Eberhard Karls
Universität
Tübingen

Fakultät für Informations- und Kognitionswissenschaften

Wilhelm-Schickhard-Institut für Informatik

Lehrstuhl Programmiersprachen und Übersetzer

Multiprozessorunterstützung für Scheme-48

Diplomarbeit

David Frese

28. Februar 2006

Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbständig verfasst zu haben. Es wurden keine anderen als die angegebenen Quellen benutzt.

Prüfer
Prof. Dr. Herbert Klaeren

Betreuer
Dr. Martin Gasbichler

Zusammenfassung

Die vorliegende Arbeit zeigt, daß es möglich ist, das Scheme-48-System so zu erweitern und zu modifizieren, daß es die Rechenleistung einer symmetrischen Multiprozessormaschine für die Interpretation einer nebenläufigen Scheme-Anwendung effizient auszunutzen kann, und daß dabei auf größere Veränderungen des Quelltextes des Systems verzichtet werden kann.

Dazu wird die virtuelle Maschine des Scheme-48-Systems um *virtuelle Prozessoren* erweitert. Diese virtuellen Prozessoren führen jeweils eigene Scheme-Funktionen aus, greifen aber auf einem gemeinsamen Speicher zu. Sie sind so realisiert, daß sie unter einem multiprozessorfähigen Betriebsystem auf verschiedenen realen Prozessoren ausgeführt werden können. Mithilfe von zwei Erweiterungen des Pre-Scheme-Compilers kann dabei eine größere Veränderung des Quelltextes umgangen werden. Modifikationen an der Speicherverwaltung stellen dabei sicher, daß die virtuellen Prozessoren effizient Speicher allokieren können, und eine automatische Speicherbereinigung durchgeführt werden kann. Durch eine Anpassung des I/O-Systems von Scheme-48 können alle virtuellen Prozessoren auf jede geöffnete Datei zugreifen. Weiter wird das Thread-System der Scheme-48-Laufzeitumgebung um die Möglichkeit erweitert, die Ausführung der Threads von mehreren virtuellen Prozessoren gemeinsam vornehmen zu lassen. Dadurch kann eine bestehende nebenläufige Scheme-Anwendung ohne Änderungen von der Rechenleistung einer Multiprozessormaschine profitieren.

Schließlich zeigen Messungen auf verschiedenen Multiprozessorsystemen die Funktionsfähigkeit und Effizienz der vorgestellten Implementierung. Der Overhead von circa 12% gegenüber dem unveränderten Scheme-48-System, sowie Speedup-Faktoren von durchschnittlich etwa 1,7 bei Verdoppelung der Prozessoranzahl, sind sehr gute Werte für eine Multiprozessoranwendung.

Inhaltsverzeichnis

1 Einführung

Rechner mit mehreren Prozessoren finden in den letzten Jahren immer größere Verbreitung. Ein Grund dafür ist, daß eine Steigerung des Prozessortakts aufgrund der überproportional steigenden Wärmeentwicklung zunehmend schwieriger und teurer wird. Ein weiterer Grund ist die bereits sehr verbreitete nebenläufige Programmierung. Insbesondere viele Server-Anwendungen sind damit in der Lage, viele Client-Anfragen gleichzeitig zu bearbeiten. Die Verwendung mehrerer Prozessoren in einem Rechner ist daher ein relativ einfacher Weg den Durchsatz der Server zu vergößern.

Ziel dieser Arbeit ist es daher, dem Scheme-48-Programmierer die Rechenleistung einer Multiprozessormaschine auf möglichst einfache Art und Weise zugänglich zu machen. Das Scheme-48-System enthält bereits sogenannte User-Level-Threads, die dem Benutzer die nebenläufigen Programmierung ermöglichen. Diese Arbeit zeigt, wie das Scheme-48-System erweitert und verändert werden kann, um mit den selben User-Level-Threads die Rechenleistung einer Multiprozessormaschine ausnutzen zu können. Das Scheme-48-System kann dadurch ein nebenläufiges Scheme-Programm ohne Modifikation deutlich schneller interpretieren.

1.1 Programmieren von SMP-Maschinen

Symmetrische Multiprozessormaschinen zeichnen sich dadurch aus, daß alle Prozessoren *gleichartig* sind, das heißt sie erfüllen keine spezialisierten Aufgaben. Außerdem greifen alle Prozessoren *gleichberechtigt* auf einen *gemeinsamen Hauptspeicher* zu. Eine einfache Möglichkeit für Betriebsysteme solche SMP-Maschinen zu unterstützen ist, das sogenannte *Multitasking* auf alle Prozessoren auszuweiten. Dieser Abschnitt erläutert zwei verschiedene Methoden, die Betriebsysteme dem Anwendungs-Programmierer anbieten, um die Rechenleistung einer Multiprozessormaschine für eine Anwendung ausnutzen zu können.

Das Multitasking gehört, spätestens seit der Entwicklung von UNIX in den 60er und 70er Jahren, zu den zentralen Aufgaben eines Betriebsystems. Multitasking bedeutet, daß der Benutzer eine große Zahl von Programmen starten kann, und das Betriebsystem jedes Programm abwechselnd für kurze Zeit vom Prozessor ausführen lässt. Auf einem Prozessor kann dadurch eine Vielzahl von Programmen *scheinbar* gleichzeitig ablaufen. Alle dafür notwendigen Informationen über ein gestartetes Programm nennt man einen *Prozess*. Dazu gehören beispielsweise die Registerwerte, der Stapel, die vom Programm belegten Speicherbereiche und geöffneten Dateien. Das Betriebsystem verwaltet also beim Multitasking eine Liste mit allen Prozessen, und teilt sie abwechselnd für kurze Zeit dem Prozessor zu.

Verfügt eine Maschine über mehrere gleichartige Prozessoren, so kann das Betriebsystem jedem Prozessor je einen Prozess zuteilen. Von den vielen gestarteten Programmen können also zu einem bestimmten Zeitpunkt mehrere *echt* gleichzeitig ausgeführt werden. Nach welchen Kriterien diese Zuteilung genau erfolgt, ist dabei von Betriebsystem zu Betriebsystem sehr unterschiedlich [19].

Um weitere Programme zu starten, stellen UNIX-Betriebsysteme einen Betriebsystemaufruf namens `fork` zur Verfügung. Dieser Aufruf erzeugt ein vollständiges Duplikat des jeweiligen Prozesses, und fügt das Duplikat in die Prozessliste ein. Insbesondere wird dabei auch der Programmzähler übernommen,

sodaß auch im neuen Prozess hinter dem Betriebsystemaufruf fortgefahren wird. In diesem sogenannten Kind-Prozess Prozess hat der Aufruf jedoch einen anderen Rückgabewert als im Vater-Prozess, worüber der Programmierer für den neuen Prozess eine spezielle Aufgabe vorsehen kann. Das Betriebsystem kann diesen Prozessen dann verschiedene Prozessoren zuteilen, und so eine gleichzeitige Ausführung der Prozesse bewirken. Um also mit einem einzelnen Programm die Leistungsfähigkeit eines Rechners mit mehreren Prozessoren ausnutzen zu können, kann der Programmierer die Arbeit des Programms auf mehrere Prozesse verteilen. Man spricht dabei von einer *Parallelisierung* des Programms. Da aber jeder Prozess einen eigenen virtuellen Adressraum besitzt, benötigt der Programmierer in der Regel noch eine Möglichkeit zwischen den Prozessen Daten auszutauschen. Die einfachste Art dieser sogenannten *Inter-Prozess-Kommunikation* ist das Schreiben und Lesen von Dateien, da das Dateisystem immer eine *gemeinsame Ressource* ist. Moderne Betriebsysteme bieten aber abstraktere Möglichkeiten für die Kommunikation zwischen Prozessen, wie zum Beispiel die gemeinsame Nutzung bestimmter Speicherbereiche, oder der Austausch von Nachrichten [15]. Grundlegendes Muster dieses *Prozess-Modells* zur Parallelisierung ist jedoch, daß die Prozesse zunächst vollständig getrennt sind, und gemeinsame Daten *explizit* ausgetauscht oder definiert werden müssen.

Neuere Betriebsysteme bieten eine weitere Möglichkeit zur Parallelisierung eines Programms: die sogenannten *Kernel-Level-Threads*[1] [20]. Ein *Thread of Execution*, kurz *Thread*, stellt den Kontrollfluß eines Programms dar. Er besteht also insbesondere aus den Registerwerten und einem Ausführungsstapel. Jeder Prozess besteht immer mindestens aus einem Thread, aber Betriebsysteme die Kernel-Level-Threads anbieten, geben dem Programm die Möglichkeit weitere Threads hinzuzufügen. Anstelle des ganzen Prozesses dupliziert das Betriebsystem dazu allein die thread-spezifischen Daten, um so einen weiteren, unabhängigen Kontrollfluß im selben Prozess zu erzeugen. Alle anderen prozessspezifischen Daten, insbesondere der virtuelle Adressraum, sind für alle Threads des Prozesses gleich. Die Kommunikation kann in diesem *Thread-Modell* daher direkt über den gemeinsamen Speicher erfolgen. Das Betriebsystem kann nun auf SMP-Maschinen verschiedene Threads eines Prozesses jeweils einem anderen Prozessor zuteilen. Kernel-Level-Threads können also echt parallel ausgeführt werden. Abbildung 1 veranschaulicht diese Situation.

Um nun also mit einem Programm mehrere Prozessoren ausnutzen zu können, muß die Arbeit des Programms entweder auf mehrere Prozesse, oder auf mehrere Kernel-Level-Threads verteilt werden. Das Thread-Modell hat Vorteile, wenn zwischen den Threads viel kommuniziert wird, oder die Threads sehr kurzlebig sind. Ein Vorteil des Prozess-Modells ist die bessere Skalierbarkeit auf Rechner-Netzwerke. Je nach Situation kann daher das eine oder das andere Modell geeigneter sein.

1.2 Nebenläufige Programmierung

Unter nebenläufiger Programmierung versteht man im allgemeinen die Nutzung mehrerer Threads, also mehrerer sequentieller Ausführungsstränge, in *einem* Programm. Die Ausnutzung mehrere Prozessoren ist dabei nicht der einzige

[1] Diese Bezeichnung dient der Abgrenzung gegenüber User-Level-Threads, die eine Verteilung der Rechenzeit eines einzelnen Prozesses durch Multitasking bezeichnen.

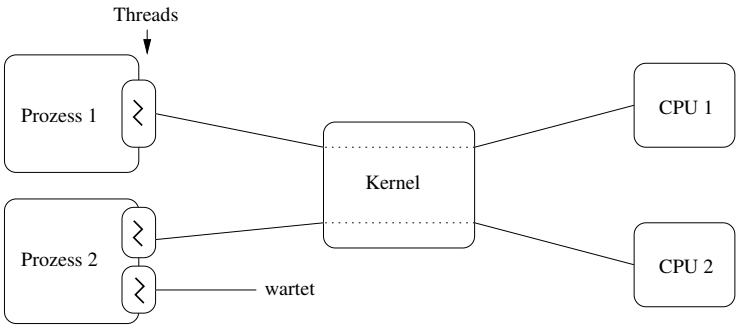

Abbildung 1: Kernel-Level-Threads

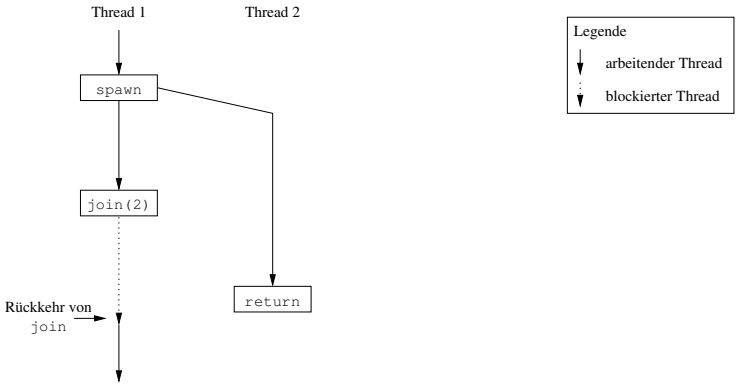

Abbildung 2: Erzeugung und Beendigung eines Threads

Anwendungsbereich der nebenläufigen Programmierung. Als Programmiermodell können damit insbesondere *interaktive Systeme* wesentlich einfacher und eleganter implementiert werden [17].

Zu jeder Programmierumgebung für nebenläufige Programmierung gehört eine Operation, die einen neuen Thread erzeugt. Diese Operation wird in der Regel `spawn` genannt. Als Parameter gehört zu dieser Operation dabei zumeist der *Eintrittspunkt* des neuen Threads, also der Anfang des Programmcodes, den der neue Thread ausführen soll. Dieser wird in höheren Programmiersprachen in der Regel in Form einer *Funktion* angegeben, nach deren Rückkehr der Thread seine Aufgabe erfüllt hat und sich beendet. Als *Join* bezeichnet man das Warten auf diesen Rückgabewert durch einen anderen Thread. Einen wartenden Thread bezeichnet man als *blockiert*, wenn die Programmierumgebung so gestaltet ist, daß sie für wartende Threads keine Rechenzeit aufwenden muß. Abbildung 2 zeigt zur Veranschaulichung eine schematische Darstellung des Kontrollflusses in zwei Threads. Der erste Thread erzeugt dabei einen zweiten Thread und blockiert, bis dieser zurückkehrt.

Neben der Erzeugung von Threads, sind *Synchronisierungmittel* ein wichtiger Bestandteil der nebenläufigen Programmierung. Mit Synchronisierung bezeichnet man die Koordinierung des Zugriffs auf gemeinsame Daten oder be-

schränkte Resourcen durch mehrere Threads. Es gibt diverse Konzepte zur Synchronisierung von Threads, darunter beispielweise die *Semaphoren* von Dijkstra [2] oder die *Monitore* von Hoare [8]. Zwei einfachere Konzepte, die mehrmals in dieser Arbeit verwendet wurden, sollen hier kurz erläutert werden: *Mutex-Locks* und *Condition-Variablen.*

Ein Mutex-Lock kann zu einem bestimmten Zeitpunkt von maximal einem Thread *besessen* werden. Ein Thread kann ein Mutex-Lock entsprechend *aquirieren* und *freigeben.* Versucht ein Thread ein von einem anderen Thread besessenes Mutex-Lock zu aquirieren, dann *blockiert* er, bis der andere Thread das Mutex-Lock freigibt. Die häufigste Verwendung von Mutex-Locks ist, daß die Threads eines Programms auf eine gemeinsame Datenstruktur nur dann zugreifen, wenn sie ein bestimmtes Mutex-Lock besitzen. Die Erhöhung eines gemeinsamen Zählers könnte in einem Pseude-Code beispielsweise folgendermaßen synchronisiert werden:

```
aquire(lock)
counter <- counter + 1
release(lock)
```

Die Synchronisierung ist notwendig, weil die Erhöhung des Zählers aus drei Schritten besteht: Lesen, Addition und Zurückschreiben. Versuchen zwei Threads den Zähler nahezu gleichzeitig zu erhöhen, dann könnte es ohne Synchronisierung beispielsweise passieren, daß zwei Threads denselben momentanen Wert auslesen. Beide würden dann im Anschluß denselben erhöhten Wert in den Speicher zurückschreiben, was bedeutet, daß der Zähler insgesamt nur um eins, und nicht um zwei erhöht wird.

Condition-Variablen ermöglichen zusammen mit einem Mutex-Lock eine bessere Synchronisierung des Zugriffs auf Datenstrukturen mit *Zustand.* Tauschen mehrere Threads beispielsweise Daten über einen einelementigen Puffer aus, dann kann der Puffer entweder den Zustand „leer" oder „voll" annehmen. Der Zugriff auf den Puffer muß dabei, ebenso wie beim Zähler über ein Mutex-Lock synchronisiert werden. Aquiriert ein Thread das Mutex-Lock, um den Puffer zu füllen, und stellt dann fest, daß der Puffer nicht leer ist, dann muß er das Mutex-Lock wieder freigeben, damit ein anderer Thread den Puffer leeren kann. Ohne Condition-Variablen müsste ein schreibender Zugriff auf den Puffer daher folgendermaßen aussehen:

```
aquire(lock)
while full?(buffer)
  release(lock)
  aquire(lock)
buffer <- data
release(lock)
```

Der Thread müsste also das Mutex-Lock immer wieder freigeben und aquirieren, bis ein anderer Thread den aktuellen Wert aus dem Puffer entfernt hat. Eine solche Situation bezeichnet man als „*busy waiting*", oder *aktives Warten.* Solche Situationen sind sehr ineffizient, da dieser Thread mit dem Durchlaufen der Schleife Rechenzeit verbraucht, die andere Threads vielleicht besser verwenden könnten. Besonders kritisch ist dies, wenn dieser Thread unterbrochen wird, während er das Mutex-Lock besitzt, denn dadurch ist es einem anderen Thread

gar nicht möglich, die gewünschte Änderung des Zustands zu bewirken. Dieses Phänomen bezeichnet man als *Prioritäten-Inversion*, da der Thread, dessen Ausführung in dieser Situation wichtiger wäre, keinen Fortschritt machen kann.

Condition-Variablen dienen dazu diese Situation zu verhindern, indem sie den Threads ermöglichen explizit auf das Eintreten eines Zustands *passiv* zu *warten*, beziehungsweise eine Änderung des Zustands zu *signalisieren*. In diesem Beispiel definiert der Programmierer dazu für jeden der beiden Zustände „leer" und „voll" eine Condition-Variable. Der schreibende Zugriff auf den Puffer kann dann beispielsweise folgendermaßen aussehen:

```
aquire(lock)
if full?(buffer)
  wait(empty-condition, lock)
buffer <- data
signal(full-condition)
release(lock)
```

Der lesende Zugriff entsprechend:

```
aquire(lock)
if empty?(buffer)
  wait(full-condition, lock)
data <- buffer
signal(empty-condition)
release(lock)
```

`Wait` gibt jeweils das Mutex-Lock frei, wartet auf die Signalisierung der entsprechenden Bedingung durch `signal`, und aquiriert das Mutex-Lock dann wieder. Abbildung 3 zeigt schematisch den Kontrollfluß zweier Threads, die eine Schreib- beziehungsweise eine Lese-Operation auf einem zunächst leeren Puffer durchführen.

Sind mehr als zwei Threads beteiligt, dann ist noch wichtig, ob `signal` nur einen einzigen wartenden Thread *aufweckt*, das heißt ob *ein* Aufruf von `signal` zur Rückkehr von *einem* `wait`-Aufruf führt, oder ob alle wartenden Threads aufwachen. Dies hängt von der Implementierung des Thread-Systems ab, und in Systemen die beide Alternativen anbieten, heißt erstere Operation meist `signal`, letztere `broadcast`.

1.3 Scheme-48

Scheme-48 ist eine Implementierung für die Programmiersprache Scheme [12, 3], deren Entwicklung 1993 von Richard Kelsey und Jonathan Rees begonnen wurde. Scheme ist eine universelle Programmiersprache mit lexikalischer Bindung, dynamischer Typisierung, higher-order Funktionen und vollständiger Endrekursion. Die Syntax ist angelehnt an die Programmiersprache Lisp, weshalb Scheme auch als Lisp-Dialekt bezeichnet wird. Grundlage für diese Arbeit ist eine Entwicklungsversion von Scheme-48 des Wilhelm-Schickhard-Instituts für Informatik an der Universität Tübingen[2], die in etwa der offiziellen Version 1.3 entspricht.

[2] Revision 1428 im Subversion-Repository des Instituts.

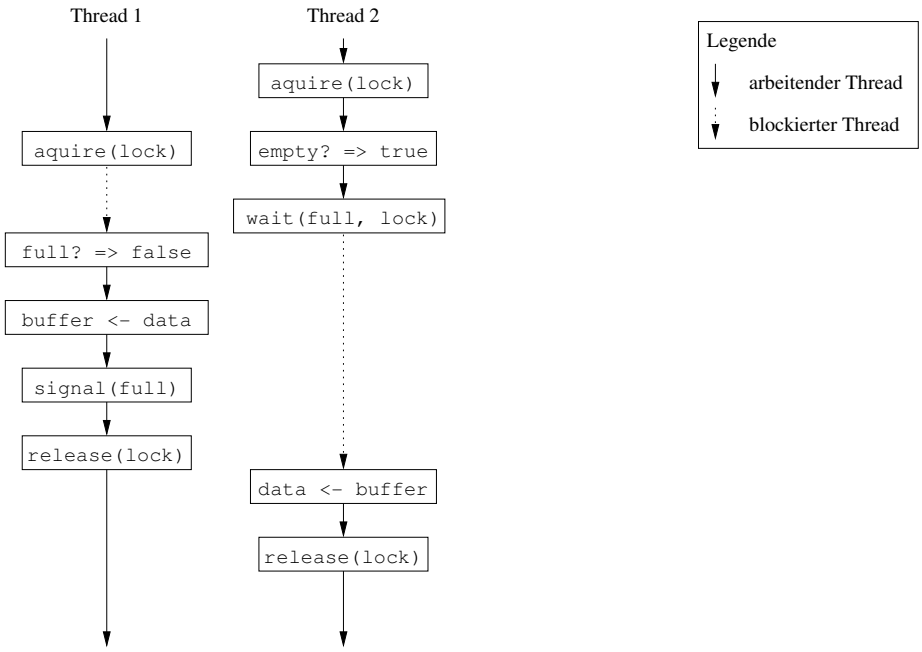

Abbildung 3: Synchronisierung mit Condition-Variablen

Das Scheme-48-System basiert auf einer virtuellen Byte-Code-Maschine, einem integrierten Compiler für Scheme-Code und einem großen Laufzeitsystem. Für diese Arbeit von besonderem Interesse ist, daß Scheme-48 bereits die nebenläufige Programmierung über *Threads* ermöglicht. Ein Fragment einer Server-Anwendung könnte beispielsweise folgendermaßen aussehen:

```
(define (server)
  (let ((client (wait-for-client)))
    (spawn (lambda () (serve client)))
    (server)))

(define (serve client)
  (let* ((request (read-request client))
         (result (handle-request request)))
    (send-response result client)))
```

Die Funktion `server` stellt den Server-Loop dar, der fortwährend auf neue Klientenanfragen wartet. Zur Bedienung eines Klienten durch die Funktion `serve` wird jedoch über die Funktion `spawn` ein neuer Thread gestartet. Das heißt, die Analyse der Anfrage, die Zusammenstellung der Antwort, sowie das Senden der Antwort findet nebenläufig zum restlichen Programm statt. Bei vielen Anfragen in kurzer Zeit kann der Server dadurch mehrere Klienten gleichzeitig bedienen, und so seinen Durchsatz steigern.

