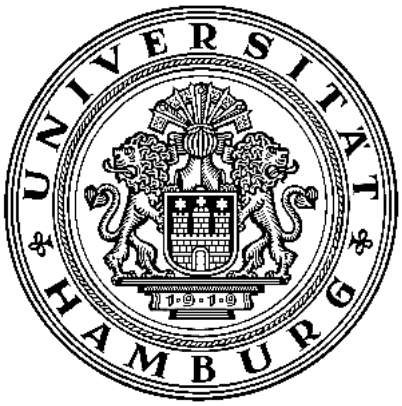

Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

STUDIENARBEIT

Dynamische Maschinencodegenerierung für das Tycoon-System

Oktober 1996

Marc Weikard
Gänseblümchenweg 18
21614 Buxtehude

Tel.: 04161 / 62239

Betreuer:
Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

1. Einleitung **1**

1.1 Zielsetzung 3

1.2 Übersicht 4

2. Vorarbeiten **5**

2.1 Maschinencodegenerierung in persistenten Systemen 5

2.2 *just in time* Übersetzung für Java 6

3. Das Tycoon-System **7**

3.1 Die Tycoon-Architektur 7

3.1.1 Die Programmiersprache 8

3.1.2 Die Zwischenrepräsentation 9

3.1.3 Das Laufzeitsystem 9

3.1.4 Die Objektspeicherschnittstelle 10

4. Die virtuelle Maschine **11**

4.1 Funktionsobjekte 11

4.2 Bytecode 12

4.3 Stapelspeicher 14

4.4 Ausnahmebehandlung 14

4.5 Parallele leichtgewichtige Prozesse 15

4.6 Persistente Threads 16

4.7 Migration 17

4.8 C-Calls 17

4.9 Callbacks 17

5. Die Maschinencodegenerierung **19**

5.1 Integration in die Tycoon-Architektur 20

5.2 Die Wahl von C als portable Zwischensprache 22

5.3 Lösung spezieller Implementierungsprobleme 23

5.3.1 Einbindung externen Programmcodes zur Laufzeit 23

5.3.2 Caching von Funktionen 23

5.3.3 Freispeicherverwaltung 24

5.3.4 Persistente Sicherungspunkte 25

5.3.5 Parallele Threads 28

5.3.6 Ausnahmebehandlung 28

5.4 Mögliche Optimierungen des Codegenerators 30

6. Das Laufzeitverhalten **33**

6.1 Performanz 33

6.2 Einfluß der Persistenz 35

6.3 Aufwand der Codegenerierung 37

6.4 Entwicklung der Codegröße 38

7. Zusammenfassung **41**

7.1 Stand der Implementierung 41

7.2 Bewertung und Ausblick 42

Literaturverzeichnis **45**

1. Einleitung

Informationssysteme, d.h. Systeme, die die kooperative Arbeit von Menschen unterstützen und koordinieren, sind gekennzeichnet durch ihre Langlebigkeit und Kompexität, ihre verteilte Ausführung in Netzwerkumgebungen und ihren Bestand an Massendaten. Die Entwicklung von Informationssystemen auf der Basis hierarchischer oder relationaler Datenbanksysteme ist von einer wesentlichen Einschränkung betroffen: Die Persistenz erstreckt sich nur auf die Datenobjekte selbst. Programme und langlebige Aktivitäten, z.B. Geschäftsprozesse, die auf diesen Daten operieren, müssen außerhalb des Datenbanksystems in einer für jeden Rechner spezifischen Darstellung vorliegen (Abbildung 1.1).

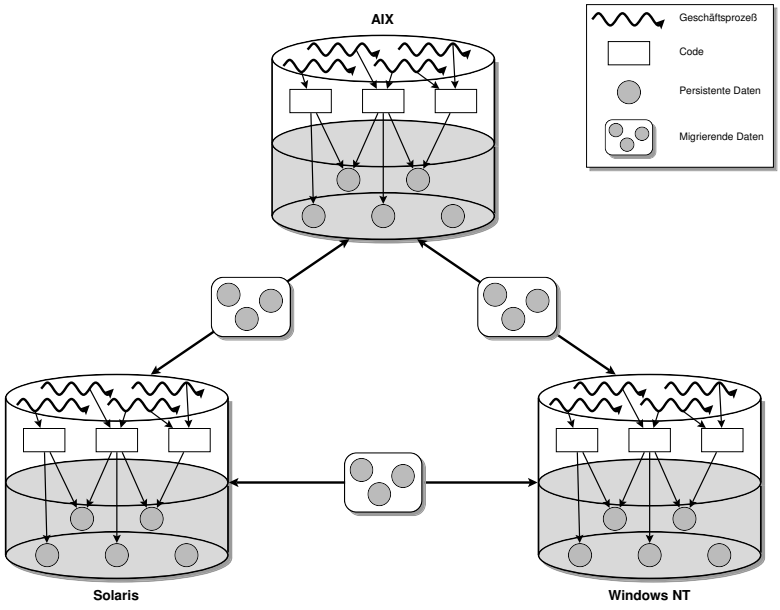

Abbildung 1.1: Informationssystem auf Basis relationaler Datenbanken

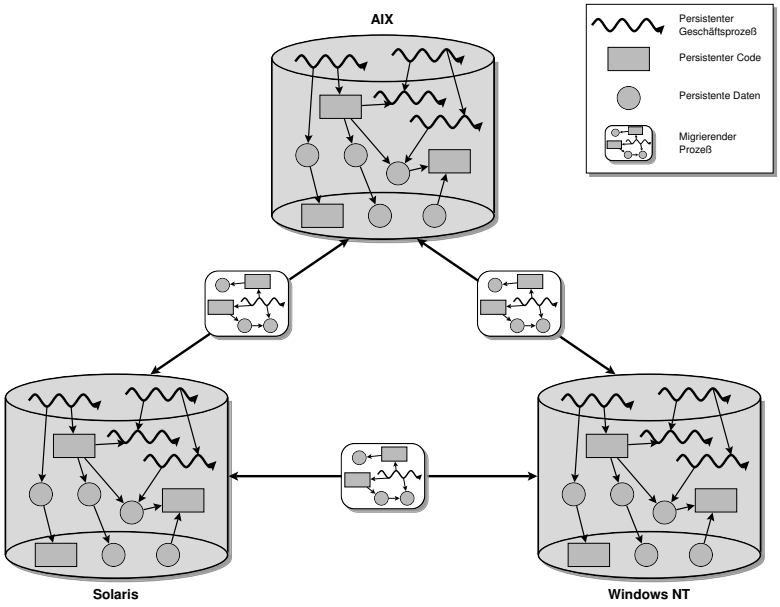

Abbildung 1.2: Informationssystem auf Basis einer persistenen Programmierumgebung

Dieser Mangel zeigt sich einerseits im Systemdesign, zum anderen wirkt er sich auf den Informationsfluß innerhalb des Systems aus. Da Informationssysteme zu einem großen Teil in heterogenen Client-Server Netzwerkumgebungen eingesetzt werden, können nur die persistenten Daten, für welche genormte Übertragungsprotokolle[1] existieren, gemeinsam auf verschiedenen Rechnerplattformen genutzt werden. Der Austausch von Programmen ist aufgrund von unterschiedlichen Betriebssystemen und Prozessorarchitekturen nur durch eine Portierung auf den jeweiligen Zielrechner möglich.

Integrierte, persistente Programmierumgebungen für datenintensive Applikationen wie das Tycoon[2]-System [MSS96] bieten als Erweiterung gegenüber traditionellen Datenbanksystemen orthogonale Persistenz, die sich auf Daten, Funktionen und in Ausführung befindliche Prozesse erstreckt, welche in einem persistenten Objektspeicher gehalten werden (Abbildung 1.2).

Um neben Daten auch Funktionen und Prozesse zwischen inkompatiblen Rechnern auszutauschen, existiert eine einheitliche Darstellung von Basisdatentypen und Programmcode. Mit der Definition einer virtuellen Maschine, einer Schnittstelle, die die Coderepräsentation und ein plattformunabhängiges Ausführungsmodell beschreibt, wird von der zugrundeliegenden Rechnerarchitektur abstrahiert. Die virtuelle Maschine wird auf jeder Plattform von einem Interpreter emuliert. Auf diese Weise kann die Frage der Portabilität in orthogo-

[1] CCITT X.409, OSI ASN.1 und BER, Sun Microsystems XDR
[2] Tycoon: *Typed Communicating Objects in Open Environments*

nal persistenten Systemen gelöst werden. Die interpretierende Ausführung bedingt jedoch einen im Vergleich zu plattformspezifischen Maschinencode erheblichen Verlust an Performanz, der sich vor allem bei Applikationen, die mit dem Benutzer interagieren, negativ bemerkbar macht. Aus diesem Grund müssen Möglichkeiten, die Programmausführung zu beschleunigen, untersucht werden:

1. Die Optimierung der Codeerzeugung in persistenten Systemen, wie sie in [Kir94] für das Tycoon-System durchgeführt wurde.
2. Die in dieser Arbeit vorgestellte dynamische Generierung von Maschinencode.

Die dynamische Maschinencodegenerierung übersetzt Programmcode, der in einer plattformunabhängigen Repräsentation vorliegt, erst unmittelbar vor der Ausführung (*just in time*) in Maschinencode für die jeweilige Rechnerarchitektur. Die Portabilität wird bei dieser Art der Codegenerierung nicht berührt, da der Daten- und Programmaustausch weiterhin auf der plattformunabhängigen Ebene erfolgt. Durch den Einsatz existierender Codegeneratoren kann auch die Implementierung des dynamischen Codegenerators auf portable Weise erfolgen und so der Implementierungsaufwand vermindert werden.

