller Maschinen

In diesem Anhang werden einzelne Implementierungen virtueller Maschinen verglichen. Dabei werden überwiegend virtuelle Java-Maschinen verglichen – lediglich die Self und Tycoon-2 sind als Referenz vertreten. Die Auswahl der dargestellten Java-Maschinen erfolgte zum einen nach ihrer Relevanz im aktuellen Markt von JVMs und zum anderen nach der Fortschrittlichkeit ihrer technischen Konzepte.

In den Tabellen A.1 bis A.3 werden die einzelnen Implementierungen zunächst in tabellarischer Form verglichen. In den folgenden Abschnitten werden daraufhin noch einzelne Besonderheiten genannter Implementierungen dargestellt.

Die betrachteten Punkte entsprechen den in dieser Arbeit identifizierten Hauptkomponenten Speicherverwaltung, Thread-System und Ausführungseinheit. Entsprechend werden bereits im Rahmen der Arbeit eingeführte Fachbegriffe nicht weiter erläutert.

A.1 Sun JDK 1.2.2 Reference Implementation

Diese Java Maschine ist eine Weiterentwicklung der ersten JVM 1.0.2. Sie stellte bis zur Version 1.2 die Standard-VM von Sun dar. Weiterhin wurde ihr Quellcode als Referenzimplementierung an Lizenznehmer weitergegeben.

Komponente	Sun JDK 1.2.2 Reference	Sun HotSpot 2.0 Server	Sun HotSpot 2.0 Client	Sun Research-VM[a]
Plattformen	Solaris, Win32, Linux-x86	Solaris, Win32, Linux-x86	Solaris, Win32, Linux-x86	Solaris
Garbage Collection	konservativer „Mark-Compact“ m. *handles*	generational (kopierend + inkrementell[b])	generational (kopierend + inkrementell[b])	generational (kopierend)[c]
Allokation	LAB, dann gesperrte lineare Suche	Eden mit CAS	Eden mit CAS	LAB dann CAS
Thread-Implementierung	*user space* oder OS[d]	OS[d]	OS[d]	OS[d]
Sperr-Implementierung	OS-Mutex, Monitorzuordnung durch Tabelle	stackalloziert, Zuordnung durch *header word displacement*	stackalloziert, Zuordnung durch *header word displacement*	Metalock-basiert, Zuordnung durch *header word displacement*
Ausführung	Interpreter, Compiler[e]	Interpreter, optimierender Compiler	Interpreter, schwächer optimierender Compiler mit geringeren Laufzeitkosten	Interpreter, Compiler, experimentell: unterschiedlich stark optimierende Compiler
Dynamische Optimierung	einfache Heuristiken	Profile durch Aufrufzähler, Methodenanalyse	Profile durch Aufrufzähler, Methodenanalyse	Optimierte Übersetzung einer Methode nach n-tem Aufruf

Abbildung A.1: Tabellarischer Vergleich verschiedener Implementierungen virtueller Maschinen.

[a] Sourcecode lag zur Analyse nicht vor; Aussagen stützen sich auf technische bzw. wissenschaftliche Publikationen.

[b] inkrementeller „Train“-Algorithmus für die alte Generation.

[c] andere Implementierungen durch GC-Abstraktion möglich.

[d] Die Betriebssysteme AIX und Solaris bieten 2-level Threadbibliotheken.

[e] unvollständige Angaben, da der Sourcecode des Compilers zur Analyse nicht vorlag.

Komponente	IBM J2SE JDK 1.3 Production[a]	IBM Jalapeño[a]	Appeal JRockit 1.1.2[a]	Intel ORP
Plattformen	AIX, Win32, OS/2, OS/390, Linux-x86	AIX	Win32, Linux-x86	Win32, Linux-x86
Garbage Collector	generational (kopierend)	mehrere (generational, inkrementell, kopierend)	generational (incrementell[b])	generational (kopierend mit *step* + inkrementell[b])
Allokation	LAB		CAS	LAB
Thread-Implementierung	OS[c]	Zweistufig mit VM-eigenem Scheduler im *user space*	Zweistufig mit VM-eigenem Scheduler im *user space*	OS
Lock-Implementierung	*thin locks*	*thin locks*	*thin locks*	*thin locks* mit Metasperrenprotokoll
Ausführung	Interpreter, optimierender Compiler	kein Interpreter, 3 unterschiedlich optimierende Compiler	kein Interpreter, unterschiedlich optimierende Compiler	kein Interpreter, 2 unterschiedlich optimierende Compiler
Dynamische Optimierung		Profiling-Infrastruktur		Profile durch Aufrufzähler, Methodenanalyse

Abbildung A.2: Tabellarischer Vergleich verschiedener Implementierungen virtueller Maschinen.

[a]Quellcode lag zur Analyse nicht vor; Aussagen stützen sich auf technische bzw. wissenschaftliche Publikationen.

[b]inkrementeller „Train"-Algorithmus für alte Generation.

[c]Die Betriebssysteme AIX und Solaris bieten zweistufige Thread-Bibliotheken.

Komponente	Tycoon-2	Self 4.1.2
Plattformen	Linux, Solaris, HP-UX, Win32	Solaris-SPARC, Mac-OS
Garbage Collector	kopierend	generational
Allokation	linear mit Speichersperre	linear, nicht thread-safe
Thread-Implementierung	OS	Scheduler im *user space*
Sperr-Implementierung	OS-Mutex im Objekt, nur für Exemplare designierter Klassen	Implementierung im *user space*
Ausführung	Interpreter	kein Interpreter, 2 unterschiedlich optimierende Compiler
Dynamische Optimierung	Globaler Methoden-Cache	Profildaten aus PIC

Abbildung A.3: Tabellarischer Vergleich verschiedener Implementierungen virtueller Maschinen.

Die Speicherverwaltung basiert noch auf Handles anstatt direkter Referenzen zwischen Objekten. Die Markbits eines Objektes sind nicht im Objektheader sondern in einem separaten Array untergebracht.

Als Garbage Collector wird ein konservativer Mark and Sweep ohne Freelist verwendet. Wird neuer Speicher aus dem globalen Heap benötigt, findet ein lineare Heap-Scan statt. Kann kein freier Speicherbereich von ausreichender Größe gefunden werden, so wird eine Mark-Compact Collection ausgeführt. Die Compaction ist dank der Verwendung von Handles möglich, obwohl der Kollektor nicht exakt ist.

Um *contention* bei der Allokation zu vermeiden wird ein *local allocation buffer* sowohl für den allozierten Speicher als auch für die notwendigen Handles verwendet. Die Größe des LAB beträgt 1K pro Thread, einen dynamische Anpassung dieses Wertes an die Anzahl der aktiven Threads findet nicht statt.

Monitore werden unter Verwendung der durch die Thread-Bibliothek bereitgestellten Mutexe und Condition-Variablen implementiert. Die Verbindung von Objekt zum zugehörigen Monitor erfolgt durch eine Hashtabelle – als Schlüssel wird die Adresse des zum Objekt gehörenden Handles verwendet.

Die Ausführung findet sowohl durch eine assembler-codierte Interpreterschleife als auch durch einen dynamischen Compiler statt. Der Sourcecode des Compilers lag uns jedoch nicht vor – ohne diesen läßt sich lediglich aussagen, daß ein Inlining von Synchronisations- oder Allokations-Code aufgrund des schwachen Designs nicht möglich ist.