Scheme-48 unterstützt bislang jedoch nur sogenanntes Multitasking, mit dem eine *scheinbar* gleichzeitige Ausführung der Threads realisiert wird. Diese Arbeit ermöglicht hingegen die *echt* gleichzeitige Ausführung der Threads auf Multiprozessormaschinen. Dazu wird die virtuelle Maschine in eine virtuelle, symme-

trische Multiprozessormaschine verwandelt, das heißt sie bietet dem Laufzeitsystem mehrere virtuelle Prozessoren an, die als Kernel-Level-Threads implementiert sind, und daher vom Betriebsystem auf verschiedene reale Prozessoren verteilt werden können. Das Laufzeitsystem nutzt diese virtuellen Prozessoren dann dazu, um über parallele Scheduler mehrere Threads *echt* gleichzeitig auszuführen.

Zum experimentellen Test eines Subsystems für optimistische Nebenläufigkeit, wurde eine ältere Version von Scheme-48 schon einmal um eine Multiprozessorunterstützung erweitert. Diese ist jedoch strukturell an ein besonderes Feature des Linux-Betriebsystems gebunden, und daher nicht auf andere Plattformen portierbar. Außerdem wurden darin viele Probleme der Multiprozessorfähigkeit nicht behandelt, wodurch die Implementierung sehr instabil ist. Diese Test-Version konnte daher nicht als Grundlage für diese Arbeit verwendet werden, jedoch diente sie als Inspiration für die Ideen der virtuellen Prozessoren und der parallelen Scheduler.

1.4 Gliederung

Diese Arbeit ist folgendermaßen gegliedert. Die Abschnitte 3 und 4 präsentieren die vorgenommenen Änderungen an der Implementierung von Scheme-48, sowie teilweise Diskussionen von alternativen Möglichkeiten.

Abschnitt 4 stellt experimentelle Ergebnisse des veränderten Interpreters auf mehreren Multiprozessormaschinen, sowie im Vergleich zum unveränderten Interpreter dar.

Schließlich gibt Abschnitt 5 eine Zusammenfassung der Arbeit, und Hinweise auf mögliche Erweiterungen und weitere Verbesserungen des multiprozessorfähigen Scheme-48-Interpreters.

2 Die virtuelle Maschine

Die Implementierung von Scheme-48 basiert auf einem integrierten Compiler und einer virtuellen Maschine. Der integrierte Compiler übersetzt Scheme-Code in Byte-Code, der von der virtuelle Maschine interpretiert wird. Die virtuelle Maschine bietet neben einem Stapel für Argumente und Funktionsaufrufe auch einen Haldenspeicher und einige grundlegende Datentypen. Der Compiler ist für diese Arbeit unbedeutend, weshalb hier nicht weiter auf ihn eingeganden wird. Die Veränderungen an der virtuellen Maschine sind jedoch der wichtigste Teil der Multiprozessorerweiterung. Der folgenden Abschnitt gibt daher zunächst einen Überblick über den Aufbau der virtuellen Maschine, sowie das Ziel der Veränderungen daran. Die Abschnitte 2.2 bis 2.4 beschreiben dann Details der wichtigsten Änderungen und Konzepte, namentlich die *virtuellen Prozessoren*, die Speicherverwaltung sowie das I/O-System.

2.1 Aufbau und Ziel

Der Aufbau der virtuellen Maschine lässt sich zunächst grob in die Verwaltung des Haldenspeichers, und den eigentlichen Interpreter des Byte-Codes, im folgenden *virtueller Prozessor* genannt, unterteilen. Die Speicherverwaltung beinhaltet mehrere Allokationsfunktionen für Speicherobjekte, eine automatische Bereinigung des Speichers von nicht mehr verwendeten Daten (eine sogenannte Garbage-Collection), Suchfunktionen für Datenobjekte eines bestimmten Typs, sowie den sogenannten *Dumper*. Der Dumper kann ein Abbild des Speichers in eine Datei schreiben, später ein solches *Image* wieder laden und die Ausführung einer bestimmten Funktion, beziehungsweise eines bestimmten Byte-Code-Programms aus diesem Image auf dem virtuellen Prozessor bewirken.

Das Ziel ist nun, die virtuellen Maschine in eine „virtuelle symmetrische Multiprozessormaschine" zu verwandeln. Das bedeutet erstens, daß mehrere virtuelle Prozessore parallel verschiedene Byte-Code-Programme ausführen, und zweitens alle virtuellen Prozessoren auf einen gemeinsamen Speicher zugreifen. Abbildung 4 zeigt eine schematische Darstellung dieser virtuellen Multiprozessormaschine. Die Abbildung zeigt zwei virtuelle Prozessoren, die jeweils eigene Register und einen Ausführungsstapel haben, von denen aus Zeiger in den Speicher auf das jeweilige Byte-Code-Programm und weitere Daten zeigen. Hervorgehoben ist ein Datenobjekt, das beiden Programmen gemeinsam ist.

Die virtuelle Maschine verwaltet außerdem die vom Scheme-Programm benötigten *Betriebsmittel*, wie zum Beispiel offene Dateien. Da die virtuellen Prozessoren weitgehend transparent für den Programmierer sein sollen, ist es notwendig, daß die auf einem virtuellen Prozessor geöffneten Dateien, auch auf anderen virtuellen Prozessoren verwendet werden können. Dies sicherzustellen ist daher eine weitere Zielsetzung für die virtuelle Multiprozessormaschine.

Die folgenden Abschnitte beschreiben nun jeweils die Realisierung eines dieser Ziele:

1. Abschnitt 2.2 zeigt die Realisierung mehrerer *virtuellen Prozessoren*, die jeweils eigene Programme ausführen, sowie parallel auf den Prozessoren einer (realen) Multiprozessormaschine ausgeführt werden können.

2. Abschnitt 2.3 behandelt den Zugriff auf einen gemeinsamen Speicher, die Synchronisation der Allokation im gemeinsamer Speicher, sowie dessen

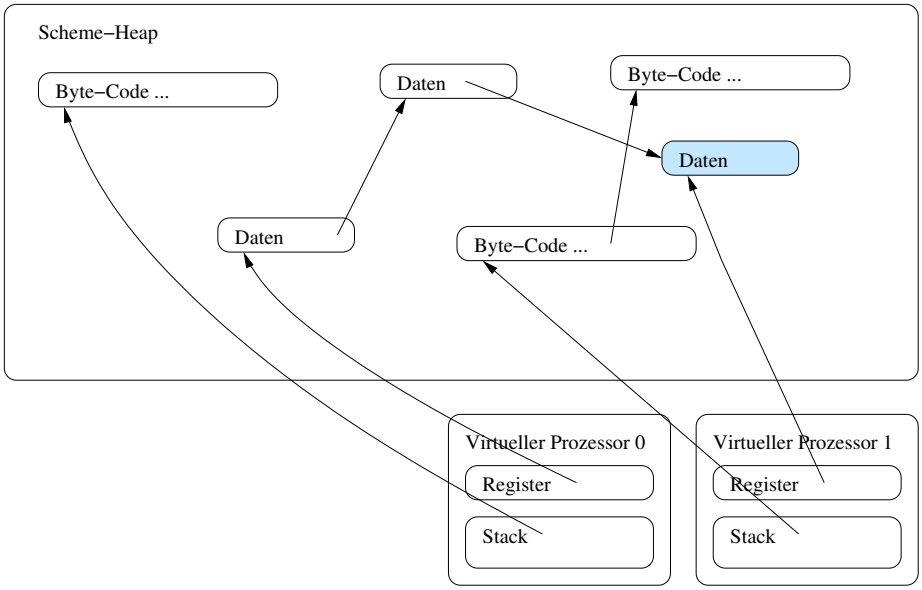

Abbildung 4: Die virtuelle Maschine

automatischer Bereinigung.

3. Abschnitt 2.4 zeigt wie alle virtuellen Prozessoren auf gemeinsame Betriebsmittel zugreifen können.

2.1.1 Pre-Scheme

Für das Verständnis der folgenden Abschnitte ist es wichtig zu wissen, daß die virtuelle Maschine zum Teil in C, und zum Teil in *Pre-Scheme* [11] programmiert ist. Pre-Scheme ist ein Scheme-Dialekt, der die Sprachelemente von Scheme gerade soweit einschränkt, beziehungsweise verändert, daß Pre-Scheme-Programme in effizienten C-Code übersetzt werden können. Außerdem enthält Pre-Scheme eine automatische Typ-Inferenz und Typ-Prüfung zur Übersetzungszeit. Pre-Scheme wurde von Richard Kelsey eigens für die virtuelle Maschine von Scheme-48 entwickelt, und ein Pre-Scheme-Übersetzer ist in der Scheme-48-Distribution enthalten.

2.2 Virtuelle Prozessoren

Dieser Abschnitt erläutert, wie die virtuelle Maschine die Rechenleistung einer Multiprozessormaschine mithilfe des Betriebsystems ausnutzen kann, und wie diese Rechenleistung dem Laufzeitsystem über die virtuellen Prozessoren zur Verfügung gestellt wird.

Das Ziel der Multiprozessorerweiterung von Scheme-48 ist es, die Rechenleistung mehrerer *realer* Prozessoren in einem Computer auszunutzen. Wie in Abschnitt 1.1 beschrieben, gibt es dazu zwei verschiedene Ansätze: das Prozeß-Modell und das Thread-Modell. Beim Prozeß-Modell erzeugt das Programm zur Laufzeit weitere Kopien von sich selbst, sogenannte Kind-Prozesse, die vom Betriebsystem auf verschiedenen Prozessoren parallel ausgeführt werden können.

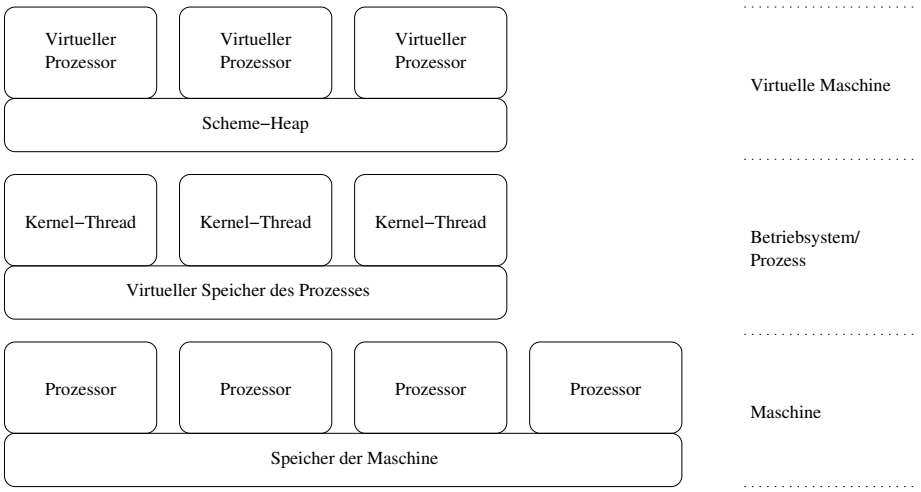

Abbildung 5: Symmetrisches Multiprocessing auf den drei Ebenen der realen Maschine, des Prozesses und der virtuellen Maschine.

Die parallelisierten Teile der Anwendung sind also zunächst vollständig getrennt. Informationen über gemeinsame Speicherbereiche oder Betriebsmittel müssen explizit über sogenannte *Inter-Prozess-Kommunikation* ausgetauscht werden. Beim Thread-Modell erzeugt das Programm zur Laufzeit weitere *Kernel-Level-Threads*, also separate Kontrollflüsse im selben Prozess. Haldenspeicher und Betriebsmittel nutzen alle Threads eines Prozesses gemeinsam. Eine Trennung bestimmter Speicherbereiche muß explizit programmiert werden. Das Betriebsystem muß dazu eine API für Threads enthalten, und insbesondere die Threads eines Prozesses auf verschiedenen Prozessoren ausführen können.

Der Autor hat für diese Arbeit das Thread-Modell gewählt, da einerseits die Voraussetzungen des gemeinsamen Speichers und der gemeinsamen Betriebsmittel bereits besser zum gesetzten Ziel der *virtuellen* symmetrischen Multiprozessormaschine passen, und andererseits insbesondere die Realisierung gemeinsamer Betriebsmittel im Prozeß-Modell als sehr schwer erschien[3].

Die Arbeit der virtuelle Maschine muß also in mehrere, möglichst unabhängige Kernel-Level-Threads aufgeteilt werden. Da die virtuellen Prozessoren allein der Ausnutzung der realen Prozessoren dienen sollen, ist es in der Regel unnötig auf einer konkreten Maschine mehr virtuelle Prozessoren zu verwenden, als reale Prozessoren vorhanden sind. Dementsprechend ist es sinnvoll, für jeden virtuellen Prozessor je einen Kernel-Level-Thread zu verwenden. Virtueller Prozessor und Kernel-Level-Thread stellen so eine feste Einheit dar, sodaß die Begriffe im folgenden teilweise synonym verwendet werden. Damit entspricht sowohl die Hardware-Ebene, die Betriebsystem-Ebene, sowie die Scheme-Ebene dem Modell der symmetrischen Multiprozessormaschine (siehe Abbildung 5).

Wie bereits erwähnt, muß bei der Verwendung von Threads die Separierung von Daten explizit programmiert werden. Die einzigen Daten, die für jeden virtuellen Prozessor separat vorhanden sein müssen, sind seine Register und sein Ausführungsstapel. Bevor aber ab Abschnitt 2.2.2 auf die damit zusammenhän-

[3]Nur Linux bietet hierbei eine Unterstützung durch den Betriebsystemaufruf `clone()` an. Auf anderen Systemen wäre die Realisierung nocheinmal schwieriger.

genden Veränderungen der virtuellen Maschine eingegangen wird, soll der folgenden Abschnitt zunächst einige Details über die Schnittstelle zur Betriebsystembibliothek für Kernel-Level-Threads und deren Verwendung in Pre-Scheme erläutern.

2.2.1 POSIX-Threads

Das *Portable Operation System Interface for Unix* [15], kurz POSIX, enthält die Beschreibung einer Schnittstelle für Kernel-Level-Threads. Um die Portierbarkeit von Scheme-48 auf verschiedene Unix-Systeme zu erhalten, beziehungsweise zu erleichtern, hat der Autor sich entschieden diese Standard-Schnittstelle zu verwenden. Die Funktionen des POSIX-Standards zur Erzeugung und Synchronisation von Kernel-Level-Threads mussten dazu zunächst für Pre-Scheme-Programme nutzbar gemacht werden. Pre-Scheme enthält dazu zwar die Möglichkeit externe Funktionen zu deklarieren, aber das Design der POSIX-Schnittstelle basiert zum großen Teil auf der Deklaration von Variablen, die als Referenz an die Funktionen der Bibliothek übergeben werden müssen. Pre-Scheme enthält aber nicht die Möglichkeit Variablen zu deklarieren, sondern basiert auf dem Variablen-Bindungs-Konzept von Scheme. Daher wurden zunächst die relevanten POSIX-Funktionen gekapselt. Als Erläuterung sei dies am Beispiel der Thread-Erzeugungsfunktion `pthread_create` demonstriert. Der Prototyp von `pthread_create` ist:

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                   void * (*start_routine)(void *), void * arg);
```

Die Funktion veranlasst die Erzeugung eines neuen Threads durch das Betriebsystem und gibt im Erfolgsfall 0, sonst einen Fehlercode zurück. Der neue Thread ruft die als drittem Parameter übergebene Funktion mit dem Argument `arg` auf. Das Ergebnis des Threads ist der Rückgabewert dieser Funktion. Als zweitem Parameter von `pthread_create` kann die Adresse einer Struktur übergeben werden, die einige Eigenschaften des neu erzeugten Threads festlegt. Wird ein Null-Zeiger übergeben, dann verwendet das Betriebsystem Standard-Eigenschaften. Eine Standard-Eigenschaft ist zum Beispiel, daß ein anderer Thread ein sogenanntes *Join* durchführen kann, das heißt der andere Thread kann auf das Ergebnis des Threads warten. Als erstem Paramter muß der Programmierer eine Referenz auf eine Variable vom Typ `pthread_t` übergeben, in die `pthread_create` eine Repräsentation des Threads schreibt.

Die Verwendung in einem C-Programm kann zum Beispiel folgendermaßen aussehen:

```
void* do_something(void* arg) {
  ...
}
void main() {
  pthread_t th;
  pthread_create(&th, NULL, &do_something, NULL);
  ...
}
```

Das Programm erzeugt einen neuen Thread mit Standard-Eigenschaften, der die Funktion `do_something` ausführt.

Zur Verwendung in Pre-Scheme wurde `pthread_create` nun in der Funktion `s48_create_joinable_thread` gekapselt (zur besseren Übersichtlichkeit fehlen einige Typecasts und Fehlertests):

```
char* s48_create_joinable_thread(char*(*routine)(char*),
                                 char* arg) {
  pthread_t* th = (pthread_t*)malloc(sizeof(pthread_t));
  pthread_create(th, NULL, routine, arg);
  return (char*)th;
}
```

Die Funktion nimmt die vom neuen Thread auszuführende Funktion und deren Argument als Parameter. Zunächst allokiert sie Speicher für eine Variable vom Typ `pthread_t`, und erzeugt dann einen Thread mit Standard-Eigenschaften durch einen entsprechenden Aufruf von `pthread_create`. Die Adresse der `pthread_t`-Variablen wird schließlich zurückgegeben. Weil das Typsystem von Pre-Scheme nicht um externe Typen erweiterbar ist, muß die Adresse der Variablen als Zeiger auf ein Zeichen zurückgegeben werden. Aus dem selben Grund ist der Typ der vom Thread auszuführenden Funktion und ihrem Argument verändert.

Diese Funktion kann dann folgendermaßen in Pre-Scheme definiert und verwendet werden:

```
(define s48-create-joinable-thread
  (external "s48_create_joinable_thread"
            (=> ((=> (address) address) address) address)))

(define (do-something arg)
  ...)

(let ((th (create-joinable-thread do-something
                                  null-address)))
  ...)
```

Der erste Ausdruck definiert den Namen `s48-create-joinable-thread` als externen Wert in Pre-Scheme. Dazu muß erstens der externe Name angegeben werden, den der Pre-Scheme-Compiler in den erzeugten C-Code einsetzt, und zweitens, für die Typprüfung des Pre-Scheme-Compilers, der Typ des Wertes angegeben werden. Die Syntax `(=> (argument-type ...) return-type)` definiert dabei über die Angabe der Typen der Argumente und des Rückgabewertes einen Funktionstyp, der hier auch für den ersten Parameter von `s48_create_joinable_thread` verwendet wurde. Der interne Typname `address` entspricht einem Zeiger auf ein Zeichen. Die folgenden beiden Ausdrücke entsprechen obigem C-Beispiel. Der erste definiert die Funktion `do-something`, der zweite erzeugt einen neuen Thread, der die Funktion `do-something` ausführt, und bindet die zurückgegebene Adresse der `pthread_t`-Variablen als Repräsentation des neuen Threads an den Namen `th`.

Diese Kapselung war notwendig, um die POSIX-Funktionen in Pre-Scheme verwenden zu können, hat aber darüber hinaus zwei Vorteile. Erstens ist der Pre-Scheme-Programmcode der virtuellen Maschine dadurch nicht direkt von der Verwendung einer POSIX-konformen Bibliothek abhängig. Damit ist eine

Portierung auf andere Bibliotheken, insbesondere für Windows, leichter möglich. Zweitens sind manche Funktionalitäten des POSIX-Standards nicht auf allen Plattformen verfügbar[4], sodaß diese unter Umständen, mittels bedingter Compilierung für diese Plattform simuliert werden können[5].

Die virtuelle Maschine kann mit den Mechanismen aus diesem Abschnitt Kernel-Level-Threads erzeugen, um in ihnen weitere virtuelle Prozessoren arbeiten zu lassen, und kann mit diversen Synchronisationsmitteln des Betriebssystems, darunter auch die in Abschnitt 1.2 beschriebenen Mutex-Locks und Condition-Variablen, den Zugriff der virtuellen Prozessoren auf gemeinsame Datenstrukturen und Resourcen kontrollieren.

2.2.2 Register und Stapel

Eine zentrale Rolle in der Implementierung der virtuellen Maschine spielt die Funktion `s48-restart`. Sie führt mithilfe einer großen Zahl von Hilfsfunktionen ein übergebenes Byte-Code-Programm aus und gibt sein Ergebnis zurück. Die Funktion verwendet dazu eine Reihe von Variablen als Register, und einen Speicherbereich in der Größe einiger Kilobytes als Ausführungsstapel. Die Arbeit eines virtuellen Prozessors von Scheme-48 besteht also im wesentlichen aus einem Aufruf von `s48-restart`. Die Register und der Ausführungsstapel eines virtuellen Prozessors sind seine einzigen spezifischen Daten, und bilden damit eine Art interner Repräsentation des virtuellen Prozessors. Da der Stackzugriff ebenfalls über bestimmte Register erfolgt, ist im folgenden nur noch von Registern die Rede.

In der bisherigen Implementierung der virtuellen Maschine sind die Registerwerte in *globalen* Variablen gespeichert. Dadurch sind die Registervariablen in den Umgebungen aller beteiligten Funktionen sichtbar, und können von ihnen entsprechend gelesen und verändert werden. Das erleichtert die Implementierung der einzelnen Teile, und wirkt sich zudem positiv auf die Effizienz aus. Für die Realisierung von mehreren virtuellen Prozessoren ergibt sich aber das Problem, daß jeder virtuelle Prozessor einen *eigenen* Satz Register benötigt. Eine naheliegende Lösung dieses Problem ist, die Register in einer Datenstruktur zusammenzufassen, der Funktion `s48-restart` einen solchen spezifischen *Registersatz* als Parameter zu übergeben, und die Zugriffe auf die globalen Variablen in Zugriffe in den Registersatz unzuwandeln. Da aber die Implementierung von `s48-restart` sehr lang, und vor allem auch in viele kleine Hilfsfunktionen aufgeteilt ist, ist mit dieser Lösung eine erhebliche Veränderung des Quelltextes verbunden. Eine solch große Veränderung ist aber insbesondere wegen der daraus resultierenden schlechten Portierbarkeit der Multiprozessorerweiterung auf zukünftige Scheme-48-Versionen möglichst zu vermeiden. Daher wurde für dieses Problem eine zwar etwas weniger effiziente Lösung implementiert, die aber dafür den Pre-Scheme-Quellcode weitgehend unverändert lässt.

Diese Lösung verwendet zum einen die Funktionalität des POSIX-Standards für sogenannte *thread-lokale* oder *thread-spezifische* Daten, und besteht zum anderen aus zwei Erweiterungen des Pre-Scheme-Compilers. Die erste Erweiterung des Compilers besteht in der Möglichkeit alle Lese- und Schreibzugriffe auf bestimmte Variablen automatisch durch Aufrufe von Lese- und Schreibfunktionen ersetzen zu lassen. Dazu erhält der Compiler eine Liste mit den betreffenden

[4] Beispielsweise implementiert die Bibliothek von Solaris 9 keine Spin-Locks.
[5] Pre-Scheme unterstützt selbst keine bedingte Compilierung.

Variablennamen und den einzusetzenden Namen der Lese- und Schreibfunktionen als Parameter `replace`. Diese Erweiterung wird dazu benutzt alle Zugriffe auf Registervariablen durch Aufrufe geeigneter Hilfsfunktionen zu ersetzen. Zur Verdeutlichung sei dies am Beispiel der Erhöhung des Registers für den Programmzähler demonstriert. Ausgangspunkt ist folgender Pre-Scheme-Code:

```
(set! *code-pointer* (address+ *code-pointer* 1))
```

Der Pre-Scheme-Compiler wird nun über einen entsprechenden Parameter angewiesen, den hierin enthaltene Lesezugriff auf die Registervariable `*code-pointer*` durch einen Aufruf der Funktion `get-*code-pointer*`, und den Schreibzugriff durch einen Aufruf der Funktion `set-*code-pointer*!` zu ersetzen. Das Ergebnis der Ersetzung ist also:

```
(set-*code-pointer*! (address+ (get-*code-pointer*) 1))
```

Auf die konkreten Definitionen der Funktionen `set-*code-pointer*!` und `get-*code-pointer*` wird weiter unten eingegangen. Zunächst muß dazu noch erklärt werden, wie ein virtueller Prozessor seinen Registersatz findet.

In der Initialisierungsphase eines neuen virtuellen Prozessors, das heißt bevor `s48-restart` von der Einstiegsfunktion eines neuen Kernel-Threads aufgerufen wird, wird ein neuer Registersatz erzeugt, und als sogenanntes *thread-spezifisches* Datum gespeichert. Dafür bietet der POSIX-Standard die Funktionen `pthread_getspecific` und `pthread_setspecific`, deren Prototypen folgendermaßen aussehen:

```
void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

Sie liefern beziehungsweise setzen zu einem Schlüssel einen Wert, der davon abhängt aus welchem Thread heraus diese Funktionen aufgerufen werden. Da jeder virtuelle Prozessor in einem eigenen Thread abläuft, lässt sich damit realisieren, daß an jeder beliebigen Stelle der Implementierung der Registersatz des virtuellen Prozessors abgerufen werden kann. Nun wäre es aber zu ineffizient, den Registersatz bei jedem einzelnen Zugriff auf ein Register auf diese Art auszulesen, daher wurde der Pre-Scheme-Compiler um ein weiteres Feature erweitert.

Diese Erweiterung bezieht sich auf den vom Compiler erzeugten C-Code: in jeder C-Funktion, die eine bestimmte globale Variable verwendet, kann die Erweiterung eine *lokale* Deklaration dieser Variablen einfügen. Die lokale Deklaration überdeckt dann eine gleichnamige, globale Variable. Den Namen der Variablen, sowie einen Initialisierungsausdruck für die lokale Variable, werden dem Compiler über einen Parameter gegeben. Der Name dieses Parameters ist `insert-local-declaration`.

In der Realisierung der virtuellen Prozessoren wird dieser Parameter dazu benutzt, um die Variable `*my-register-set*` in jeder C-Funktion lokal zu definieren. Wie der Name bereits andeutet, soll diese Variable immer genau den spezifischen Registersatz des virtuellen Prozessors enthalten. Die Initialisierung der Variablen besteht daher gerade darin, über die Funktion `pthread_getspecific` diesen thread-spezifischen Registersatz, und damit den Registersatz des virtuellen Prozessors, zu holen.

Die oben erwähnten Lese- und Schreibfunktionen für den Registerzugriff verwenden also gerade die Variable `*my-register-set*`, wodurch sie automatisch auf den Registersatz des jeweiligen virtuellen Prozessors zugreifen. Die Implementierung der Funktionen für den Programmzähler `*code-pointer*` sieht also folgendermaßen aus:

```
(define (get-*code-pointer*)
  (rs:code-pointer *my-register-set*))
(define (set-*code-pointer*! value)
  (set-rs:code-pointer! *my-register-set* value))
```

Die Funktion `rs:code-pointer` ist dabei der entsprechende Selektor, und `set-rs:code-pointer!` der Mutator der Registersatz-Datenstruktur. Damit der Pre-Scheme-Compiler diesen Code akzeptiert, muß noch eine globale Defintion von `*my-register-set*` vorhanden sein, die die virtuelle Maschine dann aber nicht verwendet[6].

In die generierten C-Funktionen, die `*my-register-set*` verwenden, fügt der Pre-Scheme-Compiler also automatisch eine Deklaration einer lokale Variablen ein, die mit dem spezifischen Registersatz des virtuellen Prozessors initialisiert wird. Dabei muß noch angemerkt werden, daß der Pre-Scheme-Compiler für kleine Funktionen, wie diese Register-Zugriffsfunktionen, keine C-Funktionen generiert, sondern deren Aufrufe direkt durch die Funktionsrümpfe ersetzt (sogenanntes *Inlining*). Dadurch ist die Anzahl der Aufrufe von `pthread_getspecific` um circa das 20.000-fache geringer als die Anzahl der Registerzugriffe.

Durch die zwei Erweiterungen des Pre-Scheme-Compilers, und der Verwendung der POSIX-Funktionen für thread-spezifische Daten, können die virtuellen Prozessoren also relativ effizient auf ihren spezifischen Registersatz zugreifen, ohne daß der Quelltext von Scheme-48 hierfür allzu sehr verändert werden musste.

2.2.3 Anwendung und Zusammenfassung

Aus Gründen der Kompatibilität mit der Uniprozessorversion von Scheme-48, und aufgrund der höheren Flexibilität, ist es sinnvoll, daß zusätzliche virtuelle Prozessoren erst zur Laufzeit des Scheme-Programms hinzugefügt werden können. Das bedeutet, Scheme-48 startet mit nur einem virtuellen Prozessor, und die virtuelle Maschine bietet ein Primitivum an, das einen neuen virtuellen Prozessor erzeugt, und die Interpretation einer bestimmten Funktion auf diesem Prozessor veranlasst. Der Name dieses Primitivums ist `spawn-processor`, und seine Signatur ist:

`(spawn-processor` ***thunk***`)` → `integer`

Das Primitivum nimmt eine Funktion ohne Parameter (einen sogenannten Thunk) als Argument, und gibt eine Ganzzahl zurück. Der übergebene Thunk ist die Funktion, die der neue virtuelle Prozessor ausführt, und der Rückgabewert ist eine eindeutige Identifikationsnummer für den virtuellen Prozessor.

[6] Eine weitere Modifikation des Pre-Scheme-Compilers, die dieses Problem behebt, wurde nicht implementiert.

`Spawn-processor` erzeugt also einen neuen Kernel-Level-Thread mit dem auszuführenden Thunk als Argument, und kehrt mit der Rückgabe einer eindeutigen Nummer zurück. Die einzelnen Schritte, die dann in dem neuen Thread ausgeführt werden, sind:

1. Allokation eines neuen Registersatzes und eines Aufführungsstapels.
2. Speicherung des Registersatzes als thread-spezifischen Wert für den neuen Kernel-Level-Thread.
3. Aufruf von `s48-restart` mit dem auszuführenden Thunk.
4. Deallokation des Registersatzes und Ausführungsstapels.

Das Hauptprogramm, also der erste virtuelle Prozessor, führt die ersten drei Schritte entsprechend beim Start von Scheme-48 durch.

Der Rückgabewert des virtuellen Prozessor, beziehungsweise des Thunks, wird grundsätzlich verworfen. Dies stellt jedoch keine Einschränkung der Funktionalität dar, weil der Programmierer ein Ergebnis leicht in einer dem Thunk und dem Erzeuger bekannten Variablen ablegen kann.

2.3 Speicherverwaltung

Scheme-48 verfügt über eine eigene Verwaltung des Speicherbedarfs von Scheme-Programmen. Die Speicherverwaltung kontrolliert alle Allokationen eines Programms, und gibt, durch eine regelmäßige Speicherbereinigung, den vom Programm nicht mehr verwendeten Speicher automatisch wieder frei. Eine explizite Deallokation durch das Programm ist daher nicht notwendig. Dieser Abschnitt erläutert zunächst die Grundlagen der automatischen Speicherverwaltung, und erklärt dann, wie die Speicherbereinigung und Allokation an die Multiprozessorsituation angepasst werden musste.

Es existieren sehr viele unterschiedliche Speicherbereinigungsverfahren (siehe dazu [21]). Das liegt einerseits an den unterschiedlichen Eigenschaften der Programmiersprachen, und andererseits an unterschiedlichen Zielsetzungen bestimmter Sprachimplementationen. Beispielsweise kann für C-Programme nur eine sogenannte *konservative* Speicherbereinigung implementiert werden, da die Sprache keine sichere Unterscheidung von Zeigern und Zahlen ermöglicht. Ferner gibt es Verfahren, die eine Bereinigung selten durchführen, dafür aber die normale Programmausführung lange unterbrechen, und es gibt Verfahren die eine Bereinigung in mehrere kleine Schritten unterteilen, zwischen denen das Programm weiter ausgeführt werden kann. Je nach Voraussetzungen der Sprache und den gewünschten Auswirkungen auf den Programmablauf, sind daher andere Verfahren zu wählen.

Scheme-48 enthält zwei alternative Speicherbereinigungsverfahren. Das erste ist ein klassischer sogenannter Two-Space-Copier mit einer relativ einfachen Implementierung, aber langen Unterbrechungszeiten. Das zweite Verfahren ist wesentlich komplexer, und soll unter anderem kürzere Unterbrechungszeiten realisieren [5]. Zum Zeitpunkt dieser Arbeit war das zweite Verfahren noch in einem Experimentierstadium, weshalb nur das einfachere Verfahren modifiziert wurde.

Die Speicherverwaltung ist für die Effizienz der Multiprozessorerweiterung von sehr großer Bedeutung. Das liegt erstens daran, daß die Allokationen in

Scheme-48 in sehr kleinen Einheiten, und damit auch sehr häufig erfolgen[7], sowie daran, daß durch die gemeinsame Verwendung des Speichers, die Allokationen der virtuellen Prozessoren synchronisiert werden müssen. Eine ineffiziente Implementierung der Allokation kann daher sehr schnell das Geschwindigkeitsplus einer Multiprozessormaschine zunichte machen. Ein weiterer Grund liegt in der Konstruktion heutiger symmetrischer Multiprozessormaschinen. Wie Uniprozessormaschinen spiegeln sie kleine Teile des Hauptspeichers in schnellere, aber teurere Zwischenspeicher, sogenannte *Caches*. Dadurch soll sich die durchschnittliche Zugriffszeit auf ein Speicherwort verringern, und damit der sogenannte Von-Neumann-Flaschenhals erweitert werden. Ein Cache besteht aus mehreren sogenannten *Cache-Lines*, in die der Prozessor jeweils einen zusammenhängenden Bereich von einigen Dutzend Wörtern aus dem Hauptspeicher kopiert. Da jeder Prozessor in Multiprozessormaschinen aber seinen eigenen Cache besitzt, muß bei Schreibvorgängen durch einen Prozessor unter Umständen der Cache von anderen Prozessoren akualisiert werden. Dies ist dann der Fall, wenn der andere Prozessor den selben Bereich des Hauptspeichers zwischengespeichert hat, und nach der Änderung erneut auf ihn zugreifen will. Das bedeutet aber auch, daß der andere Prozessor nicht unbedingt auf dasselbe Speicherwort zugreifen muß, sondern es genügt, wenn er ein Speicherwort in unmittelbarer Nähe lesen oder verändern will. Aus diesem Grund ist für eine optimale Effezienz des Interpreters eine möglichst große Trennung der Speicherbereiche, auf die die virtuellen Prozessoren jeweils zugreifen, notwendig. Die Speicherverwaltung muß daher versuchen diese *Lokalität* der Daten möglichst groß zu halten.

Der folgende Abschnitt geht zunächst auf einige Grundlagen der Speicherverwaltung in Scheme-48 ein. Die Abschnitte 2.3.3 und 2.3.2 beschreiben die Veränderungen der Speicherallokation beziehungsweise der Speicherbereinigung.

2.3.1 Grundlagen

Die Speicherverwaltung stellt der virtuellen Maschine die Möglichkeit zur dynamischen Allokation von Speicher zur Verfügung. Die Gesamtheit aller allokierten Speicherbereiche wird *Scheme-Heap* genannt. Die virtuelle Maschine beschreibt diesen Scheme-Heap mit sogenannten *Stored-Objects*, im folgenden kurz Objekte genannt. Diese Objekte müssen einer festen Struktur genügen. Sie beginnen mit einem Header-Wort, gefolgt vom Inhalt des Objekts. Der Header besteht aus einer Längenangabe des Objekts, einem 6-bit-Feld mit der Angabe eines Datentyps, sowie aus Implementationsgründen den Bits `10`. Zu diesen maximal 2^6 primitiven Datentypen gehören zum Beispiel Strings, Symbole, Paare und Vektoren. Die Speicherverwaltung muß innerhalb dieser primitiven Typen im wesentlichen zwei Kategorien unterscheiden: *Byte-Vektoren* und *Deskriptor-Vektoren*. Deskriptor-Vektoren können Zeiger auf andere Objekte enthalten; Byte-Vektoren enthalten beliebige Daten, aber keine Zeiger auf andere Objekte. Strings sind beispielsweise Byte-Vektoren, Paare sind Deskriptor-Vektoren. Der Inhalt eines Deskriptor-Vektors besteht aus Zeigern auf Objekte und aus sogenannten *Immediates*. Zu den Immediates gehören unter anderem boolsche Werte, Ganzzahlen von -2^{29} bis $2^{29}-1$ oder einzelne Zeichen. Immediates und Zeiger belegen jeweils nur ein Speicherwort. Entscheidend für die Speicherberei-

[7] Messungen mit einem der Benchmarkprogramme aus dieser Arbeit ergeben beispielsweise circa 75000 Anfragen an die Speicherverwaltung pro Sekunde auf einer Maschine mit einem 1,4-GHz-Prozessor.

nigung ist, daß die Speicherverwaltung durch die feste Strukturierung der Daten erstens die Länge eines Objekts kennt, und zweitens alle Zeiger auf Objekte finden kann.

Ausgangspunkt für die Speicherbereinigung ist das sogenannte *Root-Set*. Das Root-Set besteht aus denjenigen Werten, von denen das Programm *unmittelbar* Kenntnis hat, das heißt ohne dazu einen Zeiger derefenzieren zu müssen. Im wesentlichen sind das die Register und der Ausführungstapel der virtuellen Maschine. Diese enthalten, ebenso wie Deskriptor-Vektoren, Zeiger auf Objekte oder Immediates. Alle Objekte, die zu einem bestimmten Zeitpunkt von diesem Root-Set aus, durch eine Kette von Zeigern, erreichbar sind, werden als *lebendige* Objekte bezeichnet. Alle anderen Objekte im Speicher bezeichnet man entsprechend als *tote* Objekte. Da Scheme dem Programmierer keinen wahlfreien Zugriff auf den Speicher erlaubt, insbesondere keine Umwandlung von Zahlen in Zeiger, ist garantiert, daß das Programm von diesem Zeitpunkt an nie mehr auf die toten Objekte zugreifen kann. Die Speicherverwaltung kann also in einer Speicherbereinigung, mit der Kenntniss des Root-Sets, und der grundlegenden Struktur der Objekte, die *gestorbenen* Objekte identifizieren, und deren Speicherplatz für neue Allokationen freigeben.

Diese Suche und Freigabe erfolgt nach dem Prinzip des *Cheney-Algorithmus* [1]. Dieser Algorithmus verwendet zwei große, monolithische Speicherbereiche, die sogenannten *Semispaces*. Die Speicherverwaltung nutzt einen dieser Semispaces für neue Allokationen, indem sie bei jeder Allokationen einen Zeiger in diesen Semispace um die Anzahl der zu allokierenden Bytes erhöht. Dieser Zeiger wird *Free-Pointer* oder *Frontier* genannt. Beginnend am unteren Ende, füllt sich der Semispace also konsekutiv mit neuen Objekten. Ist das Ende des Semispaces erreicht, führt die Speicherverwaltung eine Speicherbereinigung durch. Dazu *kopiert* der Algorithmus zunächst alle Objekte, auf die das Root-Set einen Zeiger enthält, an den Anfang des zweiten Semispaces. Die Zeiger im Root-Set werden dabei durch die neuen Adressen der Objekte ersetzt, sowie die alten Objekte mit einem sogenannten *Forwarding-Pointer* markiert. Dieser kennzeichnet das Objekt einerseits als bereits kopiert, und hinterlässt gleichzeitig die neue Adresse des Objekts. Dies ist notwendig, weil mehrere Zeiger auf das selbe Objekt existieren können, aber ein Objekt nur einmal kopiert werden darf. Der Algorithmus beginnt dann damit, alle Objekte im zweiten Semispace nach Zeigern zu durchsuchen. Mit diesen Zeigern verfährt er genauso, wie mit den Zeigern im Root-Set, das heißt falls das verzeigerte Objekt nicht markiert ist, kopiert er es aus dem ersten in den zweiten Semispace, korrigiert den Zeiger, und markiert das alte Objekt mit einem Forwarding-Pointer. Ist es bereits kopiert, dann wird nur die neue Adresse eingetragen. Dadurch füllt sich der zweite Semispace erneut mit Objekten, die ebenfalls durchsucht werden müssen. Der Algorithmus wiederholt daher diese Schritte, bis er alle Objekte im zweiten Semispace durchsucht hat, beziehungsweise keine neuen Objekte mehr hinzu kommen. Im zweiten Semispace existieren dann keine Zeiger in den ersten Semispace mehr. Auf diese Weise kopiert der Algorithmus also alle *lebendigen* Objekte in den zweiten Semispace, und *kompaktiert* diese gleichzeitig an den Anfang des Semispaces. Die beiden Semispaces tauschen daraufhin ihre Rollen, indem die Speicherverwaltung den zusammenhängenden, oberen Bereich des zweiten Semispaces für neue Allokationen verwendet, und den ersten Semispace (und damit alle *toten* Objekte) verwirft, beziehungsweise bei der nächsten Speicherbereinigung als leer betrachtet.

An diesem prinzipiellen Ablauf der Speicherbereinigung wurde für die Multiprozessorerweiterung nichts verändert. Insbesondere ist die Speicherbereinigung nicht parallelisiert, das heißt sie wird nicht auf mehrere Prozessoren verteilt, sondern nutzt nur einen einzigen Prozessor. Eine Parallelisierung verbleibt als mögliche weitere Aufgabe [18, 9]. Die folgenden Abschnitte beschränken sich auf die in dieser Arbeit vorgenommenen Änderungen an der Speicherverwaltung.

2.3.2 Speicherbereinigung

Das Root-Set besteht in Scheme-48 im wesentlichen aus den Registern und dem Ausführungsstapel der virtuellen Maschine[8]. Entsprechend muß daher für die Multiprozessorerweiterung das Root-Set um die Register und die Ausführungsstapel aller zusätzlichen virtuellen Prozessoren erweitert werden. Außerdem ist das Speicherbereinigungsverfahren von Scheme-48 ein sogenanntes Stop-the-World-Verfahren, das heißt während der Speicherbereinigung darf die virtuelle Maschine weder neuen Speicher allokieren, oder Speicherobjekte verändern noch auf den Speicher zugreifen. Die virtuellen Prozessoren müssen daher während der Speicherbereinigung angehalten werden. Wie diese *Suspendierung* genau funktioniert, ist im folgenden beschrieben, gefolgt von der Beschreibung der Änderungen zur Erweiterung des Root-Sets.

Suspendierung Die virtuellen Prozessoren allokieren neuen Speicher indem sie bestimmte Funktionen der Speicherverwaltung aufrufen. In diesen Funktionen überprüft die Speicherverwaltung ob eine Bereinigung notwendig ist, und führt sie gegebenenfalls durch. Derjenige virtuelle Prozessor, bei dem diese Notwendigkeit eintritt, wartet also im Rahmen des normalen Kontrollflusses auf das Ende der Speicherbereinigung. Andere virtuellen Prozessoren laufen aber zunächst noch parallel weiter. Sie müssen auf andere Art und Weise angehalten werden. Dies kann jedoch nicht zu jedem beliebigen Zeitpunkt erfolgen, denn die virtuellen Prozessoren speichern während der Ausführung einer Instruktion Objektadressen außerhalb der Register und des Ausführungsstapels in lokalen Variablen. Diese Objekte müssten als lebendig betrachtet werden, obwohl unter Umständen weder ein Register noch ein Stapeleintrag auf sie verweist. Diese Adressen zum Root-Set hinzuzufügen wäre jedoch viel zu aufwendig, daher müssen die virtuellen Prozessoren in einer Situation anhalten, in der sie keine Objektadressen außerhalb des normalen Root-Sets speichern. Dies ist insbesondere zwischen der Ausführung zweier Instruktionen der Fall, aber auch zum Zeitpunkt des Aufrufs derjenigen Speicherverwaltungsfunktionen, die eine Speicherbereinigung auslösen können.