1.1 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung eines dynamischen Codegenerators für die persistente Programmierumgebung Tycoon. Von vorrangiger Bedeutung ist dabei eine Implementierung, die weiterhin die Portabilität des Systems garantiert und zu keiner Einschränkung der Funktionalität führt.

Neben der eigentlichen Technik der dynamischen Codeerzeugung soll ein Weg aufgezeigt werden, wie ein Codegenerator in das Tycoon-Schichtmodell integriert werden kann. Von Bedeutung sind hierbei die Vorteile, die sich aus der Nutzung existierender plattformübergreifender Werkzeuge zur Codeerzeugung ergeben.

Das Tycoon-System stellt besondere Anforderungen an seine Laufzeitumgebung, die von Seiten des dynamischen Codegenerators berücksichtigt und unterstützt werden müssen. Für diese sind Lösungsmöglichkeiten zu erarbeiten und zu realisieren. Im Einzelnen betrifft dies:

- Einbindung von dynamisch erzeugtem Code in ein laufendes System
- Persistente Sicherungspunkte
- Persistente leichtgewichtige Prozesse
- Ausführung paralleler Prozesse
- Ausnahmebehandlung
- Unterstützung einer Freispeicherverwaltung

Daneben ist die Frage zu beantworten, wie gut sich das Ausführungsmodell der virtuellen Maschine auf eine tatsächliche Rechnerarchitektur umsetzen läßt, und inwieweit Optimierungen während der Transformation von der plattformunabhängigen Repräsentation zum Maschinencode möglich sind.

Eine Analyse sowohl der durch den Maschinencode erzielten Performanzsteigerung als auch des nötigen Generierungsaufwands soll eine realistische Abschätzung hinsichtlich der Effizienz der implementierten Lösung ermöglichen. Zusätzlich soll das Potential der dynamischen Maschinencodegenerierung näher beleuchtet werden, um zukünftige Entscheidungen, die ihren Einsatz in Weiterentwicklungen des Tycoon-Systems (Tycoon2) betreffen, zu erleichtern.

1.2 Übersicht

Maschinencode wurde bereits mit Erfolg für die persistente Programmiersprache TL in einer älteren Version des Tycoon-Systems generiert. Kapitel 2 beschreibt kurz, welche veränderten Rahmenbedingungen eine neue Implementierung notwendig machen.

Eine Übersicht des Tycoon-Schichtmodells wird in Kapitel 3 gegeben. Im daran anschließenden Kapitel folgt eine genauere Betrachtung der virtuellen Maschine und des Laufzeitsystems unter besonderer Berücksichtigung der für die Implementierung relevanten Details.

Die Technik der dynamischen Codegenerierung und die Einbindung des Codegenerators in das Gesamtsystem werden in Kapitel 5 dargestellt. Hier werden auch die Probleme, die sich in persistenten Systemen ergeben, erläutert sowie Lösungsmöglichkeiten aufgezeigt.

Schließlich erfolgt in Kapitel 6 eine Analyse des in einer Reihe von Testmessungen ermittelten Performanzgewinns und des bei der Codeerzeugung entstehenden Aufwands.

2. Vorarbeiten

Zwei Techniken werden in dieser Arbeit vereint, zum einen die Generierung von Maschinencode für persistente Programmiersprachen und zum anderen die dynamische Codegenerierung aus einer plattformunabhängigen Repräsentation, wie sie von Smalltalk-80 [DS84] und Java [GM95] bekannt ist.

2.1 Maschinencodegenerierung in persistenten Systemen

Maschinencodegenerierung für die persistente, polymorphe Programmiersprache TL (*Tycoon Language*) wird bereits in [Mat92] beschrieben. Grundlage der Implementierung ist dort eine ältere, noch auf Quest [Car90] basierende Version des Tycoon-Systems.

Ausgangspunkt der Codegenerierung ist die Zwischenrepräsentation TML (*Tycoon Machine Language*), eine auf dem λ-Kalkül basierende Programmdarstellung, in die der TL-Quellcode nach erfolgter Syntax- und Typüberprüfung übersetzt wird. Aus dieser Darstellung wird – alternativ zur Abarbeitung durch einen TML-Interpreter – C-Quellcode erzeugt, welcher mittels eines externen C-Übersetzers in Maschinencode transformiert und direkt ausgeführt wird.

Im Zuge der Entwicklung eines von Quest unabhängigen Systems wurden, vor allem in Zusammenhang mit der Umgestaltung des Laufzeitsystems, zum Teil drastische Änderungen und Erweiterungen durchgeführt, so daß die bisherige Methode, Maschinencode zu generieren, aufgegeben werden mußte:

- Umstellung auf Bytecode
  Die Einführung einer neuen virtuellen Maschine, die nicht mehr auf TML sondern auf einer plattformunabhängigen, bytecodierten Darstellung von Programmen operiert, stellt die wichtigste Änderung des Laufzeitsystems dar. Eine modifizierte Variante von TML ist zwar weiterhin im System integriert, sie ist allerdings nur noch eine Zwischenrepräsentation in der Übersetzungsphase von TL zu Bytecode und wird ausschließlich zu Optimierungszwecken verwendet.

- Erweiterung der Funktionalität
  Mit der Umstellung auf ein neues Laufzeitsystem wurde die Möglichkeit geschaffen, dessen Funktionalität gegenüber dem alten System zu verbessern. Erwähnenswert sind hierbei die Unterstützung paralleler leichtgewichtiger Prozesse (*threads*) und die Erweiterung des Persistenzkonzeptes auf diese Prozesse.

- Neue innovative Konzepte
  Die plattformunabhängige Darstellung von Daten, Programmen und Prozessen ermöglicht innovative Konzepte hinsichtlich des Informationsaustauschs in heterogenen Netzwerken. Persistente Prozesse und die mit ihnen assoziierten Daten können zwischen unterschiedlichen Rechnerarchitekturen zur Laufzeit migrieren und im Netz verteilt werden.

Diese im Vergleich zur alten Tycoon-Version geänderten Vorraussetzungen bedingen einen neuen Ansatz für der Implementierung eines Maschinencodegenerators: die Abkehr von der bisherigen statischen[1] Codeerzeugung hin zu einer dynamischen, *just in time* Übersetzung der plattformunabhängigen Programmrepräsentation in Maschinencode.

2.2 *just in time* Übersetzung für Java

Java von Sun Microsystems ist eine Programmiersprache, die die Entwicklung von Applikationen in heterogenen Netzwerkumgebungen in effizienter Weise erlaubt. Java Programme müssen aus diesem Grund auf einer Reihe unterschiedlicher Betriebssysteme und Rechnerarchitekturen ablaufen.

Der Java-Übersetzer erzeugt, genau wie das TL-Übersetzer des Tycoon-Systems, plattformunabhängigen Bytecode einer virtuellen Maschine. Die Distribution von Daten und Programmen zwischen unterschiedlichen Architekturen wird durch die eindeutige Repräsentation des Bytecodes und der Basisdatentypen ermöglicht. Diese Plattformunabhängigkeit ist wie im Tycoon-System mit dem Nachteil verbunden, daß Bytecode durch einen Interpreter ausgeführt wird und so gegenüber Maschinencode zu einer Performanzeinbuße führt.

Abhilfe versprechen u.a. die Firmen Borland und Symantek[2] durch spezielle *just in time* Compiler für ihre Java-Entwicklungsumgebungen. Anstatt den Bytecode zu interpretieren, wird dieser unmittelbar vor der Ausführung in Maschinencode übersetzt und direkt vom Prozessor abgearbeitet. Der erreichbare Geschwindigkeitsgewinn soll nach Angaben von Borland bei typischen Applikationen ca. den Faktor 5-10 erreichen, bei rechenintensiven Anwendungen zum Teil auch deutlich darüber liegen.

[1] statisch: TL-Übersetzer erzeugt Machinencode, dynamisch: Codeerzeugung erst unmittelbar vor der Programmausführung

[2] http://cafe.symantec.com/

3. Das Tycoon-System

Die Programmierumgebung Tycoon [Mat93] bietet einen geeigneten Rahmen zur effizienten Implementierung moderner Informationssysteme [MSS96]. Ihre offene und skalierbare Architektur eröffnet vielfältige Einsatzmöglichkeiten, von der Einzelplatzinstallation bis hin zur Nutzung in verteilten heterogenen Mehrbenutzerumgebungen.

Die persistente, polymorphe Programmiersprache TL (*Tycoon Language*) mit ihrem Typsystem höherer Ordnung ermöglicht die Beschreibung und Integration generischer Dienste in einem einheitlichen Kontext. Sie kombiniert Vorteile und Leistungsmerkmale integrierter Datenbankprogrammiersprachen wie DBPL [SM92], z.B. orthogonale Kombinierbarkeit elementarer Konzepte zur Persistenzabstraktion, typvollständige Datenrepräsentation und Iterationsabstraktion, mit der orthogonalen Persistenz von Systemen wie Napier88 [DCBM89] oder P-Quest [Mül91].

Die Integration verschiedener existierender Diensterbringer, z.B. relationaler Datenbanken oder grafischer Benutzerschnittstellen, durch polymorphe TL-Bibliotheken gewährleistet den typsicheren Zugriff auf externe Daten und Programme. Gemeinsame Benennungs-, Typisierungs- und Bindungsmechanismen erhöhen die Flexibilität und Korrektheit. Es existiert keine Beschränkung auf bestimmte vordefinierte Dienste oder Objekte, das System kann in einfacher Weise an gewünschte Erfordernisse angepaßt und erweitert werden.

Dieses Kapitel bietet einen Überblick über die Tycoon-Systemarchitektur, wobei vor allem auf die für die dynamische Codeerzeugung wichtigen Aspekte eingegangen wird.