Die JDK 1.2.2 Referenzimplementierung stellt eine ausgereifte JVM der ersten Generation dar. Designentscheidungen wie handle-basierte Speicherverwaltung oder tabellenbasierte Monitorzuordnung limitieren die erzielbare Effizienz. Der Schwerpunkt der Referenzimplementierung lag berechtigterweise auf Portabilität und Einfachheit, nicht auf Effizienz.

A.2 Sun Hotspot 2.0

Die HotSpot VM ist eine komplette Neuentwicklung einer JVM, ohne auf eine vorhandene Java-Maschinen aufzusetzen. Das Design ist eine konsequente Fortentwicklung der Self- und Animorphic Smalltalk Maschinen[1].

Zur Garbage Collection wird ein exakter generationaler Algorithmus mit zwei

[1]Dies ist nicht zuletzt eine Folge des beteiligten Teams.

Generationen verwendet.

- Die junge Generation hat eine variable Größe von bis zu 2,5 MB und ist in "Eden" (80%) und "Survivor-Space" (20%) unterteilt. Zur ihrer Bereinigung wird "Copying Collection" verwendet.
- Die alte Generation wird mit dem inkrementellen "Train"-Algorithmus bereinigt. Bevor der Heap vergrößert wird, wird eine vollständige "Mark-Compact" Garbage Collection mit Hilfe einer "Break-Table" ausgeführt.
- Es existiert ein zusätzlicher Bereich, der sogenannte "Permanent Object Space", in dem Klassen und Methoden abgelegt werden. Dieser Bereich wird nur bei einer komplette Garbage Collection bereinigt.

Um Zeiger zwischen Generationen zu protokollieren, wird "Card Marking" mit einer Kartengröße von 512 Bytes verwendet. Zur thread-sicheren Allokation wird die globale Freispeichergrenze durch atomares "Compare-And-Swap" inkrementiert. Es finden keine thread-lokalen Allokationsbuffer Verwendung.

Die Thread-Implementierung der HotSpot-Maschine stützt sich auf die Betriebssystem-Threads der jeweiligen Platform; die Monitore sind hingegen in der Maschine selbst implementiert. Zum schnelleren rekursiven Erwerb und zur schnelleren Allokation werden Monitore zunächst auf dem Stack angelegt – erst bei Erwerbungskonflikten werden sie auf den Heap befördert.

Der Bytecode wird durch einen in Assembler geschriebene Interpreter und durch einen optimierenden Compiler ausgeführt. Dabei wird der Interpreter erst beim Starten der VM durch einen Makro-Assembler generiert. Dies erlaubt eine optimale Anpassung an die Platform (CPU-Version, OS-Version) und das *inlining* bestimmter Speicheradressen, die im Verlauf der Sitzung konstant sind (z.B. die Obergrenze des Allokationsbereiches "Eden"). Die Entscheidung, ob und wann eine Methode übersetzt wird, wird dynamisch durch Profilinformationen entschieden.

Der Compiler trifft teilweise optimistische Annahmen, die durch späteres dynamisches Laden von Klassendefinitionen ungültig werden. Aus diesem Grunde unterstützt die Maschine das De-Optimieren laufender Aktivierungen optimierter Methoden.

Hotspot wird in zwei Ausführungen ausgeliefert:

1. In einer *Client*-Version, die für die Ausführung von kurzlebigeren, interaktiven GUI-Applikationen eingesetzt werden soll.

2. In einer Server-Version, die für den Einsatz von langlebigen Serverapplikationen optimiert wurde.

Der Unterschied zwischen den beiden Versionen findet sich hauptsächlich im eingesetzten Compiler. Der Compiler der *Client*-Version ist auf einen interaktiven Einsatzt optimiert, d.h., daß die Übersetztgszeit möglichst kurz gehalten wird – auf der anderen Seite ist der erzeugte Code in der Ausführung ineffizienter als der des Server-Compilers.

Der optimierende Compiler der HotSpot Server VM arbeitet mit klassischen Compileralgorithmen, u.a. Register-Allokation durch *graph coloring*[Chaitin, 1982], CSE durch *global value numbering* sowie *inline caches* — der Einsatz von *polymorphic inline caches* hat sich jedoch nicht bewährt[Bracha, 1999], daher werden polymorphe Aufrufstellen durch *vtables* implementiert.

A.3 IBM Jalapeño

Die Jalapeño VM, die am IBM T.J.Watson Research Laboratory entwickelt wird, wurde für Serveranwendungen entworfen. Jalapeño läuft auf PowerPC Multiprozessorrechnern unter AIX, ist allerdings nicht öffentlich verfügbar.

Eine Besonderheit der Jalapeño VM ist die Tatsache, daß sie nahezu vollständig in Java implementiert wurde, inklusive der Speicherverwaltung, des Schedulers und der dynamischen Compiler. In einer anderen "Gast"-Maschine kann Jalapeño einen Bootstrap anfertigen und in einen Speicherabzug sichern, der dann lauffähig ist. Insbesondere können im folgenden Teile der virtuellen Maschine selbst durch den dynamischen Compiler optimiert werden!

Für Jalapeño existiert eine Zahl unterschiedlicher Implementierungen von Garbage Collectoren, darunter inkrementelle sowie generationale. Natürlich kann eine Konfiguration nur einen Algorithmus anwenden.

Das Thread-System ist zweistufig. Als primäre Ausführungskontexte werden mehrere AIX LWP — einer pro Prozessor — verwendet, auf die der VM-Scheduler die Java-Threads verteilt. Kontextwechsel finden dabei nur an konsistenten Punkten statt und ermöglichen eine schnelle Transition zur Garbage Collection.

Jalapeño ist eine Mehrcompilermaschine ohne Interpreter. Jede Methode wird vom schwach optimierenden „Baseline"-Compiler übersetzt. Zur Lauf-

zeit gesammelte Profilinformationen steuern die Neuübersetzung mit einem der optimierenden Compiler. Der höchst optimierende Compiler wendet dabei eine Vielzahl von Optimierungen an, darunter profilgesteuerte Methodenintegration, *escape analysis*, Kontrollflußoptimierungen der Ausnahmebehandlung bis hin zu klassischen SSA-basierten Optimierungen. Die Registerallokation wird aus Laufzeitgründen durch einen linearen Algorithmus erzielt, der klassischen, "teuren" Algorithmen vergleichbaren Code erzeugt.

A.4 Appeal JRockit 1.1.2

JRockit ist eine kommerziell vertriebene JVM aus Schweden. Leider lag uns weder der Sourcecode noch eine wissenschaftliche Publikation zur Beschreibung der Maschine vor, so daß sich die hier formulierten Aussagen überwiegend auf Marketingaussagen stützen.

Die Maschine wurde mit dem Ziel hoher Skalierbarkeit auf Multiprozessorsystemen entwickelt. Da auf den Plattformen Win32 und Linux keine zweistufigen Threadbibliotheken zur Verfügung stehen, verfügt JRockit wie Jalapeño über einen eigenen Scheduler, der Java-Threads auf LWP abbildet. Die Monitore sind als *thin locks* implementiert.

Zur Garbage Collection wird ein generationaler Algorithmus eingesetzt, wobei die älteste Generation inkrementell durch den "Train"-Algorithmus bereinigt wird. Zur Allokation wird atomares „Compare-and-Swap“ verwendet.

Die Ausführung findet ohne Interpreter durch einen Compiler statt, dessen Grad der Optimierung variiert wird. Prinzipiell entspricht dies einem Modell der Mehrcompilermaschine.

JRockit stellt bisher die einzige virtuelle Maschine dar, mit der ein *2-level scheduling* auf den Plattformen Win32 und Linux erreicht werden kann. Daher skaliert sie im Bereich thread- und E/A-dominierter Serverapplikationen besser als die Konkurrenten, die allein den Scheduler des Betriebssystemkerns verwenden [Volano, 2000].