Die Suspendierung der virtuellen Maschine ist unabhängig von der Speicherverwaltung und den virtuellen Prozessoren implementiert. Das Ziel der Suspendierung ist es, jeden virtuellen Prozessoren in einen *blockierten* Zustand zu bringen, und ihn darin zu halten, bis die Suspendierung beendet wird. Die Speicherverwaltung initiiert eine Suspendierung über einen Aufruf der Funktion `suspend-vps`, die zurückkehrt wenn alle virtuellen Prozessor blockiert sind, und beendet eine Suspendierung über einen Aufruf der Funktion `continue-vps`.

Die Funktion `suspend-vps` muß dazu alle virtuellen Prozessoren kennen, weshalb in einer globalen Variablen für jeden virtuellen Prozessor eine Daten-

[8] Hinzu kommen noch Zwischenergebnisse in externem Code, sowie einige prozessorunabhängige Daten.

struktur mit einigen Informationen verwaltet wird. Die für die Suspendierung relevanten Informationen sind der Registersatz des virtuellen Prozessors, sowie eine Zustandsvariable, die angibt ob der virtuelle Prozessor blockiert ist. Ein Prozessor kann allerdings nicht nur aufgrund einer Suspendierung blockieren, sondern zum Beispiel auch wenn er auf Daten aus einer Datei wartet. In diesen Fällen müssen die Funktionen `enter-blocking-region` und `leave-blocking-region` aufgerufen werden. Wie die Suspendierung im Detail funktioniert, soll im folgenden Anhand der Implementierung der einzelnen Funktionen beschrieben werden.

Zunächst zur Funktion `suspend-vps`:

```
(define (suspend-vps)
  (get-vm-lock!)
  (cond
   (*suspension-pending?*
     (join-suspension/lock)
     (do-suspend))
   ((> *suspension-count* 0)
    (set! *suspension-count* (+ *suspension-count* 1)))
   (else
    (do-suspend))))
```

`Suspend-vps` aquiriert als erstes über die Funktion `get-vm-lock!` ein Mutex-Lock, über das die Suspendierung an mehreren Stellen synchronisiert wird (siehe dazu Abschnitt 1.2). Dann wird zunächst die globale boolsche Variable `*suspension-pending?*` überprüft. Dieses Flag gibt an, daß eine Suspendierung bereits initiiert wurde, jedoch noch nicht alle virtuellen Prozessoren angehalten haben. In diesem Fall muß also dieser virtuelle Prozessor zunächst auf das Ende der anderen Suspendierung warten, und danach die „eigene" Suspendierung durchführen. Ersteres geschieht durch einen Aufruf der Funktion `join-suspension/lock`, die weiter unten beschrieben wird, letzteres geschieht in der Hilfsfunktion `do-suspend`. Der zweite Test überprüft, ob die globale Variable `*suspension-count*` größer als Null ist. Über diesen Zähler wird eine *rekursive Suspendierung* realisiert, das heißt nachdem auf einem Kernel-Thread eine Suspendierung etabliert ist, führen weitere Aufrufe von `suspend-vps` nur zu einer Erhöhung des Zählers. Entsprechend verringert `continue-vps` den Zähler, und beendet die Suspendierung erst, wenn wieder Null erreicht ist. Mehrere Aufrufe von `suspend-vps` sind dabei möglich, weil das verwendete Mutex-Lock ein sogenanntes *rekursives* Mutex-Lock ist. Derselbe Thread kann ein solches Lock mehrmals aquirieren, und gibt es erst an andere Threads ab, wenn es genauso oft freigegeben wurde. Ist weder eine andere Suspendierung anhängig, noch eine rekursive Suspendierung gegeben, ruft `suspend-vps` direkt die Hilfsfunktion `do-suspend` auf, die die eigentliche Suspendierung vornimmt:

```
(define (do-suspend)
  (set! *suspension-pending?* #t)
  (set-vp:blocked?! (get-vp-data (get-*processor-id*)) #t)
  (for-each-register-set!
   (lambda (rs)
     (if (not (= (rs:processor-id rs) (get-*processor-id*)))
         (set-interrupt-flag/rs! rs))))
```

```
(for-each-vp!
 (lambda (vp)
  (let loop ()
    (if (not (vp:blocked? vp))
        (begin
          (s48-cond-wait *joined-suspension* (vm-lock))
          (loop))))))
(set! *suspension-count* (+ *suspension-count* 1))
(set! *suspension-pending?* #f))
```

`Do-suspend` setzt zunächst das oben bereits erwähnte globale Flag `*suspension-pending?*`, das gesetzt bleibt, bis alle Prozessoren blockiert sind. Der zweite Ausdruck, markiert den „aufrufenden" virtuellen Prozessor als blockiert, also den virtuellen Prozessor, der im aktuellen Kernel-Thread läuft. Dieser virtuelle Prozessor bestimmt sich über das Register `*processor-id*`, das mit der Funktion `get-*processor-id*` ausgelesen wird, und über die Funktion `get-vp-data`, die die oben erwähnte Datenstruktur für einen virtuellen Prozessor zurückgibt. `Set-vp-blocked?!` setzt schließlich das Blockiert-Flag in dieser Datenstruktur. Der nächste Schritt ist das Setzen des sogenannten *Interrupt-Flags* aller anderen virtuellen Prozessoren. Die Funktion `for-each-register-set!` ruft dazu die übergebene Funktion mit jedem Registersatz je einmal auf. Hier wird dies dazu benutzt für jeden Registersatz, der nicht zum „aktuellen" virtuellen Prozessor gehört, die Funktion `set-interrupt-flag/rs!` aufzurufen. Die Implementierung der Interrupt-Behandlung, auf die hier nicht genauer eingegangen werden soll, ist so gestaltet, daß bei gesetztem Interrupt-Flag eine bestimmte Funktion aufgerufen wird, die prüft ob und welche Ereignisse tatsächlich aufgetreten sind. Diese Funktion wurde um einen Aufruf der unten beschriebenen Funktion `maybe-join-suspension` erweitert. Das Setzten der Interrupt-Flags durch `set-interrupt-flag/rs!` garantiert also, daß alle aktiven virtuellen Prozessor kurze Zeit später `maybe-join-suspension`, beziehungsweise `join-suspension/lock` aufrufen (siehe unten). `Do-suspend` muß nun also warten, bis alle virtuellen Prozessoren im Blockiert-Zustand sind. Dazu erhält `do-suspend`, analog zu `for-each-register-set!`, die Datenstrukturen für jeden virtuellen Prozessor über die Funktion `for-each-vp!`. Für den Fall daß ein virtueller Prozessor noch nicht blockiert ist, muß `do-suspend` auf eine Änderung dieses Zustands warten. Dazu wird die Condition-Variable `*joined-suspension*` verwendet. Während `do-suspend` wartet, ist das Mutex-Lock freigegeben (zur Funktionsweise von Condition-Variablen siehe auch Abschnitt 1.2). Der Grund für die `loop`-Schleife ist einerseits, daß nur eine einzige Condition-Variable für alle virtuellen Prozessoren verwendet wird, und damit nicht sicher ist, daß wirklich der jeweilige virtuelle Prozessor die Zustandsänderung signalisiert hat. Andererseits hat die Funktion `s48-cond-wait`, die auf die Signalisierung der Condition-Variable wartet, die Eigenschaft gelegentlich zurückzukehren, ohne daß tatsächlich die Bedingung eingetreten ist. Ob der Blockiert-Zustand tatsächlich eingetreten ist, muß `do-suspend` daher noch einmal überprüfen, und gegebenenfalls erneut auf die Signalisierung der Bedingung warten. Vor den letzten zwei Ausdrücken in `do-suspend` steht damit fest, daß für alle virtuellen Prozessoren das Blockiert-Flag gesetzt ist, und das Mutex-Lock `vm-lock` aquiriert ist. Letzteres spielt weiter unten bei `leave-blocking-region` noch eine wichtige Rolle. Die verbleibenden Schritte sind dann noch die Erhöhung des Zählers für die rekur-

sive Suspendierung, und die Entfernung des `*suspension-pending?*`-Flags.

Zur Beendigung der Suspendierung dient die Funktion `continue-vps`:

```
(define (continue-vps)
  (set! *suspension-count* (- *suspension-count* 1))
  (if (= *suspension-count* 0)
      (begin
        (set-vp:blocked?! (get-vp-data (get-*processor-id*)) #f)
        (s48-cond-broadcast *suspension-finished*)))
  (release-vm-lock!))
```

Die Funktion verringert den Zähler `*suspension-count*`, und falls dieser Null ist, entfernt sie das Blockiert-Flag des virtuellen Prozessors, der im aktuellen Kernel-Thread läuft. Desweiteren signalisiert sie die Condition-Variable `*suspension-finished*` durch ein sogenanntes *Broadcast*. Alle blockierten virtuellen Prozessoren warten auf die Signalisierung dieser Condition-Variablen (siehe unten), und ein Broadcast führt dazu, daß alle wartenden Kernel-Threads dieses Signal empfangen. Schließlich gibt `continue-vps` das rekursive Mutex-Lock `vm-lock` frei, das, wie oben beschrieben, im Fall der rekursiven Suspendierung tatsächlich noch aquiriert bleibt.

Über die beiden folgenden Funktionen warten die virtuellen Prozessoren auf das Ende einer Suspendierung:

```
(define (join-suspension/lock)
  (let ((this-vp (get-vp-data (get-*processor-id*))))
    (set-vp:blocked?! this-vp #t)
    (s48-cond-signal *joined-suspension*)
    (let loop ()
      (s48-cond-wait *suspension-finished* (vm-lock))
      (if (or *suspension-pending?* (> 0 *suspension-count*))
          (loop)))
    (set-vp:blocked?! this-vp #f)))

(define (maybe-join-suspension)
  (get-vm-lock!)
  (if *suspension-pending?*
      (join-suspension/lock))
  (release-vm-lock!))
```

`Join-suspension/lock` setzt zunächst das Blockiert-Flag des aufrufenden virtuellen Prozessors, und signalisiert dies, wie oben erwähnt, über die Condition-Variable `*joined-suspension*`. Hierbei könnte es aber zu einer sogenannten *Race-Condition* kommen. Das bedeutet, daß wenn in `do-suspend` zwischen der Überprüfung des Blockiert-Flags und dem Warten auf die Condition-Variable, sowohl das Setzen des Flags als auch die Signalisierung ausgeführt werden, dann „verpasst" `do-suspend` dieses Signal. Unter anderem aus diesem Grund muß `join-suspension/lock` mit aquiriertem Mutex-Lock `vm-lock` aufgerufen werden. Dadurch kann hier die Condition-Variable erst signalisiert werden, wenn `do-suspend` tatsächlich darauf wartet, da es dort erst dann freigegeben wird. Alle Aufrufer von `join-suspension/lock`, darunter auch `maybe-join-suspension`, müssen dies garantieren. Der folgende Aufruf von `s48-cond-wait` gibt

dieses Lock vorübergehend wieder frei, während er auf die oben erwähnte Bedingung `*suspension-finished*` wartet. Das heißt `s48-cond-wait` kehrt zurück, wenn die Suspendierung beendet ist. Genauso wie in `do-suspend`, kann diese Funktion aber auch spontan zurückkehren, daher wird hier erneut gewartet, falls entweder `*suspension-pending?*` gesetzt ist, oder `*suspension-count*` größer als Null ist. Dieser Fall kann ebenfalls eintreten, wenn nach dem Aufruf von `continue-vps` sofort eine dritter virtueller Prozessor eine neue Suspendierung initiiert. Besteht die Suspendierung aber nicht mehr, dann entfernt `join-suspension/lock` schließlich das Blockiert-Flag und kehrt zurück. Der virtuelle Prozessor kann also daraufhin fortfahren. `Maybe-join-suspension` prüft lediglich das `*suspension-pending?*`-Flag, ruft `join-suspension/lock` auf und stellt dabei sicher, daß das Mutex-Lock `vm-lock` aquiriert ist.

Verbleiben noch die beiden Funktionen `enter-blocking-region` und `leave-blocking-region`. Die virtuellen Prozessoren müssen diese Funktionen immer vor, beziehungsweise nach Bereichen aufrufen, in denen sie Funktionen aufrufen, die den Thread auf unbestimmte Zeit blockieren können. Dies ist beispielsweise der Fall, wenn ein virtueller Prozessor auf Daten aus einer Datei wartet. Diese Funktionen sind folgendermaßen definiert:

```
(define (enter-blocking-region)
  (get-vm-lock!)
  (if *suspension-pending?*
      (join-suspension/lock)
  (set-vp:blocked?! (get-vp-data (get-*processor-id*)) #t)
  (release-vm-lock!))

(define (leave-blocking-region)
  (get-vm-lock!)
  (if *suspension-pending?*
      (join-suspension/lock))
  (set-vp:blocked?! (get-vp-data (get-*processor-id*)) #f)
  (release-vm-lock!))
```

Beide Funktionen unterscheiden sich in der Implementierung nur darin, daß `enter-blocking-region` das Blockiert-Flag des aufrufenden virtuellen Prozessors setzt, und `leave-blocking-region` es entfernt. `Enter-blocking-region` muß zunächst das Mutex-Lock `vm-lock` aquirieren um Race-Conditions zu verhindern. Dann prüft die Funktion, ob eine Suspendierung anhängig ist, und ruft `join-suspension/lock` auf, wenn dies der Fall ist. Das ist notwendig, weil `do-suspend` eventuell auf die Signalisierung des Blockiert-Zustands wartet, und das folgende Setzen des Blockiert-Flags dazu nicht ausreicht. Schließlich gibt `enter-blocking-region` das Mutex-Lock `vm-lock` wieder frei, und der virtuelle Prozessor kann die blockierenden Funktionen ausführen. In dieser Zeit könnte nun ein anderer Kernel-Thread eine Suspendierung initiieren oder sogar etablieren, da einerseits das Mutex-Lock `vm-lock` freigegeben ist, und andererseits das Blockiert-Flag dieses virtuellen Prozessors gesetzt ist. Beim Aufruf von `leave-blocking-region` kann also eine Suspendierung sowohl bereits etabliert, als auch noch anhängig sein. Im ersten Fall blockiert der Aufruf von `get-vm-lock!`, da das Mutex-Lock während der gesamten Suspendierung aquiriert bleibt. Ist eine Suspendierung allerdings noch anhängig, dann

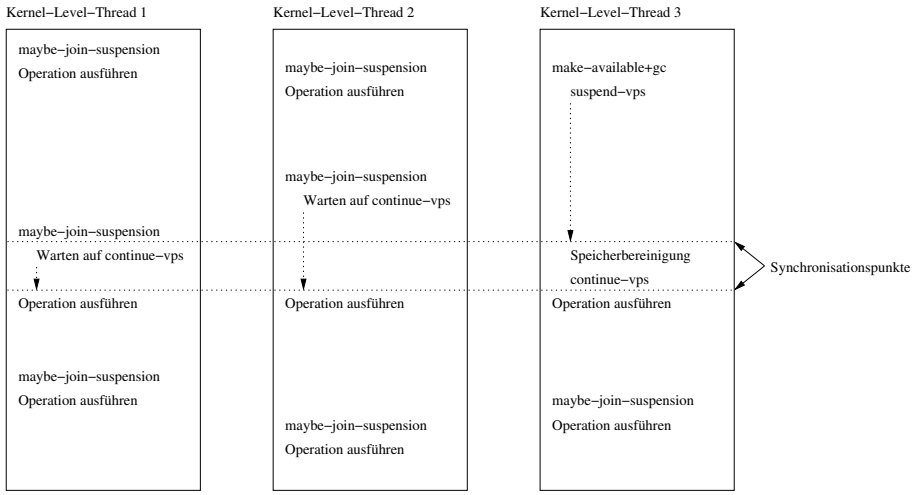

Abbildung 6: Ablauf einer Suspendierung der virtuellen Prozessoren für eine Speicherbereinigung

muß `leave-blocking-region` ebenfalls `join-suspension/lock` aufrufen, da `do-suspend` bereits das gesetzte Blockiert-Flag dieses Prozessors gesehen haben könnte, und daher davon ausgeht, daß dieser Prozessor bereits auf das Ende der Suspendierung wartet. Die in diesem Fall überflüssige Signalisierung der Condition-Variablen `*joined-suspension*` ist aufgrund der Implementierung von `do-suspend` nicht problematisch. Ist keine Suspendierung anhängig, oder wurde sie beendet, entfernt `leave-blocking-region` schließlich das Blockiert-Flag, gibt das `vm-lock` wieder frei und kehrt zurück.

Ein Ablaufdiagramm dieser Suspendierung mit drei virtuellen Prozessoren, beziehungsweise Kernel-Threads, ist in Abbildung 6 dargestellt. Darin ist zunächst dargestellt, daß jeder virtuelle Prozessor zwischen der Ausführung zweier Operationen `maybe-join-suspension` aufruft. Auf dem dritten Prozessor tritt dann die Situation ein, daß in einem Aufruf von `make-available+gc` festgestellt wird, daß eine Speicherbereinigung notwendig ist. Der Aufruf von `suspend-vps` kehr daraufhin erst zurück, wenn in den beiden anderen Threads erneut `maybe-join-suspension` aufgerufen wurde, was durch den gestrichelten Pfeil angedeutet werden soll. Ebenso warten die beiden ersten Threads dann auf den Aufruf der Funktion `continue-vps`, die im dritten Thread nach der Durchführung der Speicherbereinigung aufgerufen wird. Markiert sind außerdem zwei sogenannte *Synchronisationspunkte*, das sind Zeitpunkte in denen alle Threads einen definierten Zustand erreicht haben.

Neben der Speicherbereinigung, muß die Speicherverwaltung diese Suspendierung auch für die Funktionen `s48-gather-objects`, `s48-find-all` und `s48-find-all-records` durchführen. Diese Funktionen durchsuchen den Speicher nach bestimmten Objekten, und benötigen dazu einen konsistenten Speicher, der während der Suche nicht durch andere virtuelle Prozessoren verändert wird.

Erweiterung des Root-Sets Nachdem die Suspendierung hergestellt ist, die virtuellen Prozessoren also angehalten haben, benötigt die Speicherverwaltung den Inhalt des Root-Sets, also der Register und Ausführungsstapel al-

ler virtuellen Prozessoren. Scheme-48 enthält dafür bereits eine Schnittstelle zwischen der Speicherverwaltung und der virtuellen Maschine. Die Speicherverwaltung ruft dabei die in der virtuellen Maschine implementierte Funktion `s48-gc-roots` auf. `S48-gc-roots` ruft wiederum, ähnlich einer Callback-Funktion, die Funktion `s48-trace-value` der Speicherverwaltung für jeden Registerwert und Stapeleintrag auf. Wenn es sich um einen Zeiger auf ein Objekt handelt, kopiert `s48-trace-value` dieses Objekt in den neuen Speicherbereich, und gibt seine neue Adresse zurück. `S48-gc-roots` schreibt diese neue Adresse dann in das Register, beziehungsweise den Stapel zurück. Diese Übermittlung muß `s48-gc-roots` nun also für alle Registersätze und Ausführungsstapel der virtuellen Prozessoren durchführen. Dies ist, neben der Suspendierung, ein weiterer Grund für die globale Verwaltung von Informationen über alle virtuellen Prozessoren. So erhält der Speicherbereinigungsalgorithmus also den Inhalt des gesamten Root-Sets und kann, wie in Abschnitt 2.3.1 beschrieben, alle lebendigen Objekte in den neuen Speicherbereich kopieren. Ist die Speicherbereinigung durchgeführt, dann beendet die Speicherverwaltung die Suspendierung über einen Aufruf der Funktion `continue-vps`.

2.3.3 Allokation

Die Speicherverwaltung stellt den virtuellen Prozessoren zwei verschiedene Methoden zur Allokation von Speicher für neue Objekte zur Verfügung. Die erste Methode besteht aus einer einzelnen Funktion namens `s48-allocated-traced-+gc`, die die Größe des gewünschten Speicherbereichs als Parameter erhält, und die Startadresse des reservierten Bereichs zurück gibt. Die zweite Methode ist das sogenannte *Preallocation-System*, auf das weiter unten näher eingegangen wird. Die Speicherverwaltung verwendet zur Allokation einen einzelnen großen Speicherbereich, einen sogenannten *Semispace*, und einen Zeiger auf die nächste freie Stelle in diesem Bereich. Eine Allokation besteht also aus dem Zwischenspeichern dieser Adresse, dem Erhöhen der Adresse um die Anzahl der zu allokierenden Bytes, und dem Speichern dieser neuen Adresse für die nächste Allokation. In der Multiprozessorsituation könnten nun mehrere virtuelle Prozessoren versuchen, gleichzeitig Speicher zu allokieren. Dies würde beispielsweise bedeuten, daß zwei virtuelle Prozessor dieselbe Startadresse erhalten, weil der erste die erhöhte Adresse noch nicht zurückgeschrieben hat, wenn der zweite die Adresse bereits zwischengespeichert hat. Die Allokation stellt also einen kritischen Abschnitt dar, der vor gleichzeitigen Aufrufen geschützt werden muß. Die Speicherverwaltung verwendet dazu ein sogenanntes *Spin-Lock*. Spin-Locks zeichnen sich dadurch aus, daß der wartende Thread in einer Schleife immer wieder versucht das Lock zu aquirieren (sogenanntes Busy-Waiting). Gleichzeitig ist die Aquirierung eines *freien* Spin-Locks aber auch sehr schnell. Spin-Locks eignen sich daher für sehr kleine kritische Abschnitte, das heißt für Locks, die jeweils nur für sehr kurze Zeit gehalten werden. Die Erhöhung des Zeigers auf die nächste freie Stelle im Speicher ist ein solcher sehr kurzer kritischer Abschnitt.