3.1 Die Tycoon-Architektur

Abbildung 3.1 illustriert die Tycoon-Systemarchitektur. Das Tycoon-System wird in die Schichten Objektspeicher, Laufzeitsystem, Compiler-Backend, Compiler-Frontend und Anwendungen unterteilt, welche ihre Dienste mittels wohldefinierter Schnittstellen bereitstellen. Dieses Modell erlaubt die strikte Trennung der Funktionalität und so alternative Implementierungen für einzelne Schichten.

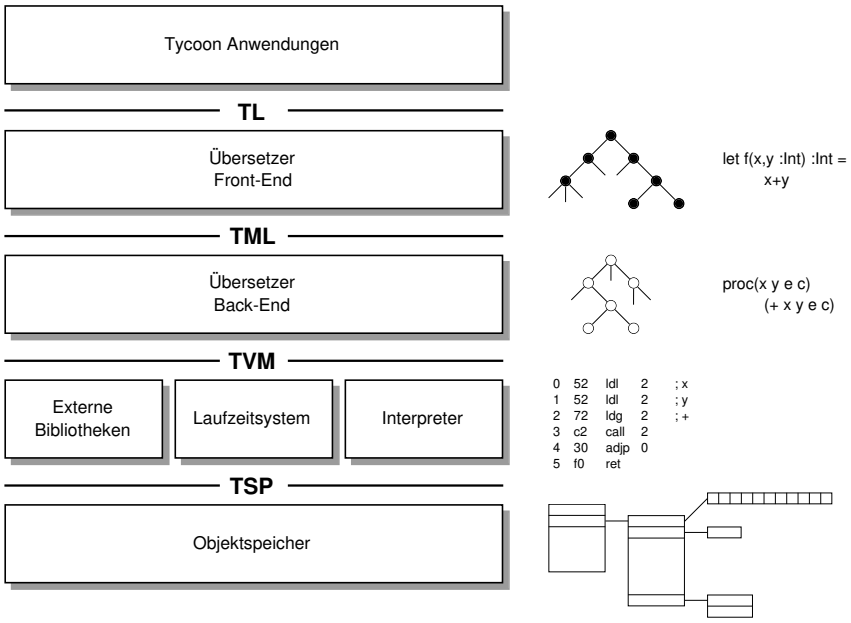

Abbildung 3.1: Tycoon-Schichtarchitektur

3.1.1 Die Programmiersprache

Die Mehrzweckprogrammiersprache TL[1] kann im wesentlichen als eine Weiterentwicklung von Quest und P-Quest angesehen werden. Semantisch ist TL mit den polymorphen funktionalen Sprachen der ML-Familie verwandt, in ihrer Modulstruktur und imperativen Konstrukten spiegeln sich darüber hinaus Konzepte der Modula-Sprachfamilie wieder.

Persistenz, Polymorphismus und Funktionen höherer Ordnung stellen grundlegende Konzepte der Programmiersprache TL dar. Die Unterstützung von Persistenz ist auf der TL-Ebene vollkommen transparent und erfolgt im Zusammenspiel mit dem Tycoon-Speicherprotokoll. Polymorphismus, ein wichtiges Werkzeug zur Erstellung generischer Bibliotheken, existiert in zwei Ausprägungen, als parametrischer und Subtyppolymorphismus. Funktionen können als Objekte erster Klasse sowohl als Argumente an andere Funktionen übergeben werden, als auch von Funktionen als Ergebnis zurückgegeben werden.

TL definiert nur einige wenige feste Datentypen, wie Ganz- und Fließkommazahlen, Zeichenketten und Wahrheits-Werte, außerdem Wertkonstruktoren für Felder, Tupel, Records, variante Tupel und Records sowie Ausnahmen. Die beliebige orthogonale Kombinierbarkeit dieser Konstruktoren zusammen mit rekursiven Typdefinitionen und Typoperatoren ermöglicht die Modellierung aller im praktischen Einsatz vorkommenden Datenstrukturen,

[1] *Tycoon Language*

wie Listen, Mengen, Relationen, etc. sowie darauf operierender polymorpher Funktionen.

TL kombiniert imperativen und funktionalen Programmierstil. Es werden imperative Kontrollstrukturen wie z.B. *loop*, *if*, *case* zur Verfügung gestellt. Darüber hinaus wird eine dynamische Ausnahmebehandlung (*exception handling*) unterstützt, um die Integration fehlerträchtiger Systeme zu vereinfachen.

Der Aufruf von C-Funktionen über eine einfache Schnittstelle ermöglicht die Anbindung der Programmiersprache TL an externe Bibliotheken und Dienste. Die typsichere Nutzung dieser Funktionen ist durch die Zuweisung passender Signaturen auf TL-Seite möglich. Die eigentliche Funktionalität, wie die dynamische Bindung der Bibliotheken und Konvertierung der Argumente bzw. Resultate, erfolgt durch das Laufzeitsystem.

3.1.2 Die Zwischenrepräsentation

Die Transformation des vom Compiler-Frontend erzeugten abstrakten TL-Syntaxbaums in die Zwischenrepräsentation TML[2] stellt eine wesentliche Vereinfachung der Programmanalyse und -optimierung dar. TML ist eine Darstellung des Programmcodes in *Continuation Passing Style* (*CPS*). CPS basiert auf dem λ-Kalkül und ist eine geeignete Repräsentation für funktionale Sprachen. Schleifen, Zuweisungen und andere imperative Konstrukte müssen in adäquater Weise transformiert werden.

CPS-Funktionen liefern keinen Wert zurück, sondern erhalten bei ihrem Aufruf ein zusätzliches Argument – die Fortsetzung (*continuation*). Diese Fortsetzung ist eine Funktion mit einem an das Funktionsergebnis zu bindenden Parameter, und stellt die Abgabe des Kontrollflusses durch einen Funktionsaufruf dar. Eine Kontrollflußanalyse, die die Grundlage für alle Optimierungen darstellt, wird somit überflüssig.

Ein großer Vorteil von TML liegt in der geringen Anzahl von Sprachkonstrukten, die von einem Optimierer gehandhabt werden müssen. Neben Funktionsabstraktion und -applikation sind dies Variablenzugriffe und elementare Operationen. Die eigentliche Funktionalität wird von den elementaren Operationen getragen, die alle arithmetischen und boolschen Operationen, die Zugriffe auf den Objektspeicher und die Definition rekursiver Funktionen realisieren.

Neben einfachen Standardoptimierungen, wie dem Propagieren von Konstanten oder der Auswertung konstanter Ausdrücke, kann der im Compiler-Backend implementierte Optimierer auch Funktionsaufrufe durch Funktionsexpansion (*inlining*) entfernen.

3.1.3 Das Laufzeitsystem

Die ausführliche Beschreibung der TVM[3] und des Laufzeitsystems unter besonderer Berücksichtigung der für die dynamische Codeerzeugung relevanten Aspekte erfolgt in Kapitel 4.

[2] *Tycoon Machine Language*
[3] *Tycoon Virtual Machine*

3.1.4 Die Objektspeicherschnittstelle

Das Speicherprotokoll TSP[4] definiert eine standardisierte Schnittstelle, die von den Details der zugrundelegenden Implementierung des persistenten Objektspeichers abstrahiert. Es beschränkt sich dabei auf einen Satz von Funktionen, die auch von existierenden kommerziellen und frei verfügbaren Objektspeichersystemen bereitgestellt werden.

Eine der wichtigsten Eigenschaften des TSP ist die Unterstützung polymorpher, inhomogener Speicherobjekte. Um dies auf einfache Weise zu erreichen, wird ein untypisiertes Objektspeichermodell verwendet, in dem alle Werte ein einheitliches Layout und Markierungsschema (*tagging*) zur Unterscheidung der einzelnen Datentypen besitzen. Auf allen Systemumgebungen haben diese Speicherworte eine Größe von 32 Bit, von denen die beiden niederwertigsten Bits zur Markierung dienen und der Rest Nutzdaten enthält.

Jedes Objekt besitzt eine Identität, die OID (*object identifier*), über die die Referenzierung und Bindung an andere Objekte erfolgt. Objekte können nur erzeugt, jedoch nicht wieder freigegeben werden. Die Freigabe erfolgt durch eine automatische Freispeicherverwaltung (*garbage collection*), die alle von einem speziell ausgezeichneten Wurzelobjekt transitiv erreichbaren Objekte persistent hält.

Objektreferenzen können durch die Freispeicherverwaltung geändert werden. Alle weiteren im Objektspeicher gehaltenen, jedoch nicht von der Wurzel aus erreichbaren Zustandsvariablen, wie die Ausführungsumgebung der virtuellen Maschine, müssen daher explizit der Kontrolle der Freispeicherverwaltung unterstellt werden. Eine dem Objektspeicher bekanntgegebene Aufzählungsfunktions, die während der Speicherfreigabe ausgeführt wird, erfüllt diesen Zweck.

[4] *Tycoon Store Protocol*

4. Die virtuelle Maschine

Die TVM (*Tycoon Virtual Machine*) stellt die Schnittstelle zum Tycoon-Laufzeitsystem dar. Sie definiert eine funktionale, kellerbasierte virtuelle Maschine mit einem bytecodierten Befehlssatz. Der unter Nutzung des unterliegenden Tycoon-Speicherprotokolls persistent gehaltene Programmcode wird durch einen im Laufzeitsystem integrierten Interpreter ausgeführt.

Die konsequente Implementierung des TVM-Interpreters und des zugehörigen Laufzeitsystems in ANSI C ermöglicht die Portierung auf eine Reihe unterschiedlicher Systemumgebungen, unter anderem Solaris, Windows NT, Linux und MacOS.

Die Plattformunabhängigkeit von TVM und TSP-Modell gestattet neben der Verwendung desselben Bytecodes auf allen Systemen sogar die Migration in Ausführung befindlicher leichtgewichtiger Prozesse (*threads*) in heterogenen Netzwerkumgebungen.