A.5 Intel Open Runtime Platform

Die Intel Open Runtime Platform ist die in den Intel Microprocessor Research Labs entwickelte Forschungsmaschine. Sie verfügt über eine definierte Schnittstelle zum Garbage Collector, die prinzipiell verschiedene GC-

Implementierungen ermöglicht. In der aktuellen (ersten) Version ist ein generationaler Algorithmus mit 2 Generationen und einem „Large Object Space“ implementiert worden.

- Die jüngere Generation wird durch einen „Copying Collection“-Algorithmus bereinigt, der das Alter der Objekte durch eine Segmentierung der Generation abbildet. Ein Segment (*step*) besteht wiederum aus mehreren Blöcken konstanter Größe. Der Vorteil dieses Algorithmus gegenüber herkömmlicher „Copying Collection“ besteht darin, daß in der Regel weniger Platz benötigt wird.
- Die alte Generation wird den inkrementellen „Train“-Algorithmus bereinigt.
- Objekte, die größer als 64KB sind, werden im „Large Object Space“ (LOS) angelegt. Dieser Bereich wird ähnlich wie der „Permanent Object Space“ der HotSpot-Maschine durch einen „Mark-Compact“-Algorithmus bereinigt.

Als „Write Barrier“ zum Protokollieren von Zeigern zwischen Generationen findet „Card Marking“ mit eine Kartengröße von 256 Bytes Verwendung.

Die Allokation in der jungen Generation findet in einem LAB einer Größe zwischen 64KB und 256KB statt.

Die Thread-Implementierung basiert auf Betriebssytem-Threads; für die Monitore wird einen Kombination aus *thin locks* und einem Metalockprotokoll verwendet.

Zur Ausführung verwendet ORP zwei Compiler aber keinen Interpreter. Wie bei Jalapeño wird der Bytecode zunächst von einem Basis-Compiler übersetzt. Gesteuert durch dynamische Profilinformationen wird ggf. die Neuübersetzung einer Methode durch den optimierenden Compiler ausgelöst.

Es gibt keine feste Rahmendarstellung, jeder Compiler kann eine eigene Repräsentation bestimmen. Muß eine Methode aufgrund einer Ausnahme verlassen werden, so wird der Compiler, der die betreffende Methode compiliert hat, aufgefordert, die aktuelle Aktivierung abzubauen (*unwind*). Für die Garbage Collection identifiziert der entsprechende Compiler die Referenzen auf dem Stack und in den Registern. Referenztabellen kann jeder Compiler also in dem von ihm bevorzugten Format ablegen.

Der Basis-Compiler nutzt diese Flexibilität, indem bei der Übersetzung keine Referenztabellen generiert werden, statt dessen werden diese bei Bedarf

berechnet. Der optimierende Compiler legt die Referenztabellen hingegen in komprimierter Form für praktisch jede Instruktion ab. Hiervon ausgenommen sind nur Instruktionsfolgen, die die Speicherverwaltung in einen inkonsistenten Zwischenzustand bringen[2]. Durch die Bereitstellung von Referenztabellen für praktisch jede Instruktion sind damit jederzeit nur wenige Threads inkonsistent — bei einer Garbage Collection müssen daher nur wenige Threads vorgerollt werden.

A.6 Tycoon-2

Das Tycoon-2-System ist ein an der Universität Hamburg entwickeltes persistentes Objektsystem. Die Programmierung der Tycoon-2-Systems erfolgt in der rein objektorientierten Sprache TL-2 [Gawecki and Wienberg, 1998]. Die gut dokumentierte virtuelle Maschine [Weikard, 1998] ist mit dem Ziel der Portabilität und Einfachheit entwickelt worden und nimmt daher nur wenige Optimierungen vor.

Zur Garbage Collection wird „Copying Collection“ eingesetzt; die Allokation erfolgt durch Erwerb einer globalen Speichersperre und anschließender Erhöhung des Freispeicherzeigers.

Die Thread-Implementierung stützt sich auf die Bibliotheken des jeweiligen Betriebssystems. Zur Implementierung der Synchronisationsobjekte wird ebenfalls auf die Funktionalität der Plattform zurückgegriffen. Anders als bei Java stellt in Tycoon nicht jedes Objekt automatisch einen Monitor dar, so daß ein direkter Verweis vom jeweiligen Objekt auf das verwendete Synchronisationsobjekt unproblematisch ist.

Die Ausführung von Bytecode erfolgt lediglich durch einen Interpreter. Der Quellcode-Compiler führt bereits Optimierung von Endrekursion und einfache Konstantenfaltung aus, die virtuelle Maschine selbst verwaltet einen zentralen Cache für die Methodensuche — andere Optimierungen werden nicht vorgenommen.

Insgesamt stellt die Tycoon-2-VM eine einfache, gut verständliche Implementierung dar, deren Schwerpunkt auf Portabilität, nicht aber auf Ablaufgeschwindigkeit liegt.

[2]Beispielsweise muß das Speichern einer Referenz und die anschließende Markierung ihrer Karte aus Sicht des Garbage Collectors atomar ausgeführt werden.

A.7 SELF 4.1.2

Das Self-System ist der "Pionier" der dynamisch optimierenden virtuellen Maschinen. Viele Techniken der dynamischen Compiler wie beispielsweise PIC und OSR wurden im Rahmen des Self-Projektes entwickelt. Die Sun HotSpot-VM ist stark vom Self-Projekt beeinflußt, da sie in großen Teilen von "Self-Veteranen" entwickelt wurde.

Die Programmierung des Self-Systems erfolgt in der objektbasierten Sprache Self. Im Unterschied zu klassenbasierten Sprachen existieren in Self keine Klassen — hingegen werden neue Objekte als *clone* eines anderen Objektes erzeugt. Dieser gegenüber herkömmlichen klassenbasierten Sprachen leicht veränderte Ansatz hat aber nur geringe Auswirkungen auf die Optimierungen, die eine virtuelle Maschine vornehmen kann. Prinzipiell können in objekt- und klassenbasierten Systemen die gleichen Techniken angewendet werden.

Das Self-System verwendet einen generationalen Garbage Collector mit zwei Generationen und einem "Eden-Space" für die Allokation. Zur Protokollierung von Referenzen zwischen Generationen wird „Card-Marking“ mit einer Kartengröße von 128 Bytes verwendet.

Das Self-System verfügt über keinen Interpreter — hingegen werden ein nichtoptimierender und ein optimierender Compiler verwendet. Letzterer führt Optimierungen wie *inlining, customization, splitting, copy propagation und dead code elimination* durch. Die Registerallokation findet mit Hilfe eines einfachen Verwendungszählers anstatt eines graphen-basierten Algorithmus verwendet. Zur Beschleunigung des dynamischen Methodenaufrufs werden *polymorphic inline caches* verwendet, die gleichzeitig zur Profilerstellung herangezogen werden.

Anhang B

Glossar

Aktivierung: Der Speicherbereich innerhalb des →*Ausführungs-Stacks*, der einem bestimmten Aufruf einer Funktion entspricht. Er enthält die Parameter und lokalen Variablen der Funktion, sowie einen Verweis auf die vorherige Aktivierung und sprachspezifische Daten.

Algebraische Vereinfachungen: Klassischen Optimierung, bei der algebraische Ausdrücke durch Umformungen vereinfacht werden. Dabei werden beispielsweise Multiplikationen mit 2er-Pozentzen durch Verschiebe-Operationen ersetzt.