Diese Allokationsmethode ist allerdings für die Arbeit eines virtuellen Prozessors problematisch[9], da er beispielsweise für die Ausführung einer Instruktion die Adressen mehrerer Objekte vom Stapel holt, und Ergebnisse in mehrere neu allokierte Objekte schreibt. Wäre bei jeder Allokation eine Speicherberei-

[9] Diese Methode wird jedoch an einigen Stellen verwendet.

nigung möglich, und damit eine Änderung der Adressen dieser Objekte, müsste ein erheblicher Aufwand betrieben werden, um die Ausführung der Instruktion mit den neuen Adressen fortsetzen zu können. Die Implementierung des virtuellen Prozessors ist daher einfacher, wenn die Speicherverwaltung garantieren kann, daß sie während der Berechnung der Ergebnisse einer Instruktion, beziehungsweise der Allokation der Ergebnisobjekte, keine Speicherbereinigung durchführt. Dafür stellt die Speicherverwaltung das sogenannte *Preallocation-System* bereit, das aus zwei Funktionen besteht: `s48-make-available+gc` und `s48-allocate-small`. Beide Funktionen haben einen Integer als Parameter, der die Anzahl von Bytes angibt, die reserviert beziehungsweise allokiert werden sollen. Zu Beginn einer Instruktion kann ein virtueller Prozessor `s48-make-available+gc` aufrufen, um sich von der Speicherverwaltung die Verfügbarkeit einer bestimmten Anzahl von Bytes garantieren zu lassen. Das heißt, die Speicherverwaltung kann hierfür eine Speicherbereinigung durchführen, garantiert aber, daß der virtuelle Prozessor in der weiteren Ausführung einer Instruktion mindestens diese Anzahl von Bytes durch einen oder mehrere Aufrufe von `s48-allocate-small` allokieren kann, ohne daß die Speicherverwaltung dabei eine Speicherbereinigung durchführt. Dies zu garantieren, ist in der Multiprozessorsituation relativ kompliziert, denn eine bestimmte Anzahl Bytes, die zu einem Zeitpunkt noch frei ist, kann in nächsten Moment von einem anderen virtuellen Prozessor allokiert werden. Hinzu kommt, daß die virtuellen Prozessoren die garantierte Speichermenge nicht immer voll ausschöpfen, das heißt sie lassen sich in einer Instruktion teilweise mehr Speicher garantieren, als sie tatsächlich allokieren.

Das Preallocation-System ist für die Effizienz der Multiprozessorerweiterung von sehr großer Bedeutung, denn einerseits wird es sehr häufig aufgerufen, und andererseits kann hierüber eine bessere Lokalität der Speicherzugriffe eines virtuellen Prozessors bewirkt werden. Wie oben erwähnt, ist die Lokalität wichtig, um möglichst wenig des Geschwindigkeitsvorteils durch Caching des Hauptspeichers auf (realen) Multiprozessormaschinen einzubüßen.

Diese Arbeit erweitert das Preallocation-System um eine prozessorspezifische Datenstruktur. Diese Datenstruktur enthält die Anfangs- und die Endadresse eines speziell für den einzelnen virtuellen Prozessor allokierten Speicherbereichs – *Creation-Space* genannt. Die Preallocation-Funktionen `s48-make-available+gc` und `s48-allocate-small` operieren nun ausschließlich innerhalb dieses Creation-Spaces. Das heißt `s48-make-available+gc` prüft, ob der Creation-Space mindestens die von dem virtuellen Prozessor angefragte Größe hat, und allokiert einen neuen Creation-Space wenn dies nicht mehr der Fall ist. Entsprechend gibt `s48-allocate-small` die Start-Adresse des Creation-Spaces zurück, und erhöht sie um die Anzahl der übergebenen Bytes. Da für jeden virtuellen Prozessor ein spezifischer Creation-Space zur Verfügung steht, muß die Prüfung des noch freien Speichers, und die Erhöhung der Startadresse nicht mit anderen virtuellen Prozessoren synchronisiert werden. Nur für die Allokation eines neuen Creation-Spaces ist eine Synchronisierung notwendig. `S48-make-available+gc` ruft dazu die oben beschriebene Funktion `s48-allocate-traced+gc` auf.

Auf die Creation-Spaces muß in zwei weiteren Situationen zugegriffen werden. Zum einen setzt die Speicherverwaltung die Creation-Spaces am Ende einer Speicherbereinigung zurück, das heißt sie setzt Anfangs- und Endadressen auf Null, woraufhin beim nächsten Zugriff durch den virtuellen Prozessor ein neuer Creation-Space allokiert wird. Dies ist notwendig, da die Adressen nach der

Speicherbereinigung in den alten, ungültigen Semispace zeigen. Zum anderen muß in den Funktionen, die den Speicher nach bestimmten Objekten durchsuchen, der ungenutzte Platz in einem Creation-Space zuvor mit einem Pseudo-Objekt beschrieben werden. Dies ist notwendig, da die Suchfunktionen nicht nur die vom Root-Set aus erreichbaren Objekte finden, sondern den allokierten Speicher vollständig durchsuchen. Sie gehen dabei davon aus, daß der gesamte allokierte Speicher mit gültigen Objekten beschrieben ist. Da ein Creation-Space aber einen allokierten, unbeschriebenen Bereich darstellt, muß an den Anfang dieses Bereichs ein entsprechender Objekt-Header geschrieben werden[10]. In beiden Situationen besteht aber eine Suspendierung der virtuellen Prozessoren, sodaß der unsynchronisierte Zugriff auf die Creation-Spaces möglich ist.

Die Größe eines neuen Creation-Spaces ist ein Optimierungsproblem. Ist er zu klein, dann muß die Speicherverwaltung zu viele synchronisierte Allokationen durchführen. Ist er zu groß, dann müssen mehr Speicherbereinigungen durchgeführt werden als nötig, da ungenutzter Platz in einem Creation-Space nicht von anderen virtuellen Prozessoren verwendet werden kann. Das Optimum ist zudem abhängig vom ausgeführten Programm und der Anzahl der virtuellen Prozessoren, sodaß eventuell eine Adaption der Größe zur Laufzeit vorteilhaft sein könnte. Von einer Optimierung ist jedoch insgesamt nur eine kleine Effizienzsteigerung zu erwarten, daher wurde im Rahmen dieser Arbeit die Mindestgröße auf 128 Kilobytes festgelegt. Die maximale Objektgröße beschränkt dies nicht, da bei einer größeren Reservierung ein entsprechend größerer Creation-Space allokiert wird.

Die Preallocation-Funktionen konnten so sehr einfach und effizient implementiert werden. Zur weiteren Effizienzsteigerung wird die Datenstruktur mit den beiden Zeiger des Creation-Spaces nicht über die POSIX-Funktionen für thread-spezifische Daten gespeichert (siehe Abschnitt 2.2.2), sondern von den virtuellen Prozessoren in einem speziellen Register abgelegt, und den beiden Preallocation-Funktionen als zusätzlichen Parameter übergeben[11]. Jeder virtuelle Prozessor ruft dazu in seiner Initialisierungsphase, und nach seiner letzten Allokation, die in der Speicherverwaltung implementierten Funktionen `s48-create-gc-processor-data` beziehungsweise `s48-free-gc-processor-data` auf. Die erste Funktion erzeugt eine Datenstruktur für die beiden Zeiger des Creation-Spaces und gibt einen Zeiger darauf zurück. Die zweite gibt die Datenstruktur wieder frei. Dadurch kann der konkrete Inhalt der Datenstruktur dem virtuellen Prozessor verborgen bleiben.

2.4 I/O-System

Für den Zugriff auf Ein-/Ausgabekanäle, also zum Beispiel Dateien oder Sockets, stellt die virtuelle Maschine dem Laufzeitsystem sogenannte „Unbuffered Non-Blocking Channels" zur Verfügung. „Non-blocking" bedeutet, daß die primitiven Operationen der virtuellen Maschine auch dann sofort zurückkehren, wenn eine Lese- oder Schreiboperation nicht sofort durchführbar ist. „Unbuffered" bedeutet, daß die virtuelle Maschine keine Daten zwischenspeichert, die vom Lauf-

[10] Aus dem selben Grund muß bei der Allokation eines neuen Creation-Spaces der ungenutzte Bereich im alten Creation-Space gefüllt werden.

[11] Im Gegensatz zu den Registerzugriffen, war im Quellcode schon über die Funktionen des Preallocation-Systems abstrahiert, weshalb jeweils nur an einer einzigen Stelle im Quellcode der zusätzliche Parameter hinzugefügt werden musste.

zeitsystem entweder noch nicht geschrieben oder gelesen worden sind[12].

Die Realisierung dieses Systems basiert auf insgesamt drei globalen Listen in der virtuellen Maschine, und dem auch in Abschnitt 3.2.2 beschriebenen `wait`-Primitivum. Um das I/O-System für die Multiprozessorsituation thread-sicher zu machen, wurden die Zugriffe auf alle diese Listen mit Mutex-Locks geschützt. Die erste Liste enthält für jeden offenen Ein-/Ausgabekanal einen Zeiger auf ein spezielles `Channel`-Objekt. Die virtuelle Maschine benötigt diese Liste, um sicherzustellen, daß für jeden Filedeskriptor des Betriebsystems nur ein `Channel`-Objekt existiert. Die zweite Liste ist eine Warteschlange mit Zeigern auf diejenigen Kommunikationskanäle, für die eine Lese- oder Schreiboperation noch anhängig ist. Die virtuelle Maschine verwendet diese Warteschlange für die Realisierung der sogenannten „I/O-Completion"-Interrupts. Die virtuelle Maschine löst einen solchen Interrupt für einen anhängigen Kanal aus, wenn sie feststellt, daß die entsprechende Lese- oder Schreiboperation möglich geworden ist. Damit die virtuelle Maschine dies feststellen kann, werden bei Lese- und Schreiboperationen, die nicht sofort erfolgreich durchgeführt werden konnten, die zugrundeliegenden Filedeskriptoren in einer weiteren Liste zwischengespeichert. Um darauf zu warten, kann das Laufzeitsystem das Primitivum `wait` aufrufen, in welchem genau diese Filedeskriptoren überprüft, und gegebenenfalls für die Auslösung der Interrupts vorgemerkt werden.

Im Laufzeitsystem wird diese von der virtuellen Maschine angebotene Infrastruktur, also die nicht-blockierenden Lese- und Schreiboperationen und die I/O-Completion-Interrupts, genutzt, um *blockierende* Lese- und Schreibfunktionen zu implementieren, die also erst zurückkehren, wenn die entsprechende Operation erfolgreich durchgeführt worden ist. An dieser Implementierung mußten jedoch keine Veränderungen vorgenommen werden, weshalb hier nicht weiter darauf eingegangen werden soll.

[12] In der Regel puffert das Betriebsystem solche Daten. Für die Schnittstelle der virtuellen Maschine zum Laufzeitsystem spielt dies jedoch keine Rolle.

3 Das Laufzeit-System

Sprachimplementierungen enthalten häufig ein sogenanntes Laufzeit-System, oder eine Laufzeit-Bibliothek. Das Laufzeit-System beinhaltet dabei Implementierungen von häufig wiederkehrenden Aufgaben eines Programmierers, und erleichtert so das Programmieren erheblich. Die Grenze zwischen Laufzeit-System und Sprache verschwimmt dabei häufig, da es für den Programmierer zumeist unerheblich ist, ob eine bestimmte Funktion oder ein Konstrukt im Laufzeit-System implementiert ist, oder vom Compiler direkt in Maschinencode übersetzt wird. Dies ist insbesondere dann der Fall, wenn eine standardisierte Schnittstelle zum Laufzeitsystem existiert. Beispielsweise würden die meisten Programmierer die Funktion `malloc()` als Teil der Sprache C betrachten, obwohl sie Teil der Laufzeit-Bibliothek „Standard C Library" [14] ist.

Das Laufzeit-System von Scheme-48 enthält unter anderem den Compiler, der Scheme-Code in Code für die virtuelle Maschine übersetzt, ein Modulsystem, einen Kommando-Prozessor zum Laden und zur Eingabe von Scheme-Programmen und -Modulen zur Laufzeit, vereinfachte I/O-Funktionen, sowie eine Unterstützung für User-Level-Threads. Insbesondere das Thread-System ist für diese Arbeit von zentraler Bedeutung. Die folgenden drei Abschnitte befassen sich daher ausschließlich mit dem Thread-System. Abschnit 3.1 beschreibt zunächst die Grundlagen das Thread-System, Abschnitt 3.2 befasst sich mit dem Multitasking auf mehreren virtuellen Prozessoren, und Abschnitt 3.3 behandelt schließlich das automatische Verteilen der Threads.

3.1 Grundlagen des Thread-Systems

Abschnitt 2 hat die Veränderungen an der virtuellen Maschine beschrieben, die die *echt* nebenläufige Ausführung einer Funktion auf einem neuen virtuellen Prozessor ermöglichen. Ein Ziel dieser Arbeit ist es aber, daß parallelisierte Scheme-Programme die Rechenleistung einer Multiprozessormaschine *automatisch* ausnutzen können. Voraussetzung dafür ist die Parallelisierung eines Programms, das heißt die Aufteilung des Programms in parallel ausführbare Teile. Scheme-48 ermöglicht bereits die Parallelisierung von Programmen über *Threads*, und die *scheinbar* nebenläufige Ausführung der Threads durch sogenanntes *präemptives Multitasking*. Thread und Task, beziehungsweise Multitasking und Multithreading sind hier Synonyme. Der Prozessor arbeitet dabei zu einem bestimmten Zeitpunkt zwar immer nur einen einzelnen Thread ab, aber durch häufiges Wechseln dieses aktiven Threads, wird der Eindruck der gleichzeitigen Ausführung aller Threads erweckt. Das sogenannte *Scheduling* bestimmt dazu in Multitasking-Systemen, wann welcher Task vom Prozessor abgearbeitet wird. Abschnitt 3.1.1 geht näher auf das Scheduling in Scheme-48 ein.

Beim *präemptiven* Multitasking findet, im Gegensatz zum *kooperativen* Multitasking, der Thread-Wechsel *zwangsweise* statt, erfordert also keine *Kooperation* durch den aktiven Thread. Die Thread-Wechsel sind dadurch für den Programmierer *nichtdeterministisch*, das heißt er muß davon ausgehen, daß ein Thread an jeder beliebigen Stelle angehalten, und die Ausführung eines anderen Threads fortgesetzt wird. Dieser Nichtdeterminismus ist in manchen Situationen für die Ausführung des Programms problematisch. Dies ist häufig dann der Fall, wenn die Threads auf gemeinsame Datenstrukturen zugreifen und diese verändern. Solche Bereiche im Programm werden häufig als *kritische Abschnit-*

te bezeichnet. Der Programmierer muß die mehrfache verschränkte Ausführung solcher kritischen Abschnitte mithilfe sogenannter *Synchronisierungsmittel* verhindern. Beim Multitasking können weitgehend dieselben Synchronisierungsmittel wie bei einer *echt* gleichzeitigen Ausführung der Threads eingesetzt werden, allerdings hat man in der Regel zusätzlich die Möglichkeit den *Thread-Wechsel* kurzfristig zu verhindern, um einen kritischen Abschnitt zu schützen. In Scheme-48 ist zwar nicht vorgesehen, daß der Benutzer diese Methode verwendet, jedoch basiert die *Implementierung* der grundlegenden Synchronisierungsmittel von Scheme-48 teilweise auf dieser Möglichkeit den Thread-Wechsel zu verhindern. Für die echt nebenläufige Ausführung, die mit dieser Arbeit ermöglicht wird, mussten diese Implementierungen daher entsprechend verändert werden. Abschnitt 3.1.2 beschreibt die grundlegenden Verfahren zur Synchronisierung in Scheme-48, und die Veränderungen in deren Implementierung für die Multiprozessorerweiterung.

3.1.1 Scheduling

Das Thread-System von Scheme-48 ist ein User-Level-Thread-System. Das heißt, das Laufzeitsystem, und nicht die virtuelle Maschine sorgt für die Verteilung der verfügbaren Rechenzeit auf die einzelnen Threads, bestimmt also über das sogenannte *Scheduling* der Threads. Das Scheduling entscheidet die Fragen wann, welcher Thread, wie lange ausgeführt wird. Das Laufzeitsystem bedient sich dazu zweier Einrichtungen der virtuellen Maschine: *Continuations* und *Interrupts*. Diese sollen im folgenden zunächst erklärt werden, bevor näher auf das Scheduling in Scheme-48 eingegangen wird.

Continuations Eine Continuation ist eine Repräsentation der *Zukunft* des Programms in einem bestimmten Punkt der Ausführung. Anders ausgedrückt, ist bei jeder Auswertung eines Ausdrucks zur Laufzeit, die *momentane Continuation* eine Repräsentation dessen, was mit dem *Ergebnis* dieses Ausdrucks im weiteren Programmverlauf gemacht wird. Scheme bietet dem Programmierer die Möglichkeit eine Continuation *einzufangen* und *wieder einzusetzen*. Einfangen bedeutet dabei, ein Objekt zu erhalten, das die Continuation eines bestimmten Ausdrucks repräsentiert, und das der Programmierer wie jedes andere Objekt für eine spätere Verwendung zwischenspeichern kann. Beim Wiedereinsetzen einer Continuation springt der Interpreter in den Kontext dieses Ausdrucks zurück. Dazu muß ein Wert angegeben werden, der quasi anstelle des Ausdrucks in diesen Kontext eingesetzt wird. Das Wiedereinsetzen ist dabei nicht mit einem Funktionsaufruf zu verwechseln, denn die momentane Continuation wird dabei *verworfen*, das heißt, das Wiedereinsetzen einer Continuation kehrt *nicht* zurück. Zwischen dem Einfangen und dem Einsetzen einer Continuation vorgenommene Mutationen an Speicherobjekten bleiben dabei allerdings erhalten. Der Scheme-Standard R^5RS [12] spezifiziert die Funktion `call-with-current-continuation` für das Einfangen der momentanen Continuation. Die Repräsentation der Continuation ist dabei eine Funktion, deren Aufruf die Wiedereinsetzung der Continuation bewirkt.

Folgendes Beispiel soll die Verwendung von Continuations verdeutlichen:

```
(define (sum-list lst)
  (call-with-current-continuation
```

```
   (lambda (k)
     (sum-list-2 lst k))))

(define (sum-list-2 lst return)
  (if (null? lst)
      0
      (+ (if (number? (car lst))
             (car lst)
             (return "Error"))
         (sum-list-2 (cdr lst) return))))
```

Bei einem Aufruf der Funktion `sum-list` wird zunächst die momentane Continuation eingefangen und an den Namen `k` gebunden. Diese Continuation repräsentiert also alles, was nach der Rückkehr von `sum-list` an der Aufrufstelle mit dem Rückgabewert weiter passiert. Die Continuation wird zusammen mit dem Parameter `lst`, einer Liste, an die Hilfsfunktion `sum-list-2` übergeben. Diese Funktion summiert rekursiv alle Elemente der Liste, wobei sie jeweils überprüft, ob das Listenelement eine Zahl ist. Ist das nicht der Fall, dann setzt sie, durch einen Aufruf von `return`, die zuvor eingefangene Continuation wieder ein. Der String „`Error`" wird dadurch direkt zum Ergebnis von `sum-list`, und die angesammelte Rekursion von `sum-list-2`-Aufrufen wird komplett verworfen. Das Einsetzen der Continuation stellt also hier eine Art Rücksprung dar.

Mithilfe von Continuations können in Scheme eine ganze Reihe von Kontrollstrukturen implementiert werden, für die in anderen Sprachen spezielle Sprachelemente eingeführt werden müssen. Dazu gehören beispielsweise Nicht-Lokale-Sprünge, Ausnahmen, und eben User-Level-Threads.

Aus Gründen auf die hier nicht näher eingegangen werden soll, verwendet das Thread-System allerdings nicht `call-with-current-contination`, sondern die grundlegenderen Funktionen `primitive-cwcc` und `with-continuation`. `Primitive-cwcc` fängt eine Continuation als ein spezielles Continuation-Objekt ein, das `with-continuation` wieder einsetzen kann (siehe hierzu „How to Add Threads to a Sequential Language Without Getting Tangled Up" von Gasbichler et al. [6]).

Interrupts Weiter verwendet das Thread-System das Interruptsystem der virtuellen Maschine. Ein Interrupt ist allgemein die Unterbrechung der Ausführung des Programms aufgrund eines externen Ereignisses, um eine Behandlungsfunktion für dieses Ereignis, den sogenannten *Interrupt-Handler*, auszuführen. Man spricht auch davon, daß das externe Ereignis den Interrupt *auslöst*. Nachdem der Interrupt-Handler zurückgekehrt ist, setzt die virtuelle Maschine die Ausführung des Programms an der unterbrochenen Stelle fort. Im Gegensatz zu den meisten Interrupts der Betriebsystem-Ebene, ist das Interruptsystem von Scheme-48 *synchron*, das heißt eine Unterbrechung findet niemals während der Ausführung einer Byte-Code-Instruktion statt, sondern immer zwischen zwei Instruktionen.

Scheme-48 unterstützt verschiedene Typen von Interrupts, von denen das Thread-System den *Timer-Interrupt* verwendet. Wie der Name andeutet, ist das externe Ereignis, das diesen Interrupt auslöst, das Eintreten eines bestimmten Zeitpunktes. Das Laufzeitsystem hat dabei die Möglichkeit, über das Primitivum `schedule-interrupt` eine Zeitspanne anzugeben, nach deren Ablauf die

virtuelle Maschine eine vom Laufzeitsystem definierte Behandlungsfunktion für Timer-Interrupts ausführt.

Scheduling Das Scheduling in Scheme-48 ist nach dem Prinzip der sogenannten *schachtelbaren Engines* [4, 7] organisiert. Die Analogie ist dabei, daß ein Thread mit Rechenzeit „betankt" wird, genauso wie ein Motor (englisch „Engine") mit Benzin betankt wird, um dann eine Zeit lang zu laufen. Die Engines sind außerdem schachtelbar, das heißt eine Engine kann weitere Engines enthalten, an die sie einen Teil ihres Benzins abgeben kann.

Scheme-48 folgt diesem Modell, indem die Threads in einer Baumstruktur organisiert sind. Die Knoten in diesem Baum werden als *Scheduler-Threads*, oder kurz *Scheduler* bezeichnet, die Blätter sind „gewöhnliche" Threads. An der Wurzel steht dementsprechend der sogenannte *Root-Thread*, oder *Root-Scheduler*. Jeder Thread ist durch ein Verbundobjekt vom Typ `Thread` repräsentiert, das für Scheduler-Threads zwei spezielle Felder enthält: eine Warteschlange mit sogenannten *Scheduling-Ereignissen*, sowie der momentan aktive „Kind-Thread", der sogenannte *Current-Task*. Jedes `Thread`-Objekt, bis auf den Root-Thread, enthält zudem einen Verweis auf seinen Scheduler, den „Vater-Thread".