4.1 Funktionsobjekte

Die Programmiersprache TL unterstützt Funktionen höherer Ordnung, d.h. Funktionen können Argumente und Resultate anderer Funktionen sein. Die einfache Repräsentation einer Funktion als Bytecode reicht nicht aus, dieses Konzept zu realisieren.

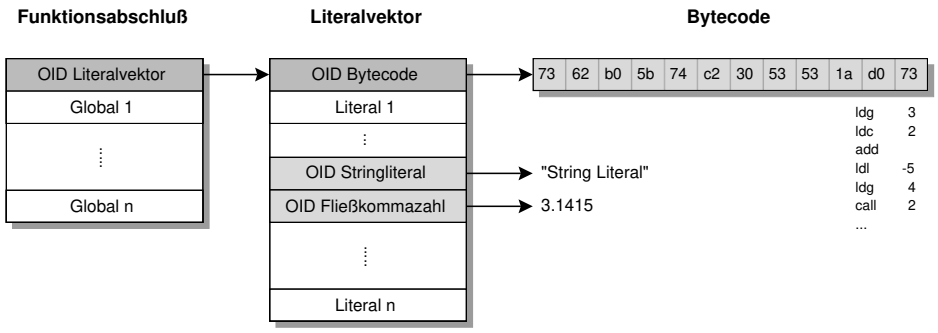

Abbildung 4.1: Funktionsobjekt der virtuellen Maschine

Funktionen stellen im TVM-Modell aus 3 Komponenten bestehende Objekte dar (siehe Abbildung 4.1):

- Funktionsabschluß (*closure*).
  Alle Funktionsabstraktionen liefern in TL gleichberechtigte Werte. Dies trifft insbesondere für geschachtelte Funktionen zu, die auf Datenobjekte ihres jeweiligen Sichtbarkeitsbereichs zugreifen. Da sich der sie umgebende Kontext jedoch zu ihrer Lebenszeit ändern kann, müssen alle von ihnen eingegangenen globalen Bindungen fixiert werden. Der Funktionsabschluß ist eine Datenstruktur, in die die OIDs der referenzierten globalen Objekte eingetragen werden. Sie wird dynamisch erzeugt.

- Literalvektor.
  Die Basisdatentypen Fließkommazahl und Zeichenkette (*string*) können nicht direkt im Bytecode dargestellt werden. Für sie werden zur Übersetzungszeit eigene Objekte erzeugt, deren OIDs in den Literalvektor eingetragen werden.

- Bytecode.
  Der Bytecode ist eine Folge bytecodierter Instruktionen der virtuellen Maschine, die vom TVM-Interpreter abgearbeitet wird. Er wird vom Compiler-Backend aus der Zwischenrepräsentation TML erzeugt.

4.2 Bytecode

Die Spezifikation der TVM-Byteinstruktionen lehnt sich an die von Transputern bekannten Befehlssätze an. Alle Opcodes weisen eine feste Größe von 8 Bit auf, von denen die 4 höherwertigen Bit den Operator und die restlichen 4 Bit den Operanden kodieren. Da eine Restriktion auf 16 Operationen bzw. des Operandenbereichs auf 16 Werte die Verwirklichung des kompletten Befehlssatzes nicht gestattet, besitzen 3 Instruktionen eine spezielle Semantik:

- *PFIX* und *NFIX* für die vorzeichenbehaftete Erweiterung des Operanden des Folgebefehls um 4 Bit. Durch wiederholte Ausführung lassen sich beliebige Operandengrößen erreichen.

- *OPR* interpretiert seinen Operanden als neuen Operator und erweitert so, insbesondere in Verbindung mit *PFIX/NFIX*, den Instruktionssatz.

Neben mathematischen und logischen Funktionen und Operationen zur Ablaufsteuerung existieren auch Bytecodes für Objektspeicherzugriffe. Diese garantieren die Atomarität der punktuellen Zustandsänderungen und vereinfachen dadurch sowohl das TSP wie auch die Implementierung der höheren Systemebenen. Tabelle 4.1 zeigt eine Übersicht der vorhandenen Instruktionen.

LDL, STL	Laden und Modifikation von lokalen Variablen
LDC	Laden von ganzzahligen Konstanten
LDOK, LDFALSE, LDTRUE	Laden spezieller Konstanten
LDG	Laden von globalen Variablen
LDLC	Laden von Literalen
LDP, STP	Feldzugriff und Modifikation
LDB, STB	Byte-Feldzugriff und Modifikation
ADD, SUB, MUL, DIV, MOD, NEG	ganzzahlige Arithmetik
GT, GE, LT, LE	ganzzahlige Vergleiche
NOT, AND, OR, XOR	ganzzahlige boolsche Operationen
SHL, SHR	Bit-Schiebeoperationen
C2I,I2C	Konvertierung von Ganzzahlen und Zeichen
PUSHH, POPH	Installation/Entfernung eines Ausnahmepakets
RAISE	Auslösen einer Ausnahme
J, CJ, JE	Unbedingte und bedingte Sprünge
SWITCH	Bedingter Sprung mit mehreren Sprungzielen
CALL	Aufruf einer Funktion
CCALL	Aufruf einer externen C-Funktion
RET, RETURN	Rückkehr aus einer Funktion
ADJ, ADJP	Korrektur des Stapelzeigers
CLOSURE	Erzeugen eines Funktionsabschlusses
ARRAY	Erzeugen eines veränderbaren Feldes
VECTOR	Erzeugen eines unveränderbaren Feldes
NSLOTS	Größe eines Feldes
MOVE, BMOVE	Kopieren von Feldbereichen
PFIX, NFIX	Erweiterung des Befehlsoperanden
OPR	Erweiterung der Instruktionssatzes

Tabelle 4.1: Befehlssatz der virtuellen Maschine

Alle Bytecodes arbeiten auf einem Kellerspeicher, dem sie eventuell nötige Argumente entnehmen und auf dem sie ihre Ergebnisse ablegen. Da der Kellerspeicher der Kontrolle des TSP unterliegt, müssen alle auf ihm befindlichen Speicherworte einem einheitlichen Markierungsschema folgen. Dies bedeutet vor allem bei mathematischen und logischen Operationen, daß die Markierungen von den Argumenten entfernt und dem Resultat hinzugefügt werden müssen.

Bytecode ist eine äußert kompakte Repräsentation von Programmen. Dies schont zum einen Systemressourcen und erlaubt so den Einsatz des Tycoon-Systems auch auf verhältnismäßig gering ausgestatteten Plattformen. Zum anderen wird die Bandbreite in Netzwerkumgebungen beim Versenden von in Ausführung befindlichen Prozessen nicht sehr belastet.

4.3 Stapelspeicher

Der TVM-Stapelspeicher (*stack*) ist eine lineare Datenstruktur aus markierten Speicherworten (siehe Abschnitt 3.1.4). Er ist in zwei Bereiche unterteilt, die von oben und unten aufeinander zuwachsen (Abbildung 4.2):

1. Stapelspeicher für Kontexte der Ausnahmebehandlung (*exception stack*).
   Die Funktionsweise der Ausnahmebehandlung und die Rolle des Ausnahmestapels werden in Abschnitt 4.4 näher erläutert.

2. Stapelspeicher für Ausführungsrahmen (*frames*) von Funktionen.
   Jede in Ausführung befindliche Funktion besitzt einen ihr zugeordenten Speicherbereich innerhalb des Stapels, ihren Ausführungsrahmen. Dieser ist in zwei Teile gegliedert, einen nicht modifizierbaren (*parameter frame*) und einen veränderbaren Bereich (*local frame*). Funktionsargumente, OIDs von Funktionsabschluß und Literalvektor und der Zustandsvektor der aufrufenden Funktion werden im *parameter frame* abgelegt. Der *local frame* speichert veränderliche Werte wie z.B. lokale Variablen.

Mögliche Überschneidungen beider Bereiche können vom Interpreter im Voraus erkannt und durch eine dynamische Vergrößerung des Stapelspeichers verhindert werden.

Die Adressierung des aktuellen Rahmens, Ausnahmepaketes bzw. einzelner Speicherworte erfolgt durch Rahmen-, Ausnahme- und Stapelzeiger.

4.4 Ausnahmebehandlung

Die Ausnahmebehandlung in TL bietet eine einfache Möglichkeit, Laufzeitfehler abzufangen und gezielt zu behandeln. Dieses Konzept wird vom Tycoon-Laufzeitsystem mit Hilfe spezieller Bytecodes und eines Stapelspeichers für Ausnahmekontexte direkt unterstützt. Ein Ausnahmekontext enthält alle für eine Programmfortsetzung im Fehlerfall erforderlichen

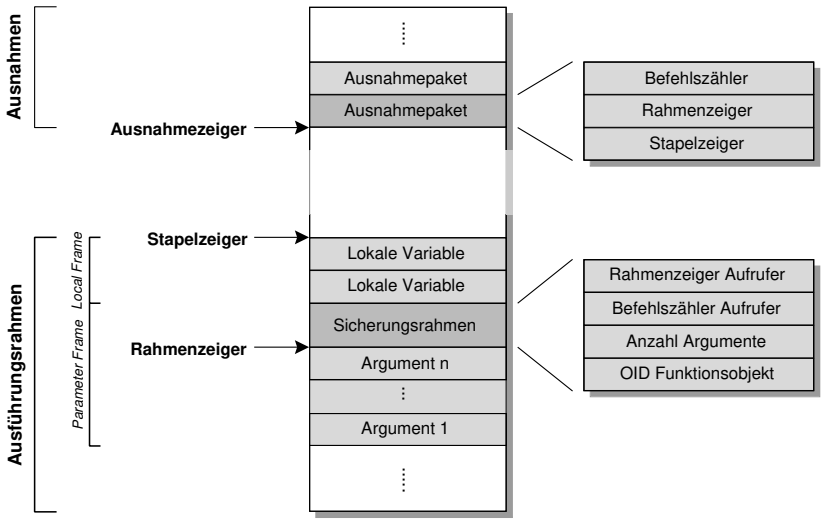

Abbildung 4.2: Stapelspeicher der virtuellen Maschine

Informationen: den Rahmenzeiger der auszuführenden Funktion, ihren Befehlszähler und Stapelzeiger.