Alignment: Die Ausrichtung einer Adresse im Speicher. Z.B. ist eine Adresse *32-bit-aligned*, wenn sie sich ohne Rest durch 4 Teilen läßt (32 Bit entsprechen 4 Byte, Hauptspeicheradressen werden in Byte geführt).

Atomare Instruktion: Komplexe Speicheroperation, die von der Architektur atomar ausgeführt wird. Beispiele sind „Test-and-Set“ oder „Compare-and-Swap“.

Ausführungs-Stack: Speicherbereich, in dem Paramerter und lokale Variablen beim Aufruf einer Funktion im Falle der Stack-Allokation abgelegt werden.

Bytecode: Serialisierte Instruktionsfolge für eine abstrakte Maschine in binärer Form.

CAS: („Compare-and-Swap“) →*Atomare Instruktion*, bei der ein Vergleich eines Registers mit einer Hauptspeicherzelle und bei Gleichheit ein Vertauschen der Inhalte atomar ausgeführt wird.

CHA: →*Class Hierarchy Analysis*

CSE: →*Common Subexpression Elimination*

Card Marking: Methode um Zeiger zwischen Generationen zu protokollieren. Der Speicher wird in n Karten gleicher Größe unterteilt und ein Bit-Array *cards* der Größe n wird angelegt. Wird ein Zeiger geschrieben, so wird die entsprechende Karte markiert, indem $cards[i]$ gesetzt wird. Alle Pointer auf der modifizierten Karten werden zu Wurzeln jeder Garbage Collection einer jüngeren Generation →*Remembered Sets*.

Class Hierarchy Analysis: Globale Programmanalyse des dynamischen Compilers bezüglich der Klassenhierarchie. So können Schlüsse über die möglichen Ziele eines dynamischen Methodenaufrufs gezogen werden.

Common Subexpression Elimination: Klassische Optimierung, durch die die mehrfache Auswertung gleicher Ausdrücke oder Teilausdrücke vermieden wird. Wird beispielsweise an zwei Stellen in einer Prozedur der Ausdruck `a + 1` berechnet und ist `a` an beiden Stellen gleich (durch Datenflußanlayse vom Optimierer zu beweisen), so kann einer Speicherung des Ergebnisses in einer Temporärvariablen erfolgen, um einen Wiederverwendung zu ermöglichen. Da eine Integer-Operation auf modernen Prozessoren scheller ist als ein Speichzugriff, lohnt sich die CSE nur wenn die erzeugte Teporärvariable einem Register zugewiesen werden kann. Andernfalls wäre eine erneute Berechnung meist schneller als der notwendige Speicherzugriff.

Constant Folding and Propagation: Klassischen Optimierung, bei der Ausdrücke, deren Operanden zur Übersetzungszeit konstant sind, bereits vom Compiler ausgewertet werden. Dabei entstehen neue Konstanten, die im Code weiterpropagiert werden und ggf. wieder mit anderen Konstanten verknüpft werden. Der Compiler nimmt also einen Teil der Berechnung des laufenden Programmes voraus, indem er die Zielumgebung simuliert — dies ist im Zusammenhang mit Fließkommazahlen nicht unproblematisch.

Control Path Dependency: Begriff aus der Datenflußanalyse: Ist der Wert einer Datenflußvariablen an einem bestimmten Punkt von dem Weg abhängig, auf dem dieser Punkt erreicht wurde, so besteht eine *control path dependency*

Control Point Dependency: Begriff aus der Datenflußanalyse: Ist der Wert einer Datenflußvariablen an einem bestimmten Punkt immer gleich, und ist die damit unabhängig vom Kontrollfluß, so besteht eine *control point dependency*

Deoptimierung: Zurücknehmen bestimmter Optimierungen beim Übersetzen von Methoden; meist durch komplette Neuübersetzung. Notwendig, wenn zuvor getroffene Annahmen z.B. durch dynamisches Laden neuer Klassen nicht länger zutreffen. Erfordert u.U. →*On-Stack Replacement*.

Dynamischer Compiler: (*dynamic compiler, just-in-time compiler*) Compiler, der im laufenden Programm abstrakten Code in Maschinencode übersetzt und so das Programm erweitert oder verändert. Zur Portabilität oder späten Optimierung aufgrund von dynamischem Programmverhalten oder Maschinenspezifika.

Dynamischer Methodenaufruf: (*dynamic dispatch*) Mechanismus objektorientierter Sprachen, die zu Empfängerklasse und Nachricht zugeordnete Methode zu ermitteln und aufzurufen. →*inline cache, polymorphic inline cache, vtable*

Eden: Begriff aus der →*generationalen Garbage Collection*: Fester Bereich innerhalb der jüngsten Generation, in dem neue Objekte alloziert werden. Mit der nächsten Garbage Collection werden sie in den Hauptbereich der jüngsten Generation evakuiert.

Ereignisvariable: (*condition variable*) Synchronisierungskonstrukt. Ereignisvariablen erlauben den Austausch anonymer Ereignisse zwischen →*Threads*.

Exakter Garbage Collector: Kann ein Garbage Collection für jede Speicherzelle im System zum Zeitpunkt der Garbage Collection bestimmen, ob sie einen Zeiger oder einen Skalar (z.B. eine Ganzzahl) enthält, so wird er als exakt bezeichet. Ein exakter Garbage Collector kann im Gegensatz zu einem →*konservativen Garbage Collector* Objekte auf den Heap verschieben.

Fast Path: Der schnellste "Weg", den eine Berechnung nehmen kann. Oftmals stellt dieser den Regelfall dar, so daß sich besondere Optimierung lohnt. Der *fast path* einer Allokation kann beispielsweise lediglich im Erhöhen eines Zeigers bestehen, während der *slow path* eine Garbage Collection ausführt.

GC Point: (konsistenter Punkt) Programmpunkt, an dem eine Garbage Collection durchgeführt werden darf, weil hier →*Referenztabellen* vorliegen. Alle Threads müssen zu ihrem nächsten "GC Point" vorgerollt werden, wenn eine Garbage Collection durchgeführt werden soll.

Generationale Garbage Collection: Bei der generationalen Garbage Collection wird der Speicher logisch nach dem Alter der Objekte in mehrere Segmente unterteilt, die dann separat bereinigt werden können.

Gosling Property: Eigenschaft des Java-Bytecodes, die besagt, daß der Typ einer Programmvariablen keine →*control path dependency* aufweisen darf.

Header Word Displacement: Implementierungstechnik zur Reduktion des Speicherbedarfs von Objekten. Zur Aufnahme nur temporär benötigter Objektmetadaten werden Teile des →*Objektkopfes* durch Verweise auf zusätzliche Speicherbereiche ersetzt und dorthin verdrängt.

IC: →*inline cache*

Inkrementell Garbage Collection: Garbage Collection, bei der bei jedem Aufruf nur ein kleiner Teil des Speichers bearbeitet wird.