Ein zentrales Mechanismus des Schedulings ist der *Thread-Wechsel*, das heißt der Wechsel der Ausführung von einem Thread zu einem anderen. Das `Thread`-Objekt enthält dazu ein Feld für eine Continuation. Der Wechsel vom momentan aktiven Thread zu einem anderen Thread erfolgt in den folgenden Schritten:

1. Einfangen der momentanen Continuation
2. Speichern dieser Continuation im `Thread`-Objekt des momentanen Threads
3. Auslesen der Continuation aus dem `Thread`-Objekt des neuen Threads
4. Einsetzen dieser Continuation

Ein Thread-Wechsel findet in der Behandlungsroutine eines Timer-Interrupts statt, sowie in diversen anderen Situationen. Darauf wird weiter unten noch näher eingegangen.

Zur Verteilung der Rechenzeit wählt ein Scheduler zunächst einen seiner Threads aus und vermerkt diesen in seinem `Thread`-Objekt als Current-Task. Dann setzt er die Zeitspanne bis zum nächsten Timer-Interrupt, also die Zeitspanne, die dieser Thread höchsten ausgeführt werden soll. Diese Zeitspanne wird zusätzlich im `Thread`-Objekt dieses Threads vermerkt, als die Zeitspanne, die der Scheduler diesem „Kind-Thread" von „seiner" Rechenzeit abgeben möchte. Dieser „Kind-Thread" ist aber nicht unbedingt derjenige, zu dem nun gewechselt wird. Falls dieser Thread ebenfalls ein Scheduler ist, und falls dieser bereits einen Current-Task vermerkt hat, dann findet der Thread-Wechsel direkt zu diesem Thread statt, oder entsprechend zu einem Thread noch weiter unten in der Hierarchie. Wenn dann die Zeitspanne abgelaufen ist, führt die virtuelle Maschine den Timer-Interrupt-Handler aus. In diesem Handler wird nun ein Wechsel „nach oben", das heißt zu einem über dem aktuellen Thread liegenden Scheduler durchgeführt. Dazu wird zunächst die abgelaufene Zeitspanne allen darüber liegenden Scheduler-Thread „abgebucht", das heißt die in den jeweiligen `Thread`-Objekten vermerkte Zeitspanne reduziert. Ist die Zeitspanne eines

Ereignis	Auslöser	Übliche Behandlung
`blocked`	Aktiver Thread wartet beispielsweise auf ein Mutex-Lock	Nächsten Thread auswählen
`completed`	Aktiver Thread ist fertig	Nächsten Thread auswählen
`out-of-time`	Aktiver Thread ist für die ihm zugewiesene Zeitspanne aktiv gewesen	Diesen Thread in die Runnable-Queue einfügen, und nächsten Thread auswählen
`runnable` *thread*	Ein Thread, der beispielsweise ein Mutex-Lock freigibt, meldet einen anderen, darauf wartenden Thread als lauffähig	Diesen anderen Thread in die Runnable-Queue einfügen; aktiven Thread fortsetzen
`spawned` *thread*	Der aktive Thread erzeugt einen neuen Thread. Die Funktion `spawn` löst dieses Ereignis aus.	Den neuen Thread in die Runnable-Queue einfügen; aktiven Thread fortsetzen

Tabelle 1: Scheduling-Ereignisse

der Threads in dieser Kette abgelaufen, dann findet der Wechsel zum Scheduler dieses Threads statt, damit dieser einen seiner „Kind-Threads" als neuen Current-Task auswählen kann. Ist dies nicht der Fall, dann bildet der Interrupt-Handler das Minimum aller verbleibenden Zeitspannen, setzt diese Zeitspanne für einen neuen Timer-Interrupt, und kehrt zurück, das heißt der momentan aktive Thread wird für die verbliebene Restzeit weiter ausgeführt. Auf diese Weise verteilt jeder Scheduler die ihm zur Verfügung stehende Rechenzeit an seine „Kind-Threads", wobei der Root-Scheduler sozusagen die unendliche Rechenzeit der virtuellen Maschine verteilt.

Für die Implementierung von spezialisierten Schedulern enthält das Laufzeitsystem einige Hilfsfunktionen. Eine dieser Funktionen, `run-threads`, abstrahiert über die Behandlung der *Scheduling-Ereignisse*. Diese Ereignisse spielen im Thread-System eine zentrale Rolle. In verschiedenen Situationen fügt das Thread-System Ereignisse in die Ereignis-Warteschlange eines Schedulers ein, und sobald ein Scheduler ausgeführt wird, behandelt er zunächst ausstehende Ereignisse, und veranlasst dann die Ausführung einer seiner „Kind-Threads", wie oben beschrieben. Üblicherweise verwaltet ein Scheduler dazu eine sogenannte *Runnable-Queue*, in der er alle „Kind-Threads" vorhält, die auf eine Zuteilung von Rechenzeit warten. Tabelle 1 enthält die wichtigsten Ereignisse, die Situationen, in denen diese Ereignisse ausgelöst werden, und die übliche Behandlung des Ereignisses durch den Scheduler. Die Ereignisbehandlungsroutine gibt dabei jeweils den nächsten Thread zurück, sowie die Zeitspanne, die dieser Thread entsprechend obigem Schema ausgeführt werden soll.

Eine weitere Funktion, `round-robin-event-handler`, abstrahiert über ei-

ne einfache Scheduling-Ereignisbehandlung, das sogenannte Round-Robin-Verfahren. Die wichtigsten Paramter sind die Zeitspanne, nach der jeweils ein neuer Thread ausgewählt wird, und eine Funktion `wait`, die aufgerufen wird, wenn weder ein Ereignis noch ein lauffähiger Thread vorhanden ist. Das Round-Robin-Verfahren wählt den nächsten Thread nach dem Last-In-First-Out-Prinzip aus, das heißt nachdem ein Thread für die gewünschte Zeitspanne gelaufen ist, ist er erst wieder nach allen anderen Threads dieses Schedulers an der Reihe. Für die Funktion `wait` existiert eine Standard-Implementierung, die von allen Schedulern außer dem Root-Scheduler verwendet wird. Diese markiert den Scheduler-Thread als „wartend", und *blockiert*, das heißt wechselt zu dessen Scheduler mit einem `blocked`-Ereignis. Wird für einen als „wartend" markierten Scheduler-Thread ein Ereignis ausgelöst, beispielsweise daß einer seiner Threads wieder lauffähig ist, dann löst das Thread-System automatisch auch für dessen Scheduler ein `runnable`-Ereignis aus. Der wartende Scheduler kann dann also in nächster Zeit wieder ausgeführt werden, und das Ereignis behandeln.

An einigen Stellen mussten diese Grundlagen des Schedulings, beziehungsweise des Thread-Systems, für die Multiprozessorerweiterung verändert werden. Dies rührt vor allem daher, daß Interaktionen zwischen den Threads nun echt parallel erfolgen können. Daher ist beispielsweise die Synchronisierung des Zugriffs auf die Ereignis-Warteschlangen, sowie die Synchronisierung des Wechsels eines Schedulers in den „wartend-Zustand" notwendig. Auf die Änderungen, die über diese kleinen Anpassungen hinaus gehen, geht Abschnitt 3.2 detailliert ein.

3.1.2 Synchronisierungsmittel

Scheme-48 stellt dem Programmierer eine ganze Reihe von Synchronisierungsmitteln zur Verfügung, darunter beispielsweise Locks, sogenannte Platzhalter, oder der Nachrichtenaustausch über synchrone Kanäle nach dem Modell von Concurrent-ML [16]. Die Basis für alle Implementierungen sind die sogenannten *Proposals* [10]. Proposals sind eine Realisierung *optimistischer Nebenläufigkeit* [13] und wurden von Jonathan Rees und anderen konzipiert, und in Scheme-48 implementiert.

Die Idee der Proposals ist an das Konzept der wiederaufsetzbaren Transaktionen in Datenbank-Systemen angelehnt. Ein Proposal wird Stück für Stück *erstellt*, um dann mit dem sogenannten *Commit* ausgeführt zu werden. Vereinfacht ist ein Proposal eine Liste, deren Einträge jeweils ein Feld einer Datenstruktur bestimmen, sowie zwei Werte enthalten. Die Bedeutung der zwei Werte ist, daß der eine Wert in die Datenstruktur geschrieben werden soll, falls der an dieser Stelle vorhandene Wert mit dem zweiten Wert im Proposal-Eintrag übereinstimmt. Diese Voraussetzung gilt dabei für alle Einträge zusammen, das heißt keine Schreiboperation soll durchgeführt werden, falls sich auch nur ein in der Datenstruktur vorhandener Wert von dem Erwartungswert im Proposal-Eintrag unterscheidet. Das Ziel eines Commits ist es also, die im Proposal gespeicherten Schreiboperationen durchzuführen, unter der Voraussetzung, daß für alle Einträge die jeweilige Datenstruktur den jeweils gespeicherten Wert enthält. Ein Commit kann daher fehlschlagen, falls diese Voraussetzung nicht erfüllt ist, und das Commit muß eine *atomare* Operation sein, das heißt zu jedem Zeitpunkt darf maximal ein Commit ausgeführt werden.

Das Füllen eines Proposals mit solchen Einträgen geschieht über speziel-

le Funktionen, die entweder eine Lese- oder eine Schreiboperation darstellen[13]. Für alle primitiven Datentypen sind solche Funktionen vordefiniert, und haben gegenüber den normalen Selektoren oder Mutatoren das Prefix „`provisional-`". Das Commit wird über die Funktion `maybe-commit` durchgeführt, deren Rückgabewert angibt, ob das Commit durchgeführt wurde, oder fehlgeschlagen ist. Für jeden Thread kann dazu dynamisch ein aktuelles Proposal installiert werden, welches die „`provisional`-Funktionen" dann jeweils verwenden. Das heißt, das zu verwendende Proposal muß nicht als Parameter übergeben werden, sondern ist über den aktuellen Thread bestimmt. Ein neues Proposal kann mit dem Makro `with-new-proposal` für den aktuellen Thread installiert werden. Wie diese Funktionen nun für die Synchronisierung des Zugriffs auf Datenstrukturen verwendet werden kann, soll am Beispiel eines Zählers dargestellt werden:

```
(define (make-counter) (make-cell 0))

(define (get-counter-value-and-increment counter)
  (with-new-proposal (retry)
   (let ((v (provisional-cell-ref counter)))
     (provisional-cell-set! counter (+ v 1))
     (if (maybe-commit)
         v
         (retry)))))

(get-counter-value-and-increment (make-counter))
```

`Make-counter` erzeugt einen neuen Zähler mit dem anfänglichen Wert Null. Der Zählerstand wird dazu in einer Zelle (engl. `Cell`) gespeichert. `Get-counter-value-and-increment` soll nun den Zähler um Eins erhöhen und den vorherigen Wert zurückgeben. Dazu installiert die Funktion ein neues Proposal mit dem Makro `with-new-proposal`, welches außerdem eine Funktion ohne Parameter an den Namen `retry` bindet (hierauf wird weiter unten eingegangen). Weiter liest `get-counter-value-and-increment` den Wert des Zählers aus und schreibt den um eins erhöhten Wert zurück – beides über die Provisional-Funktionen für den `Cell`-Datentyp. Der gelesene Wert, sowie die Schreibanweisung wird also im Proposal vermerkt, und der tatsächliche Wert des Zählers bleibt zunächst unverändert. Erst durch den folgenden Aufruf von `maybe-commit` soll nun der erhöhte Wert in das `Cell`-Objekt geschrieben werden, falls sein Wert noch der selbe ist wie zuvor, das heißt falls der Zähler nicht zwischenzeitlich durch einen anderen Thread erhöht wurde. Falls dies nicht passiert ist, führt `maybe-commit` die Schreiboperation durch und gibt True zurück. `Get-counter-value-and-increment` kann dann also den vorherigen Wert zurückgeben. Hat sich allerdings der Wert des Zählers verändert, muß die *gesamte* Prozedur wiederholt werden. Die von `with-new-proposal` an den Namen `retry` gebundene Funktion ist nun genau eine solche Funktion, die das Auslesen des Zählers, die Addition, das Zurückschreiben und `maybe-commit` erneut ausführt. Der Prozess wiederholt sich also, bis das Proposal erfolgreich ausgeführt wurde.

Diese „Versuch-und-Irrtum"-Strategie ist auch der Grund für die Bezeichnung als *optimistische* Nebenläufigkeit. Die Annahme ist dabei, daß sich in den

[13]Wird das selbe Feld einer Datenstruktur mehrmals ausgelesen, mehrmals beschrieben, oder ausgelesen nachdem es beschrieben wurde, werden auch vorherige Einträge im Proposal verändert.

meisten Fällen die Threads nicht „in die Quere" kommen. Man riskiert eine mehrmalige Verarbeitung der Datenstrukturen, mit dem Vorteil die Länge des kritischen Abschnitts zu minimieren. Dieser kritische Abschnitt, der vor einer gleichzeitigen Ausführung durch mehrere Threads geschützt werden muß, besteht nämlich allein aus dem Commit des Proposals. Die Zusammenstellung eines Proposals, und damit auch eventuelle längere Rechenoperationen, können parallel in mehreren Threads erfolgen.

Wie oben erwähnt, muß die Ausführung der Funktion `maybe-commit` *atomar* sein. Atomar bedeutet, daß niemals zwei Aufrufe (scheinbar) gleichzeitig ausgeführt werden. Beim Multitasking mit nur einem virtuellen Prozessor hätte die Atomizität durch eine vorübergehende Deaktivierung der Timer-Interrupts erreicht werden können. Aus Effizienzgründen ist `maybe-commit` aber als Primitivum in der virtuellen Maschine implementiert. Und da die Interruptbehandlung in Scheme-48 immer nur zwischen zwei Instruktionen der virtuellen Maschine stattfindet, läuft jede Instruktion automatisch atomar ab. In der Multiprozessorsituation gilt dies aber nur noch innerhalb eines virtuellen Prozessors, und nicht mehr für die gesamte virtuelle Maschine. Jeder virtuelle Prozessor kann gleichzeitig mit anderen Prozessoren jeweils eine Instruktion ausführen. Die Atomizität von `maybe-commit` muß also durch eine *Synchronisation* der Aufrufe, beziehungsweise der virtuellen Prozessoren, hergestellt werden. Dazu dient ein spezielles Mutex-Lock innerhalb der virtuellen Maschine. Dies verhindert die gleichzeitige Ausführung mehrerer `maybe-commit`-Aufrufe.

Bei falscher Verwendung der Proposals, kann es nun aber zu Fehlern kommen, die mit nur einem virtuellen Prozessor nicht möglich waren. Diese Fehler treten auf, wenn der Benutzer eine Datenstruktur gleichzeitig über die „provisional-Mutatoren" und ihre normalen Gegenstücke verändert. Im Gegensatz zur Uniprozessorsituation, kann diese Mutation nämlich nun zwischen der Überprüfung des Proposals und der tatsächlichen Veränderung der Datenstrukturen innerhalb von `maybe-commit` auftreten. Die betreffenden Daten können dann einen inkonsistenten Zustand annehmen. Der Benutzer ist dafür verantwortlich solche „gemischten" Zugriffe zu verhindern. Diese gemischten Zugriffe können aber auch in der Uniprozessorsituation zu Problemen führen, daher bedeutet dies keine Einschränkung der Funktionalität der Proposals.

3.2 Multitasking auf allen Prozessoren

Ein Ziel dieser Arbeit ist, die potentiell große Zahl von User-Threads automatisch auf mehrere virtuellen Prozessoren zu verteilen. Dazu ist es zunächst notwendig, daß auf jedem virtuellen Prozessor Multitasking möglich ist. Das bedeutet, für jeden virtuellen Prozessor existiert ein eigener Root-Scheduler. Die Implementierung des Root-Schedulers, beziehungsweise der Teile des Thread-Systems, bei denen der Root-Scheduler beteiligt ist, mussten an diese Situation angepasst werden. Der Root-Scheduler

1. weckt schlafende Threads auf,
2. leert gepufferte Ausgabekanäle,
3. wartet auf externe Ereignisse, wenn keine lauffähigen Threads und keine Ereignisse vorliegen, und

4. versucht Deadlock-Situationen zu erkennen.

Die ersten zwei Operationen führt der Root-Scheduler sowohl in regelmäßigen Abständen durch (das sogenannte *Housekeeping*), als auch in dem Fall, daß seine Warteschlange der lauffähigen Threads leer ist, und keine zu behandelnden Scheduling-Ereignisse vorhanden sind. Von letzterem Fall aus gesehen, ergeben sich durch das Housekeeping jedoch keine zusätzlichen Schwierigkeiten, weshalb im folgenden die Veränderungen für die Multiprozessorerweiterung anhand der Situation der leeren Thread- und Ereignisschlange des Root-Schedulers beschrieben werden.

Wenn der Root-Scheduler eines virtuellen Prozessors keine Threads in seiner Warteschlange hat, und er keine Ereignisse zu bearbeiten hat, dann ruft er die Funktion `root-wait` auf. In der Uniprozessorsituation geht diese Funktion in vier Schritten vor. Zunächst startet sie für jeden Ausgabekanal, der Daten gepuffert hat, einen Thread, der die Ausgabe des Pufferinhalts durchführt. Dann weckt `root-wait` alle *schlafenden Threads* auf, die geweckt werden möchten. Schlafende Threads sind Threads, die für eine gewisse Zeitspanne nicht ausgeführt werden möchten. Ist diese Zeitspanne abgelaufen, dann kann der Thread wieder wie ein normaler Thread behandelt werden, und man spricht vom Aufwecken des Threads. Diese ersten beiden Schritte dienen gewissermaßen dazu, dem Root-Scheduler „neue Arbeit" zu verschaffen. Gibt es jedoch keine gepufferten Daten, und keine aufzuweckenden Threads, dann führt `root-wait` den dritten Schritt durch. Dieser besteht darin, zu prüfen, ob *in Zukunft* noch Arbeit für den Root-Scheduler zu erwarten ist. Dies ist der Fall, wenn schlafende Threads existieren, deren Zeitspanne noch nicht abgelaufen ist, oder wenn mindestens ein Thread auf Daten aus einem Eingabekanal, oder auf den Erfolg einer Ausgabe wartet. `Root-wait` ruft dann ein Primitivum der virtuellen Maschine auf, das für eine bestimmte maximale Zeitspanne auf Ein-/Ausgabeereignisse wartet[14]. Die maximale Zeitspanne wählt `root-wait` so, daß sie den nächsten schlafenden Thread pünklich wecken kann. Gibt es keinen schlafenden Thread, dann ruft `root-wait` das Primtivum so auf, daß es potentiell unendlich lange auf Ein-/Ausgabeereignisse wartet. Den vierten Schritt führt `root-wait` aus, wenn weder schlafende Threads existieren, noch Threads auf Daten aus Eingabekanälen warten. Dieser Fall stellt einen sogenannten *Deadlock* dar, bei dem das Laufzeitsystem sicher ist, daß keiner der existierenden Threads in Zukunft lauffähig werden kann. `Root-wait` gibt in dieser Situation `#f` zurück, was zu einer Beendigung des Root-Schedulers, und damit des ganzen Multitaskings führt[15]. In der Regel stellen Deadlocks einen Programmierfehler des Anwenders dar.

In der Multiprozessorsituation existieren nun mehrere Root-Scheduler, deren Threads beliebig interagieren können. Daraus ergeben sich verschiedene Schwierigkeiten für die korrekte Implementierung der hier beschriebenen Aufgaben der Root-Scheduler. Der folgende Abschnitt geht auf die schlafenden Threads ein, Abschnitt 3.2.2 beschreibt die weiteren Veränderungen an `root-wait` und die Problematik der Deadlock-Erkennung.

[14] Auf Unix-Systemen beinhaltet dies einen Aufruf der bekannten Funktion `select()`.

[15] Der Root-Scheduler ermöglicht dem Kommando-Prozessor einen Deadlock-Handler zu definieren, der dann an dieser Stelle aufgerufen wird, anstatt den Root-Scheduler zu beenden. Dieser Deadlock-Handler meldet dem Benutzer die Erkennung des Deadlocks, und ermöglicht ihm entweder einen Fehler zu bereinigen, oder neue Befehle einzugeben.

3.2.1 Schlafende Threads

Die meisten Thread-Systeme geben dem Benutzer die Möglichkeit einen Thread für eine gewisse Zeitspanne schlafen zu legen, das heißt der Thread verbraucht während dieser Zeit keine Rechenzeit. Das Thread-System von Scheme-48 bietet dazu die Funktion `sleep` an:

`(sleep` *milli-seconds*`)` $\Rightarrow$ *unspecific*

Die Funktion kehrt zurück, nachdem die angegebene Zahl von Millisekunden abgelaufen ist. In Scheme-48 ist dies über eine sortierte Liste aller schlafenden Threads implementiert, die in einer globalen Variablen abgelegt wird. Die Sortierung erfolgt dabei über den jeweiligen Zeitpunkt, an dem der Thread aufgeweckt werden möchte, sodaß der nächste aufzuweckende Thread immer an erster Stelle in der Liste steht. Die Funktion muß also den Thread und den entsprechenden Aufweck-Zeitpunkt in die Liste einsortieren, und eine Ereignis an den Scheduler des Threads schicken, das angibt, daß der aktuelle Thread blockiert ist, und der Scheduler einen anderen Thread aktivieren kann. Der Root-Scheduler durchsucht regelmäßig diese Liste und weckt alle Threads auf, deren Aufweck-Zeitpunkt erreicht ist. Dies beinhaltet entsprechend jeweils das Entfernen des Threads aus der Liste, und ein Ereignis an den Scheduler des Threads, das angibt, daß der Thread wieder in die Liste der lauffähigen Threads eingefügt werden kann.