Jeder Programmblock, für den eine Ausnahmebehandlung vorgesehen ist, wird von zwei Byteinstruktionen gekapselt:

- *PUSHH* legt zu Beginn des kritischen Abschnitts einen Ausnahmekontext auf dem Stapelspeicher ab.
- *POPH* entfernt den Kontext bei fehlerfreier Programmausführung vom Stapel.

Wird innerhalb des Blocks eine benutzerdefinierte Ausnahme durch den *RAISE*-Opcode ausgelöst, so werden der oberste Ausnahmekontext vom Stapelspeicher genommen, ungültig gewordene Ausführungsrahmen entfernt und die Programmausführung an der im Kontext definierten Stelle fortgesetzt. Auch geschachtelte Ausnahmebehandlungen können durch diesen einfachen Mechanismus gehandhabt werden.

4.5 Parallele leichtgewichtige Prozesse

Im Gegensatz zu einem Prozeß, der den Ausführungsrahmen einer kompletten Applikation definiert, sind leichtgewichtige Prozesse (*leight-weight processes*), im folgenden Threads genannt, Ausführungskontexte einzelner Funktionen. Ein Thread benötigt zur Ausführung lediglich einen eigenen Stapelspeicher und einen kompletten Registersatz (*IP, FP, EP, SP*),

alle weiteren Ressourcen werden mit seinem Erzeugerprozeß geteilt. Die Konstruktion, Destruktion und der Kontextwechsel von Threads sind daher im Vergleich zu Prozessen mit weit geringeren Kosten verbunden. Hinsichtlich Synchronisation und Kommunikation existieren den Prozessen äquivalente Mechanismen.

Das Tycoon-Laufzeitsystem unterstützt die quasi parallele Ausführung (*multitasking*) mehrerer persistenter Threads. Betriebssystemfunktionen wurden für die Implementierung nicht genutzt, da diese sich von Plattform zu Plattform unterscheiden oder teilweise noch nicht einmal vorhanden sind (beispielsweise MacOS). Verwendet werden stattdessen Coroutinen, die in C einfach zu realisieren und auch in anderen Programmiersprachen, z.B. Modula-2, nicht unbekannt sind. Coroutinen stützen sich auf eine kooperative Form des Multitasking, d.h. eine gerade laufende Funktion muß von sich aus den Wechsel des Ausführungskontextes veranlassen. Dieser im ersten Augenblick ersichtliche Nachteil gegenüber präemptivem Multitasking, das Kontextwechsel implizit durchführt und nicht auf die Unterstützung der beteiligten (leichtgewichtigen) Prozesse angewiesen ist, relativiert sich jedoch bei genauerer Betrachtung des im Laufzeitsystem implementierten Verfahrens.

Parallele Ausführungskontexte sind auf Ebene des Bytecodes vollkommen transparent. Es existieren im TVM-Befehlssatz keine speziellen Operationen, die zur Abgabe der Programmsteuerung an einen konkurrierenden Thread führen. Die entsprechenden Mechanismen sind direkt in den Interpreter eingebunden. Ihre Aktivierung erfolgt nach Ablauf einer vom Prozeßmanager (*scheduler*) des Laufzeitsystems zugeteilten Zeitscheibe mit der nächsten auszuführenden Byteinstruktionen zur Kontrollflußsteuerung (u.a. Sprungbefehl, Funktionsaufruf). Da jeder Bytecode den Kontrollfluß betreffende Instruktionen enthält, die in kurzen Abständen ausgeführt werden, können andere Threads nicht blockiert werden. Dieses einfache und doch leistungsfähige Konzept wird nur beim Aufruf externer C-Funktionen durchbrochen, da diese nicht der Kontrolle des Prozeßmanagers unterstehen.

4.6 Persistente Threads

Ein Merkmal des Tycoon-Systems ist seine erweiterte orthogonale Persistenz, die nicht nur bei Daten und Programmen, sondern auch bei Threads zur Anwendung kommt. Der Ausführungskontext eines Threads, d.h. sein Stapelspeicher, Registersatz, Befehlszähler und die OID der aktuellen Funktion, muß daher persistent im Objektspeicher gehalten werden. Der Interpreter operiert aus Geschwindigkeitsgründen auf im Hauptspeicher liegenden Kopien der Kontexte. Ein Abgleich mit dem Objektspeicher wird immer bei Kontextwechseln und vor Aufruf der Freispeicherverwaltung durchgeführt. Es stehen hierzu Funktionen zur Verfügung, die das Auslesen bzw. Schreiben des Ausführungszustandes in den Objektspeicher erlauben.

4.7 Migration

Die plattformunabhängige Repräsentation von Daten und Programmen bildet zusammen mit der auf allen Plattformen verfügbaren gleichen Grundfunktionalität des Tycoon Laufzeitsystems die Basis für die Systemgrenzen überschreitende Migration von Threads. Die Realisierung dieses innovativen Konzeptes erfolgt im wesentlichen auf hochsprachlicher Ebene und über externe Bibliotheken. Es erfordert keine gesonderte Unterstützung durch den Interpreter, z.B. in Form spezieller Bytecodes. Eine ausführliche Beschreibung findet sich in [MMS95].

4.8 C-Calls

TL bietet mit seiner Aufrufschnittstelle für C-Funktionen eine einfache Möglichkeit der Koppelung an externe Diensterbringer. Die hochsprachlichen Konstrukte werden direkt auf einen einfachen, portablen und effizienten Mechanismus des Laufzeitsystems abgebildet, der den Zugriff auf statisch und dynamisch gebundene Bibliotheken steuert.

Der Aufruf einer externen C-Funktion wird durch die Byteinstruktion *CCALL* implementiert, die 3 Argumente erwartet:

- Beschreibung der Argumente und des Ergebnistyps,
- Name der Funktion,
- Name der Bibliothek.

Während die ersten beiden Parameter bereits zur Übersetzungszeit bekannt sind, wird der Bibliotheksname erst zur Laufzeit ermittelt. Die Unabhängigkeit des Bytecodes vom verwendeten Bindungsmechanismus wird auf diese Weise garantiert. Auf Plattformen, die keine dynamische Bindung unterstützen, müssen alle externen Bibliotheken statisch an das Laufzeitsystem gebunden werden.

Die automatische Konvertierung von Argumenten zwischen der TVM und externen Funktionen ist auf unstrukturierte Werte wie Ganzzahlen und Zeichenketten beschränkt. Strukturierte Werte müssen in TL oder C-Funktionen gekapselt werden (*wrapper functions*). Diese übernehmen die Konstruktion zusammengesetzter Werte aus den Basisdatentypen bzw. deren Zerlegung.

4.9 Callbacks

Im Zusammenspiel mit externen Bibliotheken (z.B. grafischen Benutzerschnittstellen) ist es häufig notwendig, Funktionalität des Tycoon-Systems von außen zugänglich zu machen. Die

TVM stellt hierzu einen Mechanismus (*callback*) bereit, der Funktionsobjekte mit dynamisch erzeugtem Code zur Parameterkonvertierung kapselt und als C-Funktionszeiger exportiert.

Callbacks erfordern den direkten Eingriff in die Ablaufsteuerung des Interpreters. Vor Ausführung einer exportierten Funktion müssen ein entsprechender Funktionsrahmen aufgebaut und der aktuelle Ausführungskontext des Interpreters manipuliert werden.

5. Die Maschinencodegenerierung

Effizienz, Portabilität und einfache Skalierbarkeit sind wesentliche Grundpfeiler der Tycoon-Philosophie. Diese Aspekte spiegeln sich auch im Tycoon-Laufzeitsystem wieder.

TVM-Bytecode ist eine kompakte Programmrepräsentation, die durch ihre Plattformunabhängigkeit innovative Konzepte, wie z.B. migrierende, persistente Threads, erst ermöglicht. Seine geringe Komplexität und für Erweiterungen offene Struktur garantiert die einfache Implementierung geeigneter Interpreter bzw. Codegeneratoren. Bytecode, als Instruktionssatz einer virtuellen Maschine und der daraus resultiernden interpretierenden Ausführung, bedingt jedoch im Vergleich zu Maschinencode eine geringere Performanz, da die Ausführung einer Byteinstruktion durch eine Sequenz von 10 – 50 Maschineninstruktionen emuliert werden muß.

Die direkte Übersetzung von TL-Funktionen in Maschinencode scheidet aufgrund schwerwiegender Nachteile aus, obwohl sie zu einer höhere Ablaufgeschwindigkeit führen würde:

- Mangelnde Portabilität.
  Die fehlende Binärkompatibilität zwischen verschiedenen Prozessorfamilien erfordert für jede Systemplattform eigene Codegeneratoren und beschränkt die gemeinsame Nutzung von Daten und Programmen auf homogene Rechnerumgebungen.
- Hohe Komplexität.
  Moderne Prozessoren weisen eine erheblich komplexere Architektur und einen umfangreicheren Befehlssatz als die virtuelle Maschine auf. Für Codegeneratoren besteht ein erhöhter Implementierungsaufwand.
- Großer Speicherbedarf.
  Die Funktionalität einer Byteinstruktion (8 Bit) muß durch eine Sequenz von Maschineninstruktionen (auf RISC-Prozessoren 32, teilweise 64 Bit) realisiert werden.

Die dynamische Maschinencodeerzeugung unter Verwendung bestehender Compilertechnologie stellt adäquate Methoden bereit, die hohe Performanz des Maschinencodes ohne die oben genannten Nachteile zu nutzen.