Inline Cache: (IC) Technik zum →*dynamischen Methodenaufruf*. An der Aufrufstelle in den Code eingebetteter einelementiger Ergebniscache des letzten Aufrufs; dargestellt als direkter Sprung an die Zielmethode in Verbindung mit einem Klassentest des Empfängerobjektes. Diese Technik beschleunigt quasi-monomorphe Methodenaufrufe, da diese auf einen Vergleich und einen direkten Sprung reduziert werden. Bei einem cache miss wird der IC überschrieben. →*polymorphic inline cache, vtable*

JIT: →*Dynamischer Compiler*

Kernel Space: (kernel mode) Privilegiertes Ausführungsregime von Systemcode. Es besteht ungeschützter Zugriff auf Systemressourcen. →*user space*

Konservativer Garbage Collector: Ist ein Garbage Collector nicht in der Lage, für jede Speicherzelle zu bestimmen, ob sie einen Pointer oder

einen Skalar (z.B. eine Ganzzahl) enthält, wird er als konservativ bezeichnet. Ein konservativer Garbage Collector muß (konservative) Annahmen über den Inhalt von Speicherzellen machen um eine Garbage Collection ausführen zu können.→*exakter Garbage Collector*

LAB: →*local allocation buffer*

Liveness: Um eine Speicherzelle wiederverwenden zu können, muß der Garbage Collector beweisen, daß diese vom laufenden Programm nicht mehr benötigt wird. Dieser Beweise wird i.d.R. aufgrund einer referenziellen Erreichbarkeit geführt. Kann eine Zelle vom Programm erreicht werden so wird sie als *live* bezeichnet

Local Allocation Buffer: (LAB) Speicherbereich eines Threads, der zur Ausführung von Allokationen ohne jegliche Synchronisation mit anderen Threads dient. Ist der *local allocation buffer* erschöpft, so wird synchronisiert ein neuer Bereich angefordert.

Loop Invariant Code Motion: Klassische Optimierung, bei der Schleifenkonstanten erkannt werden und ihre Auswertung vor die Schleife in den sogenannten *preheader* verlegt wird — dies reduziert den Rechenaufwand oft erheblich. Ob ein Prozeduraufruf eine Schleifenkonstante darstellt, kann nur durch interprozedurale Analysen festgestellt werden.

Loop Unrolling: Klassische Optimierung, bei der Schleifen „entrollt“ werden, indem der Schleifenkörper mehrfach hintereinander kopiert und dabei der Schleifenkontrollcode entsprechend angepaßt wird. Dies erlaubt wiederum eine bessere Verwendung von *delay slots* und vor allem eine optimalere Ausnutzung der Ausführungseinheiten superskalarer Prozessoren. *Loop unrolling* ist also eng mit →*software pipeling* verknüpft.

Markierung: (*tagging)* Methode zur Implementierung eines →*exakten Garbage Collectors*, bei der jeder Wert seine Typinformation selbst enthält. Üblicherweise wird ein Bit der Repräsentation verwendet, um die Typinformation aufzunehmen – z.B. stellt ein Wert, bei dem das *least significant bit* gesetzt ist, einen verschobenen Integerwert dar, ein Wert, bei dem das *least significant bit* nicht gesetzt ist, einen Zeiger. →*alignment*

Memory Barrier: Instruktion, die Speicherinstruktionen vor Optimierungen schützt. *Memory barriers* sind zur Implementierung des →*Speichermodells* notwendig.

Metasperre: Primitive Sperrstruktur, die ein komplexes Synchronisationsobjekt seinerseits vor nebenläufigem Zugriff schützt. Zumeist durch →*atomare Instruktionen* realisiert.

Methodenintegration: (*inlining*) Klassische Optimierung, die den Aufruf einer Methode durch ihren Methodenkörper ersetzt. Wie bei der →*tail-call optimization* werden zum eine die Kosten eines Aufrufes gespart und zum anderen weitere Optimierungen erst ermöglicht. Bei der Methodenintegration ist zu beachten, daß eine Vergößerung des Codes eine geringere Lokalität und damit *cache misses* im Instruktions-Cache bewirken kann.

Monitor: Synchronisierungskonstrukt. Einheit von →*Sperre* und →*Ereignisvariablen*.

Nebenläufig: Programmteile sind nebenläufig, wenn sie unabhängig voneinander und damit potentiell gleichzeitig ausgeführt werden können. Ein Programm ist nebenläufig, wenn es mehrere nebenläufige Programmteile enthält. →*parallel, Synchronisierung*

Nebenläufiger Garbage Collector: Garbage Collector, der →*nebenläufig* mit dem ausgeführten Programm in einem eigenen Thread abläuft.

OSR: →*on-stack replacement*

Objektkopf: Speicherbereich für Objektmetadaten wie z.B. Klasse oder Hashwert. Dieser liegt typischerweise direkt vor den "Nutzdaten" des Objekts im Speicher.

On-Stack Replacement: (OSR). Austausch einer oder mehrerer →*Aktivierungen* durch andere. Nötig, falls die aktive Methode durch den →*dynamischen Compiler* auf unterschiedliche Art und Weise neu übersetzt wurde. →*Deoptimierung*

PIC: →*polymorphic inline cache*

Parallel: Programmteile sind parallel, wenn sie zur gleichen Zeit auf verschiedenen Prozessoren ausgeführt werden.→*nebenläufig*

Paralleler Garbage Collector: Ein Algorithmus zur parallelen Garbage Collection ist nebenläufig formuliert. Die Garbage Collection kann daher auf mehreren Prozessoren →*parallel* ausgeführt werden.

Polymorphic Inline Cache: (PIC). Technik zum →*dynamischen Methodenaufruf*. Wie →*inline cache* in den Code eingebetteter Cache, allerdings mit mehreren (üblicherweise bis zu 10) Einträgen; dargestellt als kaskadierter Klassentest mit direkten Sprüngen. Beschleunigt schwach-polymorphe Aufrufstellen, eignet sich zudem zur Profilerstellung. →*inline cache, vtable*

Prozedurspezialisierung: Klassischen Optimierung. Kann der Compiler feststellen, daß einen bestimmte Prozedur `f(x)` häufig mit dem Parameter `x=3` aufgerufen wird, so kann einen Prozedur `f(x)/x=3`, erstellt werden. Durch →*constant folding* kann diese Prozedur — möglicherweise bis auf einen konstanten Wert — weiter vereinfacht werden.

Read Barrier: Codeabschnitt, der bei jedem Lesen aus einem Speicherbereich ausgeführt wird.

Referenzlokalität: Das Maß, in dem Zugriffe auf benachbarte Speicherzellen auch zeitlich dicht beieinander liegen.

Referenztabelle: Eine Beschreibung der Prozessorregister und des Stacks, die es dem Garbage Collector erlaubt, zu bestimmen, in welchen Registern und Stack-Zellen sich Zeiger befinden.

Registerallokation: Diese Optimierung ordnet den verwendeten symbolischen Registen physische Maschinenregister zu. Stehen diese nicht nicht in ausreichender Zahl zur Verfügung, so muß *spill code* eingefügt werden, der symbolische Register in den Hauptspeicher auslagert. Die Registerallokation stellt eine wichtige Optimierung dar, da die Zugriffe auf den Hauptspeicher durch die Ausnutzung der vorhandenen Maschinenregister minimiert werden.

Remembered Sets: Methode um Zeiger zwischen Generationen zu protokollieren. Bei jeder Zeigerzuweisung wird das Ziel der Zuweisung in eine Menge aufgenommen, die den Wurzeln jeder Garbage Collection einer jüngeren Generation zugefügt werden. →*Card Marking*

Sequential Store Buffer: Methode zur effizienten Implementierung von →*Remembered Sets*.

Skalare Ersetzung von Aggregaten: Klassische Optimierung bei der Aggregate durch die enthaltenen Skalare ersetzt werden. Beispielsweise wird bei einer Struktur `komplexeZahl` anstatt `z.real` direkt eine `float`-Variable verwendet, wenn dies möglich ist. Dies vermeidet zum

einen eine Indirektion und ermöglicht zum anderen weitere Optimierungen, bei denen Aggregate unberücksichtigt blieben.