Für die Multiprozessorerweiterung muß dieses System modifiziert werden. Entweder

1. jeder Root-Scheduler verwaltet eine eigene Liste schlafender Threads, oder
2. das Einfügen in die Liste und das Senden des Blockiert-Ereignisses, beziehungsweise das Entfernen und das Senden des Lauffähig-Ereignisses, müssen zu einer atomaren Operation zusammengefasst werden.

Beide Möglichkeiten verhindern, daß das Blockiert-Ereignis gesendet wird, nachdem bereits das Lauffähig-Ereignis gesendet wurde. Das kann insbesondere bei kurzen Zeitspannen passieren, wenn auf einem anderen virtuellen Prozessor der Thread bereits wieder aus der Liste entfernt wird, bevor das Blockiert-Ereignis gesendet wurde[16].

In dieser Arbeit wurde die zweite Möglichkeit implementiert, da dadurch auch nur einer der Root-Scheduler regelmäßig die Liste der schlafenden Threads durchsuchen muß.

3.2.2 `Root-wait`

Es sind Situationen möglich, in denen ein Root-Scheduler nichts zu tun hat, das heißt es gibt keinen lauffähigen Thread oder zu behandelndes Ereignis. Dies ist der Fall, wenn

1. alle Threads entweder schlafen, das heißt auf den Ablauf einer bestimmten Zeitspanne warten, oder auf Ein-/Ausgabeergebnisse warten, oder
2. ein Deadlock eingetreten ist, das heißt kein Thread mehr vorhanden ist, der jetzt oder in Zukunft lauffähig sein wird.

[16] Zuerst das Blockert-Ereignis zu senden, und dann erst den Thread in die Liste einzufügen, ist aus systematischen Gründen nicht möglich.

In der bisherigen Implementierung stellt der Root-Scheduler im ersten Fall zunächst fest, wie groß die Zeitspanne ist, bis der nächste schlafende Thread aufweckt werden möchte, und ruft dann ein Primitivum names `wait` auf, welches zurückkehrt sobald entweder ein Ein-/Ausgabeereignis eingetreten ist, oder diese Zeitspanne abgelaufen ist. Dies geschieht auf Unix-Systemen mittels des Betriebssystemaufrufs `select()`, was bedeutet, daß der Scheme-48-Prozess nicht aktiv wartet, also keine Rechenzeit des Betriebssystems verschwendet. Im zweiten Fall, dem Deadlock, kehrt der Root-Scheduler zurück, was die Beendigung des Interpreter mit einer entsprechenden Fehlermeldung zur Folge hat.

In der Multiprozessorsituation kommt nun noch eine weitere Möglichkeit hinzu, die die korrekte und effiziente Implementierung erheblich erschwert: Auf einem anderen Prozessor, beziehungsweise Root-Scheduler, existiert noch ein lauffähiger Thread, der einen Thread dieses Root-Schedulers zu jedem Zeitpunkt wieder lauffähig machen könnte. Das Thread-System generiert in solchen Situationen ein neues Ereignis für den Scheduler des wartenden Threads, und reiht es in die Ereignis-Schlange des Schedulers ein. Ist dieser Scheduler zu diesem Zeitpunkt nicht lauffähig, das heißt im Warte-Zustand, dann generiert das Thread-System für dessen Scheduler ebenfalls ein entsprechendes Ereignis. In der Uniprozessor-Situation war es dabei nicht möglich, daß der Root-Scheduler in dieser Kette der Scheduler ebenfalls in einem inaktiven Zustand ist. Das ist nun nicht mehr der Fall.

Der Lösungsansatz für die Multiprozessorerweiterung besteht daher erstens darin, daß ein Root-Scheduler, der das Primitivum `wait` aufruft, sich in seinem Thread-Objekt, ebenso wie alle gewöhnlichen Scheduler, als „wartend" markiert. Zweitens muß das `wait`-Primitivum nun zusätzlich auch dann zurückkehren, wenn ein neues Ereignis für den Root-Scheduler vorliegt. Dazu stellt die virtuelle Maschine ein neues Primitivum namens `return-from-wait` zur Verfügung, das das Thread-System aufruft, wenn für einen wartenden Root-Scheduler ein Ereignis eintritt, er also in einem `wait`-Aufruf blockiert ist. Ferner kann eine Deadlock-Situation in der Multiprozessorsituation erst angenommen werden, wenn alle anderen virtuellen Prozessoren im Warte-Zustand sind.

Für die Implementierung von `wait` und `return-from-wait` bieten sich zwei Möglichkeiten an:

Lösung 1 Der POSIX-Standard beinhaltet eine Variante der `select`-Funktion namens `pselect`. Diese hat gegenüber `select` als weiteren Parameter eine Signal-Maske. `Pselect` setzt in einem atomaren Schritt diese Signal-Maske für den aufrufenden Thread ein, ruft `select` auf, und installiert nach der Rückkehr von `select` die vorherige Signal-Maske wieder. Es ist damit also möglich, ein bestimmtes asynchrones Betriebsystem-Signal dazu zu verwenden, den `select`-Aufruf in `wait` von einem anderen virtuellen Prozessor, beziehungsweise Kernel-Level-Thread, aus zu unterbrechen. Die normale Signal-Maske muß dazu dieses Signal als blockiert markieren, die einzusetzende Maske als deblockiert. Das `return-from-wait`-Primitivum muß dann lediglich dieses Signal an den betreffenden Kernel-Level-Thread senden, da `select` nach der Behandlung eines Betriebsystem-Signals automatisch zurückkehrt. Die Atomizität dieser Schritte in `pselect`, durch die kein Signal verloren geht, ist sehr wichtig, da nicht verhindert werden kann, daß zwischen der Markierung des Root-Schedulers als „wartend", und dem `select`-Aufruf in `wait`, bereits ein anderer Root-Scheduler

`return-from-wait` aufrufen könnte.

Leider war zum Zeitpunkt dieser Arbeit die `pselect`-Funktion in den verwendeten Betriebsystemen entweder gar nicht (Solaris), oder nicht atomar implementiert (Linux). Dieser Lösungsansatz konnte daher nicht verwendet werden, obwohl er gegenüber der zweiten Lösung Vorteile hat.

Lösung 2 Die Lösung, die in dieser Arbeit implementiert wurde, verwendet einen zusätzlichen Kernel-Level-Thread und eine sogenannte Condition-Variable. Condition-Variablen sind ein übliches Synchronisierungs-Mittel in der nebenläufigen Programmierung. Ein Thread kann auf eine *Bedingung* warten; Ein anderer Thread kann sie signalisieren (siehe dazu auch Abschnitt 1.2). Der POSIX-Standard beinhaltet außerdem eine Funktion, mit der es möglich ist, *zeitlich begrenzt* auf das Eintreten der Bedingung zu warten. Die Aufgabe des zusätzlichen Kernel-Level-Threads ist es dabei, in einer Endlosschleife `select` aufzurufen, und die globale Condition-Variable `event-occured` zu signalisieren, wenn ein I/O-Ereignis eingetreten ist. Der `select`-Aufruf wird dabei mit einer kurzen maximalen Wartezeit aufgerufen, sodaß regelmäßig neue I/O-Kanäle in die Überwachung miteinbezogen werden können. Das `wait`-Primitivum besteht nun einfach aus einem zeitlich begrenzten Warten auf das Eintreten dieser globalen Bedingung. `Return-from-wait` signalisiert diese Bedingung direkt.

Diese Lösung hat gegenüber der `pselect`-Implementierung zwei Nachteile. Erstens verringert sie die Reaktivität des I/O-System, da neue Kanäle nicht sofort überwacht werden können. Zweitens führt ein Aufruf von `return-from-wait` dazu, daß alle wartenden Root-Scheduler zurückkehren, obwohl nur für einen bestimmten von ihnen ein zu behandelndes Ereignis vorliegt. Beide Nachteile sind aber nicht sehr schwerwiegend, weil einerseits Zugriffe auf Dateien oder Netzwerk-Kanäle ohnehin immer einige Zeit benötigen, und andererseits bei gut ausgelasteten Root-Schedulern kaum `wait` beziehungsweise `return-from-wait`-Aufrufe notwendig sind. Ein weiterer geringfügiger Nachteil ist, daß der Scheme-48-Prozess nicht mehr vollständig inaktiv werden kann, da immer in der Endlosschleife im zusätzlichen Kernel-Level-Thread immer wieder die Liste der I/O-Kanäle aktualisiert wird.

3.3 Der parallele Scheduler

Die bisher beschriebenen Änderungen ermöglichen es dem Laufzeitsystem mehrere virtuelle Prozessoren zu starten, und auf jedem einzelnen Prozessor eine Funktion, oder viele User-Level-Threads durch Multitasking, auszuführen. Für den Benutzer von Scheme-48 ist diese Funktionalität jedoch noch nicht sehr praktisch, da er selbst die Verteilung der Anwendung auf die virtuellen Prozessoren vornehmen müsste. Daher ist es sinnvoll, die eventuell große Zahl von User-Level-Threads automatisch auf die zur Verfügung stehenden virtuellen Prozessoren zu verteilen. Ein solcher Automatismus erhöht auch die Kompatibilität mit der Uniprozessorversion von Scheme-48, sowie die Skalierbarkeit der Anwendung auf Maschinen mit einer größeren Anzahl von Prozessoren. Zudem lässt sich die Auslastung der virtuellen Prozessoren sicherstellen.

Im Rahmen dieser Arbeit wurde dieser Automatismus in einer einfachen Version implementiert: die sogenannten *parallelen Scheduler*. Das Prinzip der parallelen Scheduler ist:

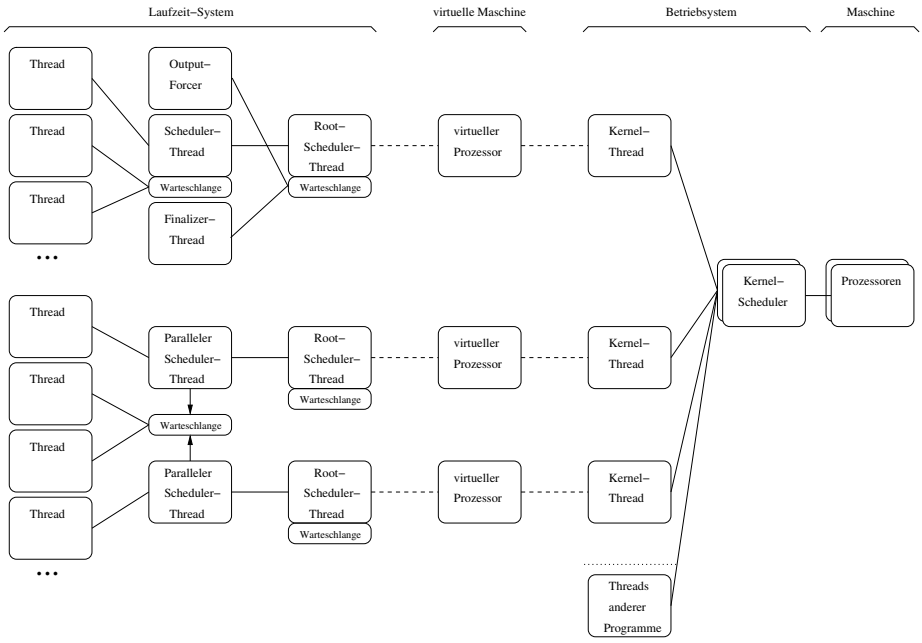

Abbildung 7: Überblick Scheme-Threads und -Scheduler, Kernel-Threads und -Scheduler, virtuelle und reale Prozessoren

- Auf dem Root-Scheduler eines jeden virtuellen Prozessors läuft jeweils ein weiterer Scheduler, ein paralleler Scheduler.
- Diese parallelen Scheduler operieren auf einer *gemeinsamen Liste* von lauffähigen Threads.
- Findet ein paralleler Scheduler keinen lauffähigen Thread in der Liste, dann legt er sich schlafen.
- Legt ein paralleler Scheduler einen lauffähigen Thread in die Liste ab, dann weckt er alle schlafenden parallelen Scheduler auf.

Die parallelen Scheduler verfügen also über eine gemeinsame Liste von lauffähigen Threads, kennen sich gegenseitig, und bilden damit eine Gruppe von „Partner-Schedulern", die gemeinsam die Ausführung ihrer Threads steuern. Abbildung 7 gibt einen Überblick über alle an der Ausführung der Threads beteiligten Komponenten aus dem Laufzeitsystem, der virtuellen Maschine, auf der Ebene des Betriebsystems bis hin zu den realen Prozessoren. Im oberen Teil der Abbildung ist die Situation ohne parallelen Scheduler dargestellt, im unteren Teil mit zwei parallelen Scheduler, die auf einer gemeinsamen Warteschlange von lauffähigen Threads operieren. Die gestrichelten Linien sollen andeuten, daß hier kein Scheduling involviert ist, sondern zu jedem Root-Scheduler genau ein virtueller Prozessor und Kernel-Thread gehört.

Die Auswahl der Threads und die Zuteilung der Rechenzeit erfolgt bei den parallelen Schedulern nach dem Round-Robin-Verfahren, das heißt die Scheduler greifen auf die Liste nach dem Last-In-First-Out-Prinzip zu, und es findet keine Priorisierung der Threads statt. Findet einer der Scheduler keinen lauffähigen

Thread, dann markiert es sich als „wartend" und blockiert, genauso wie es für andere Scheduler in Scheme-48 implementiert ist. In jedem `Thread`-Objekt kann jedoch nur ein einziger Scheduler eingetragen werden, sodaß sich der Scheduler nicht darauf verlassen kann vom Thread-System reaktiviert zu werden, sobald ein Thread wieder lauffähig wird. *Einer* der kooperierenden Scheduler erhält jedoch in jedem Fall das entsprechende Ereignis, sodaß er an dieser Stelle seine „Partner-Scheduler" reaktivieren kann, und diese gegebenenfalls die Ausführung des Threads übernehmen können. Entsprechendes gilt für die Erzeugung eines neuen Threads, bei der der Scheduler seine wartenden „Partner-Scheduler" aufwecken muß.

Die Funktion `spawn-parallel-schedulers` dient dazu eine Gruppe von kooperierenden parallelen Schedulern zu starten:

`(spawn-parallel-schedulers schedulers)` → `parallel-schedulers`

Die Funktion nimmt eine Liste von Schedulern, erzeugt auf jedem Scheduler einen neuen Thread, und startet in jedem dieser Threads einen parallelen Scheduler. Jeder der parallelen Scheduler erhält dabei eine Referenz auf dieselbe Liste für lauffähige Threads, sowie eine Liste der anderen parallelen Scheduler. Die Rückgabe von `spawn-parallel-schedulers` ist eine Liste der erzeugten Scheduler-Threads.

Für den üblichen Fall, daß die parallelen Schedulern auf den Root-Schedulern von neuen virtuellen Prozessoren gestartet werden sollen, existiert ferner die Funktion `start-multiprocessing`:

```
(start-multiprocessing number-of-processors
                       (lambda () calculations ...))
```

Die Parameter sind zum einen die Anzahl der zu startenden virtuellen Prozessoren, und zum anderen eine Funktion ohne Parameter. Die Funktion startet neue virtuelle Prozessoren, eine Gruppe von parallelen Schedulern auf deren Root-Schedulern, und auf einem der parallelen Schedulern einen ersten Thread, der die hier übergebene Funktion ausführt. Startet der Benutzer in dieser Funktion also neue Threads, so übernehmen die parallelen Scheduler, und damit verschiedene virtuelle Prozessoren, die Ausführung dieser Threads gemeinsam. `Start-multiprocessing` gibt das Ergebnis der übergebenen Funktion zurück. Daher werden auch nach der Rückkehr der Funktion die neu erzeugten virtuellen Prozessoren wieder beendet.

Ein bestehendes nebenläufiges Programm kann also durch einen Aufruf von `start-multiprocessing` sehr leicht eine Multiprozessormaschine ausnutzen. Bis auf diesen Funktionsaufruf, sind am Programm selbst keinerlei Änderungen notwendig. Um diese Unterstützung noch vollständig zu automatisieren, wäre eine Integration der Multiprozessorunterstützung in den Kommando-Prozessor von Scheme-48 notwendig, die in dieser Arbeit nicht vorgenommen wurde. Der Kommando-Prozessor stellt einen separaten Scheduler dar, der dem Benutzer beispielsweise die vorübergehende Unterbrechung der Threads erlaubt. Die im Kommano-Prozessor enthaltenen interaktiven Möglichkeiten zum Debugging und zur Ausnahmebehandlung sollten ebenfalls die gestarteten virtuellen Prozessoren berücksichtigen. Eine Integration ist daher mit einem nicht unerheblichen Aufwand verbunden.

Die hier vorgestellte Realisierung paralleler Scheduler zeigt, daß das modifizierte Thread-System für eine *Migration* der Threads gerüstet ist, das heißt daß der Wechsel eines Threads zu einem anderen virtuellen Prozessor möglich ist. Es ist jedoch anzunehmen, daß Migrationen die Effizienz der Ausführung verringern. Ein verbessertes Scheduling-Verfahren, das Migrationen vermeidet, aber dennoch eine Auslastung der virtuellen Prozessoren gewährleistet, könnte daher noch effizienter sein, als die hier vorgestellten parallelen Scheduler.

4 Analyse

Dieser Abschnitt präsentiert eine Analyse des entstandenen multiprozessorfähigen Systems. Dies beinhaltet die Gesamt-Performance einiger Benchmark-Programme, einen Vergleich mit dem unveränderten Scheme-48-System, sowie eine Analyse der Creation-Spaces aus der Speicherverwaltung, die für die Effizienz des Gesamtsystems sehr wichtig ist.

4.1 Gesamt-Performance

Ein Ziel dieser Arbeit war es, mehrere Prozessoren einer Maschine so auszunutzen, daß Scheme-48 ein nebenläufig geschriebenes Scheme-Programm schneller interpretiert, als mit nur einem Prozessor. Bei dieser Geschwindigkeit geht es um die sogenannte *Realzeit*, das heißt die Zeit vom Beginn der Ausführung des Programms bis zu seinem Ende, denn das ist die Zeit, auf die es dem Anwender letztenendes ankommt. Neben der Realzeit wird als weitere Größe die sogenannte *Laufzeit* betrachtet. Die Laufzeit ist die Summe der Zeiten, in denen ein Teil des Interpreters von einem der Prozessoren ausgeführt wurde. Teilt man die Laufzeit durch die Anzahl der verwendeten virtuellen Prozessoren, und vergleicht sie mit der Realzeit, dann erhält man ein Maß für den Ausnutzungsgrad der Prozessoren durch den Interpreter. Ein schlechter Ausnutzungsgrad kann zum einen an anderen Programmen liegen, die gleichzeitig auf der Maschine ausgeführt werden, aber auch an der Implementierung von Scheme-48, und dem ausgeführten Programm selbst.

Im folgenden werden zunächst die verwendeten Benchmarking-Programme beschrieben, dann die Rechner auf denen die Messungen durchgeführt wurden und schließlich die Ergebnisse.

4.1.1 Benchmark-Programme

Nebenläufige Programme unterscheiden sich sowohl in der Anzahl der Threads, als auch im Grad der Unabhängigkeit der Threads. Threads sind vor allem dann voneinander abhängig, wenn sie auf gemeinsamen Daten operieren, und der Zugriff auf diese Daten synchronisiert werden muß. Dementsprechend wurden die Benchmark-Programme so ausgewählt, daß sie die Bandbreite von wenigen, unabhängigen Threads, bis zu vielen, sehr abhängigen Threads abdecken. Im folgenden sind die verwendeten Programme kurz beschrieben:

`Matrix-mul` Dieses Programm multipliziert zwei 135x135 Matrizen, indem es die Ergebnismatrix in 8 Teile aufteilt, und jeden dieser Teile in einem eigenen Thread berechnet. Die Eigenschaften sind also:

- wenig Threads,
- kaum Synchronisation notwendig, da auf gemeinsame Daten nur lesend zugegriffen wird.

`Matrix-mul2` Dieses Benchmark berechnet in 8 Threads jeweils das Produkt zweier eigener 60x60 Matrizen. Die Eigenschaften sind:

- wenig Threads,

- keine Synchronisation, da keine gemeinsamen Daten.

`Stress` Bei diesem Benchmark operieren 100 Threads auf einem gemeinsamen Vektor der Länge 800. Jeder Thread tauscht 500mal die Werte an zwei zufällig gewählten Positionen des Vektor aus, und synchronisiert die Lese- und Schreibvorgänge über Proposals. Dieses Benchmark hat die Eigenschaften:

- viele Threads,
- viel Synchronisation, da auf gemeinsame Daten lesend und schreibend zugegriffen wird.

4.1.2 Rechner

Zum Test der Multiprozessorerweiterung standen zwei Rechner am Wilhelm-Schickard-Institut zur Verfügung. Die folgende Tabelle zeigt die wichtigsten Eckdaten der beiden Rechner:

	Rechner `bluff`	Rechner `galibier`
Anzahl Prozessoren	8	2
Taktrate	900MHz	1000MHz
Architektur	SparcV9	Intel Pentium III
Betriebsystem	SunOS 5.9 (Solaris 9)	FreeBSD 6.0-RC1

4.1.3 Ergebnisse

Die Abbildungen 8 und 9 zeigen jeweils die Ergebnisse für die Rechner `galibier` beziehungsweise `bluff`. Die Ergebnisse der einzelnen Benchmarks sind dabei zu einer Gruppe zusammengefasst, innerhalb derer jeweils von links nach rechts von einem bis zwei, beziehungsweise bis acht virtuelle Prozessoren verwendet wurden („sp", „mp2" bis „mp8"). Leicht versetzt ist jeweils die Realzeit und die durch die Anzahl der virtuellen Prozessoren dividierte Laufzeitmessung dargestellt. Um stabilere Ergebnisse zu erhalten, wurden die einzelnen Benchmarks jeweils 10mal wiederholt, die angegebenen Zahlenwerte sind die Durchschnittswerte dieser Wiederholungen.