Dieses Kapitel beschreibt die Implementierung eines dynamischen Maschinencodegenerators für das Tycoon-Laufzeitsystem unter besonderer Beachtung der Erfordernisse, die von

Seiten eines persistenten Systems gestellt werden, u.a. Unterstützung der Freispeicherverwaltung und der Laufzeiteinbindung (Abschnitt 5.3). Zusätzlich werden Möglichkeiten der Programmoptimierung diskutiert (Abschnitt 5.4).

5.1 Integration in die Tycoon-Architektur

Bei der Entwicklung des dynamischen Codegenerators für das Tycoon-System standen zwei Ziele im Vordergrund:

1. Der erzeugte Code soll auch weiterhin alle innovativen Konzepte, u. a. die Migration von Threads, unterstützen.

2. Die Lösung soll möglichst geringe Eingriffe und Änderungen am System erfordern.

Aus diesen Vorgaben heraus muß der Bytecode als plattformunabhängige Programmrepräsentation erhalten bleiben und als Ausgangsbasis für den Maschinencode dienen. Der Codegenerator selbst wird als *just in time* Compiler in das Laufzeitsystem integriert (siehe Abbildung 5.1). Im Gegensatz zu einer statischen Maschinencodeerzeugung im Anschluß an die Übersetzung von TL in Bytecode, die größere Änderungen an Compiler-Backend und Laufzeitsystem erfordern würde, ist diese Art der Codeerzeugung einerseits auch auf im Netz migrierende Threads anwendbar, zum anderen muß nur zur Ausführung kommender Bytecode in Maschinencode übersetzt werden.

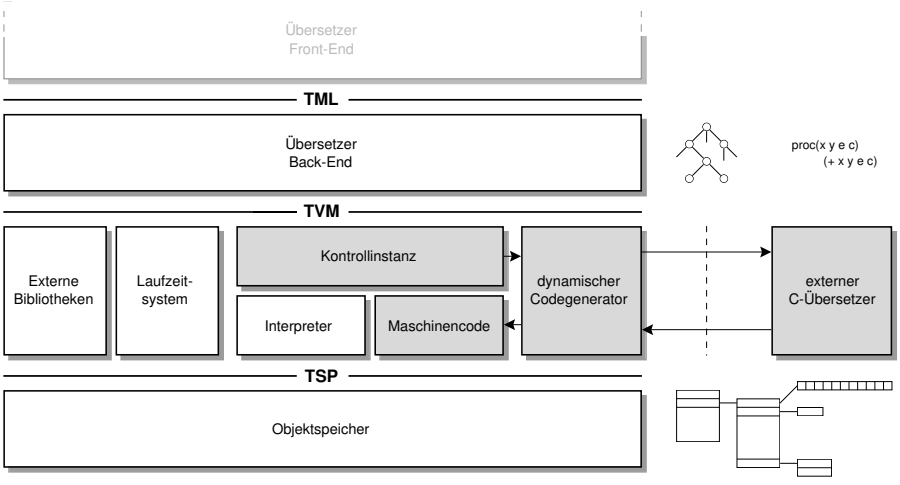

Abbildung 5.1: Tycoon-Schichtarchitektur mit integriertem Maschinencodegenerator

Interpreter und generierter Maschinencode existieren in der modifizierten Architektur der TVM gleichberechtigt nebeneinander. Zugriff und Ablaufkontrolle werden von einer übergeordneten Instanz wahrgenommen, die darüber hinaus den dynamischen Codegenerator steuert. Die Entscheidung, wann eine Funktion interpretiert bzw. in Maschinencode überführt wird, kann von verschiedenen Kriterien abhängig gemacht werden. In der aktuellen Implementierung werden alle Funktionen des TL-Übersetzers von der dynamischen Codeerzeugung ausgeschlossen, d.h. der Übersetzer selbst wird interpretiert und nur für übersetzte Funktionen Maschinencode generiert. Dieses Verhalten sorgt für eine optimale Geschwindigkeit der interaktiven Entwicklungsumgebung, da so lange Wartezeiten beim Systemstart vermieden werden. Es sind jedoch auch andere Mechanismen denkbar, z.B. nur besonders häufig genutzte Funktionen zu übersetzen oder die Codeerzeugung benutzergesteuert an- und auszuschalten.

Der Maschinencode folgt exakt dem TVM-Stapelmodell und führt insbesondere einen kompletten Registersatz, bestehend aus Bytecodebefehlszähler, Stapel-, Rahmen- und Ausnahmezeiger, analog zum Interpreter. Die Ausführungsgrundlage ist für alle innovativen Konzepte des Tycoon Systems, die auf der plattformunabhängigkeit des Bytecodes basieren, vollkommen transparent. Auch der Wechsel zwischen bzw. der gegenseitige Aufruf von Interpreter und Maschinencode ist problemlos zu realisieren.

Maschinencode wird nicht auf direktem Weg erzeugt, der dynamische Codegenerator bedient sich stattdessen der Programmiersprache C und eines externen C-Übersetzers. Die Vorteile, die sich aus der Verwendung von C als portable Zwischensprache ergeben, werden ausführlich in Abschnitt 5.2 diskutiert. Die Codeerzeugung umfaßt die Schritte:

- Bytecode reassemblieren
  Der Bytecode wird in eine für die Optimierung bzw. Codegenerierung besser geeignete Form transformiert. Opcodes und Operanden werden auf ihre volle Größe erweitert und die entsprechenden *PFIX*, *NFIX* und *OPR* Byteoperationen entfernt. Für Sprungbefehle mit relativer Adressierung werden absolute Zieladressen ermittelt. Alle Stellen, an denen die Abgabe des Kontrollflusses möglich ist, werden markiert (siehe hierzu auch Abschnitt 5.3.4).

- Optimierung
  Diese Phase ist optional und im aktuellen System nicht realisiert. In Abschnitt 5.4 wird jedoch ein mögliches Beispiel zur Programmoptimierung vorgestellt.

- C-Quellcode erzeugen
  Die in den Schritten 1 bzw. 2 gewonnene Programmdarstellung wird in C-Quelltext transformiert und auf einem externen Massenspeicher als temporäre Datei abgelegt.

- Maschinencode generieren
  Der C-Übersetzer erzeugt aus der Quelldatei Maschinencode und legt diesen als Datei auf dem Massenspeicher ab.

- Maschinencode einbinden
  Die Mechanismen, die der Einbindung des Maschinencodes dienen, werden in Abschnitt 5.3.1 erläutert.

5.2 Die Wahl von C als portable Zwischensprache

Zwei Alternativen bieten sich für die Generierung von Maschinencode aus Bytecode an:

1. Entwicklung eigener Codegeneratoren
   Sie geben dem Entwickler als größten Vorteil die Kontrolle über die Anbindung an das Laufzeitsystem und alle prozessorinternen Besonderheiten, z.B. die Verwendung von Registervariablen, Zeigerarithmetik und Zugriffe auf Datenstrukturen. Dem entgegen stehen die hohen Entwicklungskosten, da jede Rechnerarchitektur spezifische Codegeneratoren erfordert.

2. Nutzung existierender, plattformübergreifender Werkzeuge
   Sie ermöglichen die Wiederverwendung bereits erprobter Technologie, die von architekturspezifischen Grenzen abstrahiert. Bestehende Codegeneratoren gewinnen vor allem durch die Verfügbarkeit leistungsfähiger Optimierer. Selbst auf niedrigem Niveau operierende Werkzeuge wie Assembler sind in der Lage, einfache Optimierungen vorzunehmen, z.B. längere Befehlssequenzen durch wenige bzw. einzelne äquivalente Instruktionen zu ersetzen (*peephole optimizer*). Andere auf höherem Niveau operierende Werkzeuge, z.B. Compiler, implementieren wesentlich effizientere Optimierungsmechanismen [ASU88].

Der im Vergleich zu einem spezifischen Codegenerator geringe Entwicklungsaufwand und die hohe Portabilität waren die ausschlaggebenden Kriterien, ein bestehendes Werkzeug einzusetzen.

Die Mehrzweckprogrammiersprache C bietet sich aufgrund besonderer Vorteile für die Nutzung als Zwischensprache bei der dynamischen Codeerzeugung an:

- C-Quellcode ist einfach zu generieren.
- C orientiert sich an der heutigen Prozessoren zugrundeliegenden 'von Neumann'-Architektur. Diese starke Hardwarenähe führt zur Erzeugung sehr schnellen Maschinencodes.
- Durch Beschränkung auf die vom American National Standards Institute (ANSI) vorgegebenen Standards können plattformunabhängige Programme geschrieben werden.
- Es existieren leistungsfähige Optimierer.

C ist zudem nicht zuletzt Dank der Free Software Foundation und ihres auf allen gängigen Systemplattformen frei verfügbaren GNU C-Compilers eine der am weitesten verbreiteten Programmiersprachen.

5.3 Lösung spezieller Implementierungsprobleme

Persistente Programmiersprachen stellen besondere Anforderungen bei der Generierung von Maschinencode, vor allem die Unterstützung einer Freispeicherverwaltung, Ausnahmebehandlung, paralleler Threads und der Einbindung externen Programmcodes in ein laufendes System. Dieser Abschnitt zeigt Lösungsmöglichkeiten für die erwähnten Probleme auf.

5.3.1 Einbindung externen Programmcodes zur Laufzeit

Die Nutzung eines externen C-Übersetzers bedingt die Notwendigkeit, den generierten Maschinencode zur Laufzeit in das Tycoon-System einzubinden. Die unter Solaris erfolgte Implementierung verwendet hierzu die vom Betriebssystem bereitgestellte Möglichkeit, dynamisch ladbare Bibliotheken (*shared libraries*) zu verwenden. Dynamische Bibliotheken wurden ursprünglich für den Einsatz in Mehrbenutzerbetriebssystemen konzipiert, da sie in nur einer Laufzeitinstanz ihre Funktionalität mehrenen Programmen zur Verfügung stellen können und so zu erheblichen Speicherplatzeinsparungen führen.