Software Pipelining: Klassische Optimierung. Hier wird die Architektur moderner Prozessoren mit *pipelines* unterstützt, indem der Code derart umstrukturiert wird, daß *delay slots* von Verzweigungen ausgenutzt und die Latenzzeiten einzelner Operationen berücksichtigt werden. Außerdem werden bei superskalaren Prozessoren abhängige Operationen derart versetzt, daß mehrere Ausführungseinheiten ausgenutzt werden können.

Speichermodell: (*memory model*). Spezifikation der Operationen *load* und *store* auf dem globalen Speicher. Relevant für Multiprozessorarchitekturen und nebenläufige Programmiersprachen.

Sperre: (*mutex, lock*) Synchronisierungskonstrukt. Eine Sperre kann nur von einem →*Thread* zur Zeit erworben werden und realisiert so gegenseitigen Ausschluß.

Static Allocation Arena: Methode der optimierten Allokation, bei der der gesamte innerhalb eines *basic blocks* benötigte Speicher in einer Allokation am Anfang des Blocks angefordert wird.

Tail-Call Optimizations: Klassische Optimierung. Endet eine Prozedur mit dem Aufruf einer anderen oder mit einer Rekursion, so kann die *tail-call optimization* diese Aufrufe in Verzweigungen umwandeln. Es wird also zum Auswerten der gerufenen Prozedur die Aktivierung der aktuellen Methode verwendet. Dadurch werden zum einen die Kosten eines Aufrufes gespart, zum anderen werden Rekursionen zu Schleifen, die dann mittels Schleifenoptimierung weiter optimiert werden können.

Thread: Kontrollflußabstraktion. Ein Thread ist ein unabhängiger Ausführungskontext mit eigenem Aufrufstack, Instruktionszeiger und lokalem Speicher. Threads kommunizieren durch Synchronisierungskonstrukte und tauschen Daten über einen geteilten, globalen Speicher aus.

Tricolor Marking: Von Dijkstra formalisierte Abstraktion zur Betrachtung der Arbeit eines nebenläufigen Garbage Collectors.

User Space: Geschütztes Ausführungsregime von Anwendungscode. Zugriff auf Systemressourcen ist nur durch Aufruf des Betriebssytems und Wechsel in →*kernel space* möglich.

Vtable: (*virtual function table*). Vom Compiler generierte Tabelle der virtuellen Methoden einer Klasse. Jeder Methodensignatur wird eine Zeile der Tabelle zugewiesen. Gleiche Methodensignaturen in durch Vererbung abhängigen Klassen werden der gleichen Zeile zugewiesen. Dadurch kann der virtuelle Methodenlookup durch einen einfachen Array-Zugriff implementiert werden. →*inline cache, polymorphic inline cache*

Write Barrier: Codeabschnitt, der bei jedem Schreiben in einen Speicherbereich ausgeführt wird.

Literaturverzeichnis

[Adve and Gharachorloo, 1995] Adve, S. V. and Gharachorloo, K. (1995). Shared Memory Consistency Models. A Tutorial. Research Report 95/7, DEC Western Research Laboratory, Palo Alto, CA.

[Agesen, 1998] Agesen, O. (1998). GC Points in a Threaded Environment. Technical Report SMLI TR–98–70, Sun Microsystems Laboratories, Mountain View, CA.

[Agesen, 1999] Agesen, O. (1999). Personal Communication.

[Agesen and Detlefs, 1997] Agesen, O. and Detlefs, D. (1997). Finding References in Java Stacks. In Dickman, P. and Wilson, P. R., editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*.

[Agesen and Detlefs, 1999a] Agesen, O. and Detlefs, D. (1999a). Inlining of Virtual Methods. In *ECOOP '99 – Proceedings of the Annual European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag.

[Agesen and Detlefs, 1999b] Agesen, O. and Detlefs, D. (1999b). The Case for Multiple Compilers. `http://www.squeak.org/oopsla99_vmworkshop/`. In OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design.

[Agesen et al., 1998] Agesen, O., Detlefs, D., and Moss, J. E. B. (1998). Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In [PLDI, 1998], pages 269–279.

[Agesen et al., 1999] Agesen, O. et al. (1999). An Efficient Metalock for Implementing Ubiquitous Synchronisation. Technical Report SMLI TR–99–76, Sun Microsystems Laboratories, Mountain View, CA.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, Principles, Techniques, and Tools*. Addison-Wesley.

[Alpern et al., 2000] Alpern, B. et al. (2000). The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1).

[Anderson et al., 1997] Anderson, J. M. et al. (1997). Continuous Profiling: Where Have All the Cycles Gone ? In *Sixteenth ACM Symposium on Operating Systems Principles*, St Malo, France. ACM Press.

[Anderson et al., 1991] Anderson, T. E. et al. (1991). Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA. ACM Press.

[Appeal, 2000] Appeal (2000). The JRockit Virtual Machine. `http://www.appeal.se/machines/`.

[Appel, 1987] Appel, A. W. (1987). Garbage Collection Can Be Faster Than Stack Allocation. *Information Processing Letters*, 25(4):275–279.

[Appel, 1989a] Appel, A. W. (1989a). Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation*, 2:153–162.

[Appel, 1989b] Appel, A. W. (1989b). Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183.

[Appel et al., 1988] Appel, A. W., Ellis, J. R., and Li, K. (1988). Real-Time Concurrent Collection on Stock Multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20.

[Arnold et al., 2000a] Arnold, M. et al. (2000a). A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *2000 ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00)*, Boston, MA. ACM Press.

[Arnold et al., 2000b] Arnold, M. et al. (2000b). Adaptive Optimization in the Jalapeño JVM. In [OOPSLA, 2000].

[Arnold et al., 2000c] Arnold, M. et al. (2000c). An Empirical Study of Selective Optimization. In *Thirteenth Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Yorktown Heights, NY. Springer-Verlag.

[Bacon et al., 1998] Bacon, D. F. et al. (1998). Thin Locks: Featherweight Synchronisation for Java. In [PLDI, 1998].

[Baker, 1978] Baker, H. G. (1978). List Processing in Real-Time on a Serial Computer. *Communications of the ACM*, 21(4):280–94. Also AI Laboratory Working Paper 139, 1977.

[Bekkers and Cohen, 1992] Bekkers, Y. and Cohen, J., editors (1992). *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France. Springer-Verlag.

[Blanchet, 1999] Blanchet, B. (1999). Escape Analysis for Object Oriented Languages. Application to Java. In [OOPSLA, 1999], pages 20–34.

[Boehm et al., 1991] Boehm, H.-J., Demers, A. J., and Shenker, S. (1991). Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices*, 26(6):157–164.

[Bogda and Hölzle, 1999] Bogda, J. and Hölzle, U. (1999). Removing Unnecessary Synchronization in Java. In [OOPSLA, 1999].

[Bracha, 1999] Bracha, G. (1999). Personal Conversation.

[Broy, 1995] Broy, M. (1995). A Functional Specification of the Alpha AXP Shared Memory Model. SRC Research Report 136, DEC Systems Research Center, Palo Alto, CA.

[Brunck, 1994] Brunck, J. (1994). Efficient Message Passing Interface for Parallel Computing on Clusters of Workstations. Research Report RJ 9925, IBM.

[Bryant and Hartner, 2000] Bryant, R. and Hartner, B. (2000). Java, Threads and Scheduling in Linux. Patching the Kernel Scheduler for Better Java Performance. `http://www-4.ibm.com/software/developer/library/java2/`.

[Butenhof, 1997] Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522.

[Chaitin, 1982] Chaitin, G. J. (1982). Register Allocation and Spilling via Graph Coloring. *ACM SIGPLAN Notices*, 17(6):98–105.