Die Abbildungen zeigen, daß die Ausführungszeiten mit zunehmender Anzahl Prozessoren deutlich abnehmen. Sie bewegen sich in der Nähe des Optimums, das jeweils der halben Höhe des vorherigen Balkens entspricht. Die Verbesserungen sind bei dem Benchmark `stress` jedoch merklich geringer als bei den ersten zwei Benchmarks. Der Ausnutzungsgrad der Prozessoren, der in den Unterschieden der leicht versetzt dargestellen Realzeiten und Laufzeiten deutlich wird, verringert sich mit zunehmender Anzahl Prozessoren deutlich. Die stärke des Unterschieds ist aber auch sehr von dem jeweiligen Benchmark abhängig. Beim Benchmark `matrix-mul2` ist der Unterschied kaum ausgeprägt, der Ausnutzungsgrad also sehr hoch, beim Benchmark `matrix-mul` aber sehr viel stärker. Im Gegensatz zu den anderen beiden Benchmarks, greifen die Threads in `matrix-mul2` kaum auf gemeinsame Datenstrukturen zu, was die unterschiedlichen Verbesserungen und Ausnutzungsgrade erklären könnte.

Die Zahlenwerte der Realzeit-Messungen auf dem Rechner `bluff`, beziehungsweise `galibier`, sind nocheinmal in den Tabellen 2 und 3 enthalten. Die

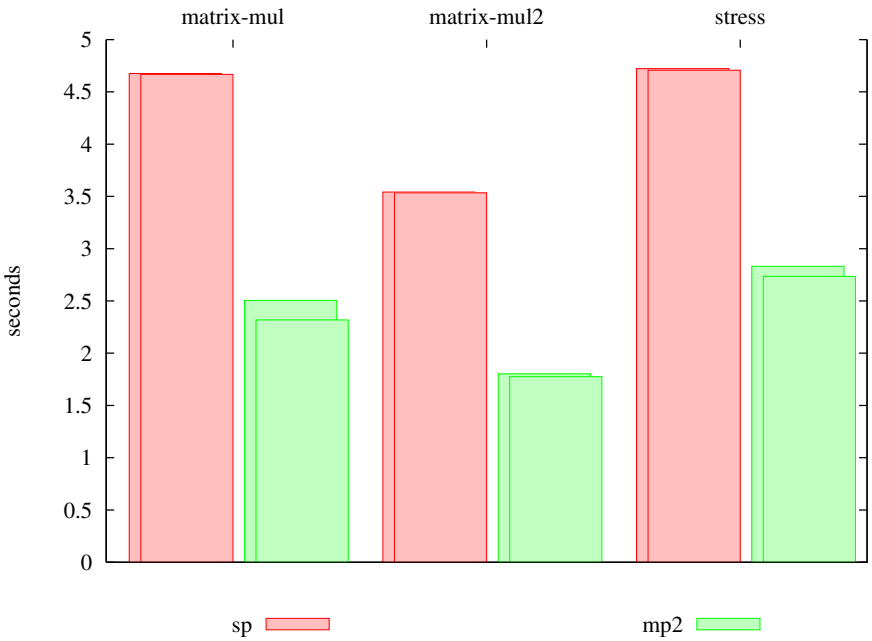

Abbildung 8: Benchmark-Ergebnisse auf `galibier`

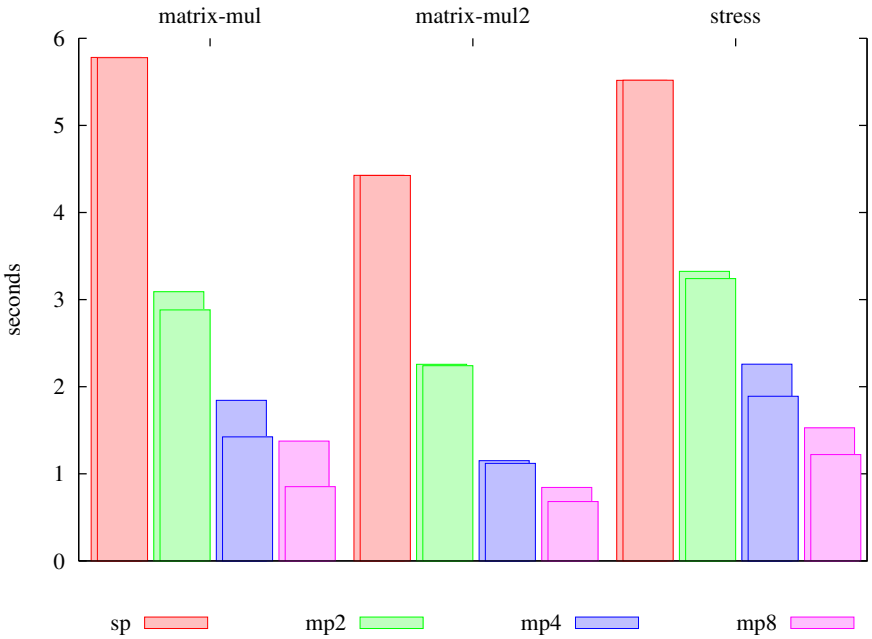

Abbildung 9: Benchmark-Ergebnisse auf `bluff`

	matrix-mul	matrix-mul2	stress
1 virtueller Prozessor	5,781s	4,4264s	5,5177s
2 virtuelle Prozessoren	3,0901s	2,2579s	3,3241s
Speedup 1 auf 2 VPs	1,87	1,96	1,66
4 virtuelle Prozessoren	1,8433s	1,1499s	2,2583s
Speedup 2 auf 4 VPs	1,68	1,96	1,47
8 virtuelle Prozessoren	1,3756s	0,843s	1,5285s
Speedup 4 auf 8 VPs	1,34	1,36	1,47

Tabelle 2: Realzeiten und Speedups auf `bluff`

	matrix-mul	matrix-mul2	stress
1 virtueller Prozessor	4,6763s	3,5415s	4,7224s
2 virtuelle Prozessoren	2,5053s	1,8025s	2,8319s
Speedup 1 auf 2 VPs	1,87	1,96	1,68

Tabelle 3: Realzeiten und Speedups auf `galibier`

Tabellen enthalten außerdem die jeweiligen Speedups, die durch die Verdoppelung der Anzahl der Prozessoren erzielt wurden.

Die Speedups werden dabei hin zur größeren Anzahl an Prozessoren kleiner. Außerdem sind die Speedups, bis auf eine Ausnahme, bei `matrix-mul2` größer als bei `matrix-mul`, und dort wiederum größer als bei `stress`. Dies entspricht auch der Reihenfolge der Benchmarks inbezug auf die Unabhängigkeit ihrer Threads.

Die genaue Interpretation der Ergebnisse ist aber sehr schwierig, da viele Faktoren zusammenspielen. Dazu gehören beispielsweise der Synchronisationsaufwand der Threads, Verzögerungen am Beginn der Berechnungen, die sich insbesondere bei sehr kurzen Laufzeiten auswirken können, die Synchronisierungen innerhalb der virtuellen Maschine, andere laufende Programme auf dem Rechner, bis hin zu Cache-Misses in der realen Maschine. Eine detaillierte Analyse nach der Rolle der einzelnen Faktoren konnte im Rahmen dieser Arbeit nicht vorgenommen werden. Insbesondere die Speedups von ein auf zwei, und von zwei auf vier Prozessoren, zeigen jedoch bereits, daß die Multiprozessorerweiterung die zusätzliche Rechenleistung zu einem sehr großen Teil in reale Laufzeitverbesserungen umsetzen kann.

4.2 Allokation und Speicherverwaltung

Wie in Abschnitt 2.3.3 erwähnt, stellt sich für die Minimalgröße der Creation-Spaces der Speicherverwaltung ein Optimierungproblem. Bei größeren Creation-Spaces erhöht sich der ungenutzte Speicherplatz und damit die Anzahl der notwendigen Speicherbereinigungen, bei kleineren Creation-Spaces erhöht sich die Anzahl der synchronisierten Allokationen. Tabelle 4 zeigt eine Messung dieser Größen für die Benchmarks `matrix-mul2` und `stress` auf dem Rechner `bluff`, sowie die jeweiligen Laufzeiten. Dabei wurden jeweils die Creation-Space-Größen

Benchmark `matrix-mul2`

VPs	Größe	GCs	Ungenutzte Bytes	Sync. Allok.	Laufzeit
1	128 kB	3	132900	32887	4,1546s
2	”	3	263700	34577	2,1278s
4	”	3	607252	37304	1,0826s
8	”	3	1390568	49932	0,7522s
1	32kB	3	37892	33382	4,1857s
2	”	3	67536	35113	2,1299s
4	”	3	154784	37805	1,0986s
8	”	3	340628	49077	0,7101s
Benchmark `stress`					
1	128 kB	35	1281512	1349582	5,519s
2	”	39	2844392	1514070	3,2415s
4	”	44	6566956	1727545	1,89s
8	”	50	25832204	1910374	1,22025s
1	32 kB	34	352400	1351284	5,502s
2	”	38	1265248	1498596	3,268s
4	”	44	2769364	1610136	1,9225s
8	”	49	7553196	1910613	1,23375s

Tabelle 4: Tests verschiedener Creation-Space-Größen

128kB und 32kB mit einem, zwei, vier und acht virtuellen Prozessoren getestet. Der Scheme-Heap hat dabei eine Gesamtgröße von 12 Megabytes.

Die Spalten der Tabelle enthalten von links nach rechts die Anzahl der virtuellen Prozessoren, die Minimalgröße der Creation-Spaces, die Anzahl der Speicherbereinigungen, die Gesamtgröße des ungenutzten Speichers in Bytes, die Anzahl der synchronisierten Allokationen, sowie die Laufzeit des Benchmarks. Dabei ist anzumerken, daß die Gesamtgröße des ungenutzten Speicherplatzes relativ starken Schwankungen von bis zu 20% unterworfen ist.

Die Laufzeiten der Benchmarks unterscheiden sich zwischen diesen beiden Creation-Space-Größen nur geringfügig. Ein Trend zur leichten Verschlechterung der Laufzeiten bei 32kB lässt sich aber deutlich erkennen. Ebenso steigt die Anzahl der synchronisierten Allokationen in 13 von 16 Fällen leicht an. Insgesamt lässt sich jedoch sagen, daß auch eine Creation-Space-Größe von 32kB ausreichend ist. Da aber anzunehmen ist, daß die optimale Creation-Space-Größe auch nicht unerheblich vom Speicherbedarf des jeweiligen Scheme-Programms abhängt, könnte sich eine Adaptionsstrategie lohnen, die die Größe der Creation-Spaces zur Laufzeit an das Allokationsverhalten des Programms anpasst.

Ein weiterer Test mit einer minimalen Creation-Space-Größe (also immer nur genau so viele Bytes, wie jeweils reserviert werden sollen), und einer Synchronisierung der Allokationsfunktion des Preallocation-Systems, zeigt jedoch auch, wie wichtig die effiziente Implementierung der Speicherallokation ist. Bei dem wesentlich speicherintensiveren Benchmark `stress`, ergibt sich dabei auf dem Rechner `bluff` und vier virtuellen Prozessoren eine Erhöhung der Laufzeit um etwa 40%.

Es lässt sich damit sagen, daß sich die Realisierung der synchronisationsfrei-

Benchmark	Original Scheme-48	Multiprozessorversion	Steigerung
`matrix-mul`	7,4289s	8,2914s	11,6%
`matrix-mul2`	5,6859s	6,3297s	11,3%
`stress`	7,6844s	8,6071s	12,0%
`fib`	4,461s	4,993s	11,9%

Tabelle 5: Overhead mit einem virtuellen Prozessor

en Allokation innerhalb der Creation-Spaces sehr positiv auf die Effizienz des Systems ausgewirkt, jedoch die konkrete Größe der Creation-Spaces keine große Auswirkung mehr hat.

4.3 Vergleich mit der Uniprozessorversion

Der Vergleich mit dem unveränderten Scheme-48-System ist für die konkrete Einsatzfähigkeit der Multiprozessorerweiterung, beziehungsweise für die Nützlichkeit eines Rechners mit mehreren Prozessoren für eine Scheme-Anwendung entscheidend. Nur wenn ein Multiprozessorsystem im Vergleich zu einem schnelleren, gleich teuren einzelnen Prozessor eine höhere Performance bringt, lohnt sich auch dessen Einsatz.

Im Vergleich des unveränderten Scheme-48-Systems mit der Multiprozessorversion auf einem Rechner mit nur einem Prozessor, zeigt sich zunächst der sogenannte *Overhead* der Multiprozessorerweiterungen. Er ist ein Maß für den zusätzlichen Aufwand, der für die Multiprozessorfähigkeit getrieben werden muß, wie zum Beispiel die Synchronisierung einiger Datenstrukturen, oder die zusätzlichen Indirektionen des Registerzugriffs in der virtuellen Maschine. Ein gewisser Overhead lässt sich nicht vermeiden, jedoch könnte bei einem genügend kleinen Overhead auf eine separate Entwicklung der Multi- und Uniprozessorversion von Scheme-48 verzichtet werden, was sowohl Vorteile für die Entwickler, als auch die Benutzer bedeuten würde.

Zur Messung des Overheads wurden die drei in Abschnitt 4.1 beschriebenen Benchmarks, sowie ein weiteres, nicht nebenläufiges Benchmark-Programm verwendet, das eine einfache Fibonacci-Zahlen-Berechnung durchführt. Die Laufzeitmessungen wurden auf einer Einzelprozessormaschine mit einer 600 MHz AMD Athlon CPU, sowie einem FreeBSD 5.4 Kernel vorgenommen. Tabelle 5 enthält die gemessenen Laufzeiten, sowie die relative Steigerung der Laufzeit von der Uni- zur Multiprozessorversion.

Durchschnittlich zeigt sich eine Steigerung um circa 12%, was bereits ein sehr kleiner Wert ist. Der Overhead ist damit sicherlich noch zu groß, um die Multiprozessorerweiterung in die normale Entwicklungsversion von Scheme-48 aufzunehmen, es bedeutet jedoch auch, daß der Overhead klein genug ist, um eine absolute Geschwindigkeitsverbesserung bereits mit zwei Prozessoren zu ermöglichen. Tabelle 6 zeigt daher die Laufzeiten der drei nebenläufigen Benchmarks auf der Uniprozessorversion, und auf der Multiprozessorversion mit zwei virtuellen Prozessoren, gemessen auf dem Rechner `galibier` (siehe Abschnitt 4.1), sowie die relativen Laufzeitverbesserungen, also die Speedups.

Die Abweichung dieser Speedups vom Idealwert 2 hat eine ganze Reihe von Ursachen. Dazu gehört zunächst der oben erwähnte Overhead der Multiprozes-

Benchmark	Original Scheme-48	Zwei Prozessoren	Speedup
`matrix-mul`	4,461s	2,41835s	1,84
`matrix-mul2`	3,3898s	1,8504s	1,83
`stress`	4,5094s	2,73515s	1,64

Tabelle 6: Speedup mit zwei virtuellen Prozessoren

sorerweiterung im Vergleich zur Uniprozessorversion. Desweiteren auch die Zeit, in der ein Kernel-Thread beispielsweise auf ein Mutex-Lock wartet, oder die Zeit, die für die Wiederholung von fehlgeschlagenen Commits des Proposal-Systems aufgewendet werden muß. Außerdem reduziert auch jede Speicherbereinigung den Speedup, da alle virtuellen Prozessoren während einer Speicherbereinigung angehalten werden müssen. Schließlich spielt auch die Hardware des Rechners eine Rolle, da Multiprozessoranwendungen üblicherweise eine wesentlich höhere Anzahl Cache-Misses aufweisen, oder der maximale Speicher-Durchsatz einen limitierenden Faktor für den Speedup spielt. Alle diese Faktoren lassen sich zwar minimieren, aber prinzipiell nicht gänzlich vermeiden. Die hier erreichten Speedups von circa 1,8 für die ersten beiden Benchmarks, und circa 1,6 für den wesentlich speicher- und synchronisationsintensiveren Benchmark `stress`, sind aber bereits in dem Bereich, der von Multiprozessoranwendungen üblicherweise erwartet werden kann. Die in dieser Arbeit vorgestellte Implementierung ist also durchaus schon sehr effizient.

5 Ausblick und Zusammenfassung

Diese Arbeit hat gezeigt, daß das Scheme-48-System zu einem leistungsfähigen Multiprozessor-Interpreter umgebaut werden kann. In der virtuellen Maschine wurden dabei Kernel-Level-Threads verwendet, um dem Laufzeitsystem virtuelle Prozessoren anzubieten, die jeweils eine Scheme-Funktion echt parallel ausführen können. Durch die Erweiterung des Pre-Scheme-Compilers, konnte dabei eine größere Änderung am Quelltext der virtuellen Maschine umgangen werden. Die Speicherverwaltung wurde so angepasst, daß sie eine effiziente parallele Allokation im gemeinsamen Haldenspeicher ermöglicht, und eine konsistente Speicherbereinigung gewährleistet. Durch die Änderungen am I/O-System können alle virtuelle Prozessoren fehlerfrei auf jede offene Datei zugreifen. Ferner wurde im Laufzeitsystem das User-Level-Thread-System an die veränderten Umstände angepasst, und um den parallelen Scheduler erweitert, der automatisch die Verteilung von User-Level-Threads auf mehrere virtuelle Prozessoren vornehmen kann. Schließlich haben Messungen mit Benchmark-Programmen gezeigt, daß der Multiprozessor-Interpreter weitere Prozessoren effizient in geringere Laufzeiten umsetzen kann, und daß der Overhead für die Multiprozessorfähigkeit sehr gering ist.

5.1 Ausblick

Dieser Abschnitt soll einen Überblick über die Bereiche geben, die für einen voll einsatzfähigen multiprozessorfähigen Interpreter noch verändert werden müssen, und auf weitere Möglichkeiten zur Verbesserung und Erweiterung hinweisen.

Kommando-Prozessor Der Kommando-Prozessor ist Teil des Laufzeitsystems vom Scheme-48. Er dient dazu Scheme-Programmcode zur Laufzeit des Interpreters zu laden, dem Benutzer die Möglichkeit zur interaktiven Eingabe und Auswertung von Ausdrücken, sowie Unterstützung beim Debugging und der Ausnahmebehandlung anzubieten. Es erscheint dem Autor sinnvoll, die Kontrolle über die virtuellen Prozessoren in den Kommando-Prozessor zu integrieren, sodaß das Multiprocessing harmonisch mit den anderen Features des Kommando-Prozessors vereinbart werden kann.

Windows Implementierung Diese Arbeit hat sich auf Betriebsysteme beschränkt, die POSIX-Threads anbieten. Windows beinhaltet jedoch eine andere Schnittstelle für Kernel-Level-Threads. Eine Implementierung der Betriebsystemrelevanten Teile für die Win32-Plattform steht daher noch aus.

Speicherverwaltung Scheme-48 enthält eine alternative Speicherverwaltung, BIBOP genannt, die in der Zukunft das bisherige System ersetzen soll. Dieses neue System enthält einige Hürden für eine effiziente Multiprozessorversion, wie beispielsweise die Verwendung einer sogenannten *Write-Barrier*, aber durch seine Struktur auch Vorteile für die Implementierung prozessorspezifische Speicherbereiche, wie die Creation-Spaces. Eine Analyse dieses Systems aus der Sicht des Multiprocessing, sowie eine Anpassung und eventuelle Erweiterung, sollte daher vorgenommen werden.

Primitive Synchronisierung Die Notwendigkeit zur Synchronisation der virtuellen Prozessoren aus dem Laufzeitsystem heraus, ist nicht zu vermeiden. In dieser Arbeit wurde dazu dem Laufzeitsystem Primitiva für die Mutex-Locks und die Condition-Variablen des Betriebsystems zur Verfügung gestellt. Eine zu erwägende Alternative könnte sein, das Provisional-System zur optimistischen Nebenläufigkeit um Zugriffsfunktionen auf primitive globale Daten und Zustände zu erweitern. Ein Beispiel ist das globale Feld `*session-data*`, das in dieser Arbeit vor verschränkten Lese- und Schreibzugriffen durch ein primitives Mutex-Lock geschützt werden musste. Durch spezielle Zugriffsfunktionen im Sinn des Proposal-Systems, und eine entsprechende Erweiterung dieses Systems, könnte diese explizite Synchronisierung im Laufzeitsystem unnötig machen, und stattdessen in das Proposal-System integrieren werden. Entsprechend könnte zum Beispiel die Implementierung des Thread-Systems von einer Provisional-Funktion zum Setzen eines Timer-Interrupts über die Funktion `schedule-interrupt` profitieren. Die dabei notwendige Synchronisierung wurde in dieser Arbeit durch ein eigenes Mutex-Lock in der virtuellen Maschine realisiert. Inwieweit hierbei aber der Overhead des Proposal-Systems, und die dadurch etwas gröbere Granularität der Synchronisierung, ins Gewicht fällt, müsste untersucht werden.

Literatur

[1] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970. ACM Press.

[2] Edsger W. Dijkstra. *Cooperating sequential processes.* Technological University, Eindhoven, The Netherlands, September 1965. Reprinted in Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, 43-112.

[3] R. Kent Dybvig. *The Scheme Programming Language: ANSI Scheme.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[4] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Journal of Computer Languages*, 14, 2:109–123, 1989.

[5] Christos Freris. Evaluierung des BIBOP-Garbage-Collectors MULCON für Scheme48. Studienarbeit. Universität Tübingen, 2004.

[6] Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard A. Kelsey. How to add threads to a sequential language without getting tangled up. *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, 2003.

[7] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Comput. Lang.*, 12(2):109–121, 1987. Pergamon Press, Inc.

[8] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[9] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 73–82, New York, NY, USA, 1993. ACM Press.

[10] Suresh Jagannathan, C. R. Kirkwood-Watts, Richard Kelsey, and Jonathan Rees. Proposals: High-level data structures for optimistic concurrency. Unpublished. 2002.

[11] R. Kelsey. Pre-scheme: A Scheme dialect for systems programming.

[12] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[13] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. ACM Press.

[14] P. J. Plauger. *The standard C library.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[15] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1995.

[16] J. H. Reppy. Synchronous operations as first-class values. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 250–259, New York, NY, USA, 1988. ACM Press.

[17] John Reppy. Higher-order concurrency. *Computer Science Technical Report 92-1285*, June 1992. Cornell University.

[18] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In *OOPSLA '91: Conference proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 16–32, New York, NY, USA, 1991. ACM Press.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 8. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2001.

[20] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 2. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2001.

[21] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.