Die Erzeugung einer dynamischen Bibliothek erfordert zwei Schritte: die Generierung einer Objektdatei aus dem C-Quelltext durch den C-Compiler und deren anschließende Transformation in eine dynamische Bibliothek durch den Linker. Die Betriebssystemfunktionen *dlopen* und *dlsym* ermöglichen das Laden einer dynamischen Bibliothek zur Laufzeit bzw. die Ermittlung der Speicheradressen exportierter Funktionen und Datenstrukturen.

Jedem Funktionsobjekt entspricht eine dynamische Bibliothek. Nachdem diese geladen wurde, kann der Funktionszeiger auf den Maschinencode mittels *dlsym* ermittelt, derefenziert und so die Funktion aufgerufen werden.

Da dynamische Bibliotheken auch die Verwendung globaler Variablen gestatten, ergibt sich die einfache Anbindung an die unterstützenden Funktionen des Laufzeitsystems und die Objektspeicherschnittstelle.

Für Systeme, die nicht über den Mechanismus des dynamischen Bindens verfügen, bietet sich die in [BDBV94] vorgestellte Methode an, selbstextrahierenden Code zu verwenden. Der Codegenerator erzeugt hierzu aus jeder Funktion ein Programm, welches sofort ausgeführt wird und seinen Maschinencode in einer temporären Datei ablegt. Diese Datei kann nun vom Laufzeitsystem gelesen und die darin enthaltene Funktion aufgerufen werden. Dieser Ansatz erfordert jedoch zwingend die Generierung vollkommen im Speicher relokatiblen Programmcodes und erlaubt nicht den direkten Zugriff auf globale Objekte, so daß diese z.B. als Funktionsargumente übergeben werden müssen.

5.3.2 Caching von Funktionen

Die Effizienz des Funktionsaufrufes spielt in einer überwiegend funktionalen Sprache wie TL für die Systemperformanz eine entscheidende Rolle. Allein die Laufzeit des externen C-Übersetzers, [EHK96] geht beim GNU C-Compiler von 30000 nötigen Zyklen für die Generierung

einer Maschineninstruktion aus, übertrifft die Ausführungszeiten der erzeugten Funktionen in der Regel bei weitem und würde so, bei mit jedem Funktionsaufruf neu zu generierenden Code, jeglichen durch den Maschinencode erreichten Geschwindigkeitsvorteil zunichte machten. Aus diesem Grund ist es zwingend notwendig, einmal eingebundenen Maschinencode möglichst über die gesamte Laufzeit hinweg im System zu halten und einen Mechanismus zur Verfügung zu stellen, der ein schnelles Auffinden der aufgerufenen Funktionen erlaubt.

Das Laufzeitsystem stellt zu diesem Zweck eine Indextabelle mit 2^{16} Einträgen bereit, in die für jede dynamisch eingebundene Funktion die OID des Literalvektors und die Speicheradresse des geladenen Maschinencodes eingetragen werden. Leere Tabelleneinträge werden als frei markiert. Der Zugriff auf die einzelnen Tabelleneinträge erfolgt über einen 16 Bit großen Index, der aus der veränderlichen OID (siehe auch Abschnitt 5.3.3) des Literalvektors errechnet wird. Dieses in der Literatur als *hashing* bekannte Verfahren [Sed88] verbindet den schnellen Zugriff auf einzelne Elemente mit geringem Speicherplatzverbrauch.

Die effiziente Implementierung des Hashing-Mechanismus ist von der richtigen Wahl des Zugriffsschlüssels abhängig. Die Berechnung eines 16 Bit großen Index aus den Literalvektor-OIDs der gesuchten Funktionen ist sehr schnell – sie erfordert auf Maschinenebene nur eine Registerschiebeoperation mit anschließender Maskierung der relevanten Bits – und liefert nur in Ausnahmefällen für zwei verschiedene Funktionen den gleichen Wert.

Die Speicheradresse des Maschinencodes kann nach erfolgter Indexberechnung und der notwendigen Überprüfung, ob der ermittelte Tabelleneintrag auch von der gesuchten Funktion belegt wird, aus der Tabelle ausgelesen werden. Sollte der entsprechende Eintrag noch frei sein, so wird der dynamische Codegenerator für die aufzurufende Funktion Maschinencode erzeugen, einbinden und die Hash-Tabelle aktualisieren. Im Falle von Konflikten kommt der in [Sed88] diskutierte Auflösungsmechanismus des *linear probing* zum Einsatz.

Dynamisch generierter C-Quellcode und der TVM-Interpreter benutzen für Funktionsaufrufe denselben Mechanismus:

```
literalOID = *FramePointer;           /* OID Literalvektor laden */
functionPointer = lookup(literalOID);/* Funktionszeiger laden */
(*functionPointer)(0);                /* Funktion aufrufen */
```

Die in der übergeordneten Kontrollinstanz (Abschnitt 5.1) lokalisierte Funktion *functionLookup* liefert einen Zeiger auf die aufzurufende Funktion. Bei diesem Zeiger kann es sich sowohl um die Speicheradresse dynamisch gebundenen Maschinencodes als auch um einen Funktionszeiger auf den Interpreter handeln. Der Wechsel zwischen beiden Ausführungsvarianten wird von der Kontrollinstanz gesteuert und ist auf der Codeebene voll transparent.

5.3.3 Freispeicherverwaltung

Wie in Abschnitt 3.1.4 beschrieben, unterstützt der Objektspeicher eine Freispeicherverwaltung, die für die automatische Deallokation nicht mehr erreichbarer Daten- und Funktionsobjekte zuständig ist. Sie ist ein interner Mechanismus des Objektspeichers und auf Ebene

des Bytecodes bzw. des generierten Maschinencodes nicht sichtbar. Dies erfordert auf der einen Seite keinerlei Unterstützung durch das Laufzeitsystem und ermöglicht die Implementierung verschiedener Speicherverwaltungsstrategien, hat aber andererseits erheblichen Einfluß sowohl auf den Interpreter als auch den Maschinencode, da sich Objektreferenzen zur Laufzeit ändern können.

Die Speicherfreigabe und damit verbundene Neuvergabe der OIDs kann prinzipiell mit jeder Allokation eines persistenten Objektes ausgelöst werden. Daher dürfen Datenwerte, die innerhalb einer Funktion mehrmals benötigt werden, nicht in (Register)Variablen zwischengespeichert werden, sondern müssen bei jedem Zugriff erneut aus dem Stapel- bzw. Objektspeicher ausgelesen werden. Betroffen von diesem Umstand ist vor allem die erreichbare Güte des generierten Maschinencodes, da die Reduzierung von Speicherzugriffen durch das Laden von Datenworten in Prozessorregister, eine für C-Übersetzer gängige und effiziente Optimierungstechnik, nicht genutzt werden kann. Hieraus ergeben sich auch für den Interpreter und den erzeugten C-Quelltext Restriktionen im Einsatz von lokalen und globalen Variablen.

Die Freispeicherverwaltung beeinflußt auch die in Abschnitt 5.3.2 beschriebene Indextabelle der dynamisch eingebundenen Maschinencodefunktionen. Mit Aufruf der Speicherfreigabe ändern sich die OIDs der Funktionsobjekte und ihrer einzelnen Komponenten und erfordern so eine Aktualisierung der Tabelle. Die Aufzählungsfunktion stellt nicht nur einen Weg bereit Objekte der Kontrolle der Freispeicherverwaltung zu unterstellen, sie liefert außerdem für bereits bekannte Objekte die neue Referenz in den Objektspeicher zurück. So kann die neue OID für jeden eingetragenen Literalvektor ermittelt und eine neue Indextabelle aufgebaut werden.

5.3.4 Persistente Sicherungspunkte

Die orthogonale Persistenz des Tycoon Systems erstreckt sich auf Daten, Funktionen und Threads. Für Threads ergeben sich aufgrund ihrer dynamischen Struktur besondere Restriktionen, die die dynamische Codegenerierung beeinflussen.

Die Ausführungskontexte jeder innerhalb eines Threads aktivierten Funktion müssen zu bestimmten Zeitpunkten, z.B. dem Auslösen der Freispeicherverwaltung, dem Setzen von Sicherungspunkten oder dem Herunterfahren des Systems, eingefroren und später wieder aktiviert werden. Ohne genaue Kenntnis der zugrundeliegenden Systemarchitektur und Unterstützung durch das Betriebssystem ist dieses Ziel bei Maschinencode nicht zu erreichen, da der Ausführungszustand durch folgende Parameter bestimmt wird: den Prozessorregistern, dem Stapelspeicher und der Speicheraufteilung. Diese Parameter sind für jede Plattform spezifisch und nicht übertragbar.

Der virtuelle Maschine schafft dagegen eine von der Rechnerarchitektur unabhängige Basis für die einheitliche Darstellung des Systemzustandes. Aus diesem Grund führt der generierte Maschinencode den kompletten TVM-Ausführungskontext analog zum Interpreter weiter.

```
    ...
    saveTVMRegisters(8);          /* TVM Register sichern, */
    {                             /* Befehlszähler = 8 */
       Word i;
       Word nElements = Operand;
       tsp_OID oid;

       oid = tsp_newArray(store, tsp_Format_TAGGED, nElements);

       for(i = 0; i < nElements; i++)
          tsp_SET_WORD(store, oid, i, *(StackPointer - i));
       StackPointer -= nElements;
       *(++StackPointer) = oid;
    }
cont8:                            /* Wiedereinstiegspunkt */
    ...
```

Beispiel 5.1: C-Quellcode der Byteinstruktion *ARRAY*

An allen Punkten, an denen der Kontrollfluß eine Funktion verläßt, werden die TVM-Register explizit aktualisiert und Wiedereinstiegspunkte generiert. Dies betrifft:

- Funktionsaufrufe,
- Aufrufe externer C-Funktionen,
- TSP-Operationen, die neue Objekte anlegen, sowie
- Ausnahmen.