[Chambers, 1992] Chambers, C. (1992). *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University.

[Cheney, 1970] Cheney, C. J. (1970). A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–8.

[Choi et al., 1999] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. (1999). Escape Analysis for Java. In [OOPSLA, 1999], pages 1–19.

[Cierniak et al., 2000] Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M. (2000). Practicing JUDO: Java under Dynamic Optimizations. In [PLDI, 2000].

[Cincom, 2000] Cincom (2000). Cincom's VisualWorks - Smalltalk Software. `http://www.cincom.com/visualworks/`.

[Cohen and Nicolau, 1983] Cohen, J. and Nicolau, A. (1983). Comparison of Compacting Algorithms for Garbage Collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553.

[Compaq, 1999] Compaq (1999). *Tru64 UNIX Guide to DECthreads*. Compaq Computer Corp. `http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V50_HTML/ARH9RATE/TITLE.HTM`.

[Dean, 1996] Dean, J. A. (1996). *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington.

[Demers et al., 1990] Demers, A., Weiser, M., Hayes, B., Bobrow, D. G., and Shenker, S. (1990). Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 261–269, San Francisco, CA. ACM Press.

[Dieckmann and Hölzle, 1998] Dieckmann, S. and Hölzle, U. (1998). A Study of the Allocation Behaviour of the SPECjvm98 Java Benchmarks. Technical Report TRCS98-33, Computer Science Department, University of California, Santa Barbara.

[Dijkstra et al., 1978] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. (1978). On-The-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):965–975.

[Doligez and Gonthier, 1994] Doligez, D. and Gonthier, G. (1994). Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press.

[Doligez and Leroy, 1993] Doligez, D. and Leroy, X. (1993). A Concurrent Generational Garbage Collector for a Multi-Threaded Implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press.

[Domany et al., 2000] Domany, T., Kolodner, E., and Petrank, E. (2000). A Generational On-the-Fly Garbage Collector for Java. In [PLDI, 2000].

[Driesen, 1999] Driesen, K. (1999). *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, University of California, Santa Barbara, CA.

[Driesen et al., 1995] Driesen, K., Hölzle, U., and Vitek, J. (1995). Message Dispatch on Pipelined Processors. In [ECOOP, 1995], pages 253–282.

[ECOOP, 1995] ECOOP (1995). *ECOOP '95 – Proceedings of the Annual European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag.

[Endo et al., 1997] Endo, T., Taura, K., and Yonezawa, A. (1997). A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. Technical report, Department of Information Science, University of Tokio.

[Ernst, 1998] Ernst, M. (1998). Typüberprüfung in einer Polymorphen Objektorientierten Programmiersprache. Analyse, Design und Implementierung eines Typprüfers für Tycoon-2. Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany.

[Franz, 1994] Franz, M. (1994). Compiler Optimizations Should Pay for Themselves. In *Advances in Modular Languages: Proceedings of the 1994 Joint Modular Languages Conference*. Universitätsverlag Ulm.

[Freund, 1998] Freund, S. S. (1998). The Costs and Benefits of Java Bytecode Subroutines. In *1998 ACM SIGPLAN Workshop on Formal Underpinnings of Java*, Vancouver. ACM Press.

[Gawecki, 1991] Gawecki, A. (1991). Ein Optimierender Übersetzer für Smalltalk unter Verwendung der Techniken der Empfängerspezifischen

Übersetzung und des Programmierens mit Fortsetzungen. Schriftenreihe Bericht 152/92, Universität Hamburg, Fachbereich Informatik.

[Gawecki and Wienberg, 1998] Gawecki, A. and Wienberg, A. (1998). Report on the Tycoon-2 Programming Language. Technical Report Version 1.0, Higher-Order GmbH.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[Gosling, 1995] Gosling, J. (1995). Java Intermediate Bytecodes. In *Proceedings of 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, ACM SIGPLAN Notices. ACM Press.

[Gosling et al., 1996] Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Addison-Wesley.

[Grove et al., 1997] Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call Graph Construction in Object-Oriented Languages. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Atlanta, GA. ACM Press.

[Gu et al., 2000] Gu, W. et al. (2000). The Evolution of a High-Performing Java Virtual Machine. *IBM Systems Journal*, 39(1).

[Gupta et al., 2000] Gupta, M., Choi, J.-D., and Hind, M. (2000). Optimizing Java Programs in the Presence of Exceptions. In *ECOOP 2000 – Proceedings of the Annual European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag.

[Hennessy and Patterson, 1996] Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, second edition.

[Herlihy, 1991] Herlihy, M. (1991). Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149.

[Hoare, 1974] Hoare, C. (1974). Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557.

[Hölzle, 1993] Hölzle, U. (1993). A Fast Write Barrier for Generational Garbage Collectors. In Moss, E., Wilson, P. R., and Zorn, B., editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*.

[Hölzle et al., 1991] Hölzle, U., Chambers, C., and Ungar, D. (1991). Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP '91 – Proceedings of the Annual European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag.

[Hölzle et al., 1992] Hölzle, U., Chambers, C., and Ungar, D. (1992). Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, San Francisco, CA. ACM Press.

[Hosking et al., 1992] Hosking, A. L., Moss, J. E. B., and Stefanović, D. (1992). A Comparative Performance Evaluation of Write Barrier Implementations. In Paepcke, A., editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia. ACM Press.

[Hudson and Moss, 1992] Hudson, R. L. and Moss, J. E. B. (1992). Incremental Garbage Collection for Mature Objects. In [Bekkers and Cohen, 1992].

[Huelsbergen and Winterbottom, 1998] Huelsbergen, L. and Winterbottom, P. (1998). Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization. In Jones, R., editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 166–175, Vancouver. ACM Press. ISMM is the successor to the IWMM series of workshops.

[IBM, 2000] IBM (2000). IBM Developer Kit for AIX, Java Technology Edition. `http://www.ibm.com/java/jdk/aix/`.

[Intel, 2000] Intel (2000). Microprocessor Research Labs – Open Runtime Platform. `http://www.intel.com/research/mrl/orp/`.

[Jessen and Valk, 1987] Jessen, E. and Valk, R. (1987). *Rechensysteme*. Springer-Verlag.

[Jones and Lins, 1996] Jones, R. E. and Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. With a chapter on Distributed Garbage Collection by R. Lins.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming*, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley, second edition.

[Kolodner and Petrank, 1999] Kolodner, E. K. and Petrank, E. (1999). Parallel Copying Garbage Collection using Delayed Allocation. Technical report, Department of Computer Science, Princeton University.

[Krall and Probst, 1998] Krall, A. and Probst, M. (1998). Monitors and Exceptions: How to Implement Java Efficiently. In *ACM 1998 Workshop on Java for High-Performance Computing*, Palo Alto, California.

[Lamport, 1979] Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C–28(9):690–691.

[Lenoski and Weber, 1995] Lenoski, D. E. and Weber, W.-D. (1995). *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann.

[Leroy, 1997] Leroy, X. (1997). The LinuxThreads Library. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[Lieberman and Hewitt, 1983] Lieberman, H. and Hewitt, C. E. (1983). A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lindholm and Yellin, 1996] Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley.

[Lockemann and Schmidt, 1987] Lockemann, P. C. and Schmidt, J. W., editors (1987). *Datenbankhandbuch*. Springer-Verlag.

[Mathiske and Schneider, 2000] Mathiske, B. and Schneider, D. (2000). Automatic Persistent Memory Management for the Spotless Java^{TM} Virtual Machine on the Palm Connected Organizer^{TM}. Technical Report TR-2000-89, Sun Microsystems Laboratories, Mountain View, CA.

[May et al., 1994] May et al., editors (1994). *The PowerPC Architecture*. Morgan Kaufmann.

[Meloan, 1999] Meloan, S. (1999). The Java HotSpot^{TM} Perfomance Engine: An In-Depth Look. Article on Sun's Java Developer Connection site.

[Miranda, 1997] Miranda, E. (1997). *VisualWorks Threaded Interconnect*. ObjectShare, Inc. `http://www.objectshare.com/thapi/`.

[Moon, 1984] Moon, D. A. (1984). Garbage Collection in a Large LISP System. In Steele, G. L., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX. ACM Press.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[NaturalBridge, 2000] NaturalBridge (2000). BulletTrain Optimizing Bytecode Compiler. `http://www.naturalbridge.com/bullettrain.html`.

[Nelson, 1991] Nelson, G., editor (1991). *Systems Programming with Modula-3*. Prentice Hall.

[Nori et al., 1974] Nori, K. V., Ammann, U., Jensen, K., and Naegeli, H. H. (1974). The Pascal (P) Compiler Implementation Notes. Technical Report 10, Eidgenoessische Technische Hochschule Zürich (ETH). Institut für Informatik.

[Nyhoff and Leestma, 1995] Nyhoff, L. R. and Leestma, S. (1995). *FORTRAN 77 and Numerical Methods for Engineers and Scientists*. Prentice-Hall.

[Onodera, 2000] Onodera, T. (2000). Personal Communication.

[Onodera and Kawachiya, 1999] Onodera, T. and Kawachiya, K. (1999). A Study of Locking Objects with Bimodal Fields. In [OOPSLA, 1999].

[OOPSLA, 1999] OOPSLA (1999). *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO. ACM Press.

[OOPSLA, 2000] OOPSLA (2000). *OOPSLA'00 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Minneapolis. ACM Press.

[Park and Goldberg, 1992] Park, Y. G. and Goldberg, B. (1992). Escape Analysis on Lists. *ACM SIGPLAN Notices*, 27(7):116–127.

[Peyton Jones, 1989] Peyton Jones, S. (1989). Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186.

[PLDI, 1998] PLDI (1998). *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal. ACM Press.

[PLDI, 2000] PLDI (2000). *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver. ACM Press.

[Poletto et al., 1998] Poletto, M. et al. (1998). 'C and tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems*.

[Poletto and Sarkar, 1999] Poletto, M. and Sarkar, V. (1999). Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913.

[POSIX, 1996] POSIX (1996). *Portable Operating Systems Interface (POSIX) — Part 1: System Application Program Interface*. IEEE.

[Pugh, 2000a] Pugh, W. (2000a). The "Double-Checked Locking is Broken" Declaration. `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[Pugh, 2000b] Pugh, W. (2000b). The Java Memory Model is Fatally Flawed. *Concurrency: Practice and Experience*, 12(1):1–11.

[Rogue Wave, 2000] Rogue Wave (2000). Threads.h++. `http://www.roguewave.com/prodcuts/xplatform/threads/`.

[Scales et al., 2000] Scales, D. et al. (2000). The Swift Java Compiler: Design and Implementation. Research Report 2000/2, Compaq Western Research Laboratory, Palo Alto, CA.

[Schorr and Waite, 1967] Schorr, H. and Waite, W. (1967). An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10(8):501–506.

[Seligmann and Grarup, 1995] Seligmann, J. and Grarup, S. (1995). Incremental Mature Garbage Collection Using the Train Algorithm. In [ECOOP, 1995].

[Shivers et al., 1999] Shivers, O., Clark, J., and McGrath, R. (1999). Atomic Heap Transactions and Fine-Grain Interrupts. In *Proceedings of 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris.*, ACM SIGPLAN Notices. ACM Press.

[Sichnoth et al., 1999] Sichnoth, J. M., Lueh, G.-Y., and Ciernak, M. (1999). Support for garbage collection at every instruction in a Java^{TM} compiler. In *Proceedings of 1999 ACM SIGPLAN International Conference on Programming Language Desing and Implementation*, ACM SIGPLAN Notices. ACM Press.

[Sites, 1992] Sites, R. L., editor (1992). *Alpha Architecture Manual*. Digital Press.

[SPARC, 1998] SPARC (1998). *SPARC Assembly Language Reference Manual*. SPARC International.

[Stoutamire and Omohundro, 1996] Stoutamire, D. and Omohundro, S. (1996). The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA.

[Sun Microsystems, 1999] Sun Microsystems (1999). Java Hotspot Performance Engine. `http://java.sun.com/products/hotspot/`.

[Sun Microsystems, 2000a] Sun Microsystems (2000a). Java 2 SDK, Standard Edition. `http://java.sun.com/products/jdk/1.2/download-solaris.html`.

[Sun Microsystems, 2000b] Sun Microsystems (2000b). Self Home Page. `http://www.sun.com/research/self/`.

[Sun Microsystems, 2000c] Sun Microsystems (2000c). *Solaris 8 Multithreaded Programming Guide*. Sun Microsystems. `http://docs.sun.com/ab2/coll.45.13/MTP/`.

[Sun Microsystems, 2000d] Sun Microsystems (2000d). Solaris Patches. `http://java.sun.com/j2se/1.3/install-solaris-patches.html`.

[Sun Microsystems, 2000e] Sun Microsystems (2000e). Sun Labs Java Technology Research Group. `http://www.sun.com/research/jtech/`.

[Sunderam, 1989] Sunderam, V. (1989). PVM — a Framework for Parallel Distributed Computing. Technical Report ORNL-TM-11375, Oak Ridge National Laboratory.

[Tanenbaum, 1992] Tanenbaum, A. S. (1992). *Modern Operating Systems*. Prentice-Hall.

[TowerJ, 2000] TowerJ (2000). A High Performance Deployment Solution for Java Server Applications. `http://www.towerj.com/`.

[Ungar, 1984] Ungar, D. M. (1984). Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[Ungar and Jackson, 1991] Ungar, D. M. and Jackson, F. (1991). Outwitting GC Devils: A Hybrid Incremental Garbage Collector. In Wilson, P. R. and Hayes, B., editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*.

[Vetter et al., 1999] Vetter, S. et al. (1999). *AIX Version 4.3 Differences Guide*. IBM Corp. `http://www.redbooks.ibm.com/abstracts/sg242014.html`.

[Volano, 2000] Volano (2000). The Volano Report. `http://www.volano.com/report.html`.

[Weaver and Germond, 1994] Weaver, D. L. and Germond, T., editors (1994). *The SPARC Architecture Manual*. SPARC International.

[Weikard, 1998] Weikard, M. (1998). Entwurf und Implementierung einer Portablen Multiprozessorfähigen Virtuellen Maschine für eine Persistente, Objektorientierte Programmiersprache. Master's thesis, Universität Hamburg, Fachbereich Informatik.

[Whaley and Rinard, 1999] Whaley, J. and Rinard, M. (1999). Compositional Pointer and Escape Analysis for Java Programs. In [OOPSLA, 1999], pages 187–206.

[White and Garthwaite, 1998] White, D. and Garthwaite, A. (1998). The GC Interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, Mountain View, CA.

[Wilson, 1992] Wilson, P. R. (1992). Uniprocessor Garbage Collection Techniques. In [Bekkers and Cohen, 1992].

Erklärung

Hiermit erklären wir, daß wir die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Hamburg, den 19. Dezember 2000

(Matthias Ernst) (Daniel Schneider)