Alle entsprechenden Byteinstruktionen werden in der Reassemblierungsphase entsprechend markiert und während der Codeerzeugung von zusätzlichem C-Quellcode umgeben. Dies wird am Beispiel der Byteinstruktion *ARRAY* gezeigt (Beispiel 5.1), die von dem C-Makro *saveTVMRegisters(8)* eingeleitet wird, welches die TVM-Register sichert und den Befehlszähler auf die nächste auszuführende Instruktion (im Beispiel 8) setzt. Eine am Ende generierte Sprungmarke (*cont8*) definiert den Wiedereinstiegspunkt und erlaubt das Wiederaufsetzen bei der folgenden Instruktion.

Die generierten Sprungziele markieren alle Programmpunkte, an denen ein Wiederaufsetzen bzw. Neustart der Funktion erfolgen kann. Da diese Stellen nicht direkt aufgerufen werden können, wird jeder Funktion der Wiedereinstiegspunkt als Argument in Form des TVM-Befehlszählers übergeben, welcher in einem die Funktion einleitenden Prolog ausgewertet wird (Beispiel 5.2).

```
Word generated_X4574(Word contIP)
{
   ...
   if(contIP == 0) {                  //
      ...                             //
      CHECK_STACK_OVERFLOW(14)        //
      goto start;                     //
   }                                  // Prolog
   switch(contIP) {                   //
      case 4:  goto cont4;            //
      case 16: goto cont16;           //
      default: return ILLEGAL_CONT;   //
   }
start:                                // Wiedereinstiegspunkt
   ...
   saveTVMRegisters(4);
   call '=='                          // Funktionsaufruf
cont4:                                // Wiedereinstiegspunkt
   ...
   saveTVMRegisters(16);
   call '+'                           // Funktionsaufruf
cont16:                               // Wiedereinstiegspunkt
   ...
   ret
}
```

Beispiel 5.2: Funktionsprolog

Beim Funktionsaufruf, d.h. bei der Übergabe einer 0 als Argument, wird überprüft, ob der Stapelspeicher noch genügend freie Einträge für die Ausführung der Funktion bereithält. Sollte dies nicht der Fall sein, wird er automatisch vergrößert. Der von jeder Funktion auf dem Stapel benötigte Platz wird bereits statisch während der Generierung des Bytecodes im Compiler-Backend ermittelt und im Literalvektor abgelegt.

Der ohnehin sehr aufwendige Funktionsaufruf wird durch den Prolog weiter verteuert (zu den Auswirkungen siehe Abschnitt 6.1). Die Verwendung der schnelleren Adressarithmetik des GNU C-Übersetzers, wie sie in [BDBV94] demonstriert wird, kann nicht zum Einsatz kommen, da sie nicht plattformunabhängig ist. Sie würde zudem, als spezielle Syntaxerweiterung, die Nutzung beliebiger C-Übersetzer verhindern.

5.3.5 Parallele Threads

Die Abarbeitung paralleler Threads, ist eng mit der Implementierung der persistenten Sicherungspunkte gekoppelt, da die Ausführung von Funktionen an definierter Stelle unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden muß. Der Kontexttransfer erfolgt durch den in Abschnitt 4.5 beschriebenen Mechanismus des Laufzeitsystems. Der Codegenerator erzeugt an ausgewählten Programmpunkten C-Quellcode, der mit Ablauf des festgelegten Zeitintervalls den Ausführungskontext wechselt. Diese Kontextwechselpunkte sind:

- Funktionseintritt
- Sprungbefehle mit Rückwärtsreferenz

Funktionsaufrufe treten während der Ausführung aufgrund des funktionalen Charakters von TL in regelmäßigen, kurzen Abständen auf und eignen sich daher gut für die Auslösung von Kontextwechseln. Um die Blockierung des Prozessmanagers durch Endlosschleifen zu verhindern, werden weitere Kontextwechselpunkte bei Sprungbefehlen mit rückwärts gerichteten Zieladressen eingefügt.

Threads sind im Tycoon System keine isolierten Objekte. Es herrscht ein gewisses Maß an Interaktion, ein Thread kann andere Threads erzeugen, suspendieren, beenden oder an einen anderen Rechner verschicken. Aus diesem Grund ist an den Kontextwechselpunkten das Speichern des TVM-Kontextes und die Generierung von Wiedereinstiegspunkten erforderlich, um die Abarbeitung suspendierter Funktionen wieder aufzunehmen. Kontextwechselpunkte stellen somit nur eine besondere Form der persistenten Sicherungspunkte dar (Beispiel 5.3).

5.3.6 Ausnahmebehandlung

Der dynamisch generierte Maschinencode arbeitet auf dem TVM-Stapelspeicher. Er belegt aber im Gegensatz zum Bytecode auch Speicherplatz auf dem vom Laufzeitsystem benutzten C-Stapelspeicher, auf dem z.B. Rücksprungadressen und Prozessorregister gerettet werden. Durch dieses Verhalten tritt insbesondere bei der Behandlung von TL-Ausnahmen ein Problem auf, da diese den aktuellen Programmfluß unterbrechen und an einer anderen, in den Ausnahmepaketen des TVM-Stapelspeichers spezifizierten Stelle, die nicht innerhalb derselben Funktion liegen muß, fortsetzen. Vom Maschinencode abgelegte Speicherworte können auf dem C-Stapel zurückbleiben und im weiteren Programmablauf zum Absturz des Systems führen. Vor der Bearbeitung einer TL-Ausnahme müssen daher überzählige Speicherworte entfernt und auf dem C-Stapelspeicher ein konsistenter Zustand hergestellt werden.

```
Word generated_X174432(Word contIP)
{
   ...
   if(contIP == 0) {
      if(timer) {              /* Ablauf Zeitscheibe testen */
         saveTVMRegisters(0); /* TVM Register sichern */
         schedule();          /* Kontextwechsel einleiten */
      }
      CHECK_STACK_OVERFLOW(9);
      goto start;
   }
   switch(contIP) {
      case 7:  goto cont7;
      default: return ILLEGAL_CONT;
   }
start:
   ...
cont7:
   ...
   if(condition) goto jump21; // Byteinstruktion CJ (cond. jump)
   ...
   if(timer) {                /* Ablauf Zeitscheibe testen */
      saveTVMRegisters(7);    /* TVM Register sichern */
      schedule();             /* Kontextwechsel einleiten */
   }
   goto cont7;                // Byteinstruktion J (jump)
jump21:
   ...
   ret
}
```

Beispiel 5.3: Unterstützung von Kontextwechseln

Zur Verwirklichung dieser Aufgabe wird die in [Mat92] vorgestellte Ausnahmebehandlung auf C-Ebene verwendet. Sie bedient sich der Betriebssystemfunktionen *setjmp* und *longjmp*.

- *setjmp* sichert den Ausführungskontext eines Programms, d.h. die Prozessorregister und die Umgebung des Stapelspeichers, in eine Datenstruktur.
- *longjmp* restauriert den von *setjmp* gesicherten Kontext und fährt mit der Programmausführung so fort, als ob der Aufruf des vorangegangenen *setjmp* gerade erfolgt wäre. Ein Argument, das an *longjmp* übergeben und als Funktionsergebnis des *setjmp*-Aufrufs durchgereicht wird, ermöglicht die Unterscheidung, ob ein Kontext gesichert oder wiederhergestellt wurde.

Mit diesen beiden Funktionen läßt sich eine vollwertige Ausnahmebehandlung realisieren, von welcher die in der Kontrollinstanz lokalisierte Steuerungsschleife (siehe Abschnitt 5.1) gekapselt wird (Beispiel 5.4).

Die Steuerschleife *globalControl* wird für jeden Thread einmal aufgerufen. Sie überwacht den TVM-Stapelspeicher und stößt, solange dieser nicht leer ist, die Ausführung der an den obersten Funktionsrahmen gebundenen Funktion an. Sie wird von den C-Makros *except_TRY* und *except_END* umschlossen. Das *TRY* Konstrukt sichert den aktuellen Ausführungskontext mittels *setjmp* und ist der Punkt, an dem die Programmausführung nach Auslösung einer Ausnahme wieder aufgenommen wird. Unterschiedliche Ausnahmen werden in den *except_WHEN* Zweigen gezielt behandelt.

Der dynamische Codegenerator erzeugt aus der Byteinstruktion *RAISE*, die eine TL-Ausnahme einleitet, das Makro *except_RAISE(exception_TL)*, welches das System durch *longjmp* auf den gesicherten Kontext zurücksetzt. Die weitere Behandlung übernimmt der entsprechende *except_WHEN* Zweig der Steuerschleife. Hier werden das oberste Paket vom Ausnahmestapel genommen, die TVM-Register neu initialisiert und anschließend zurück in die Schleife gesprungen.

Der Ausnahmemechanismus bietet zudem die Möglichkeit, über das Laufzeitsystem in den Programmablauf einzugreifen. Es existieren Hilfsfunktionen, die Ausführungsrahmen auf dem TVM-Stapel anlegen, die TVM-Register manipulieren und die entsprechenden Funktionen durch Auslösung einer Ausnahme *exception_ESCAPE* starten. Genutzt wird diese Funktionalität im TL-Übersetzer, der auf dem Tycoon-Toplevel eingegebene Ausdrücke direkt nach ihrer Übersetzung in Bytecode ausführt.

5.4 Mögliche Optimierungen des Codegenerators

Die TVM ist eine virtuelle Maschine, deren Instruktionen über einem Stapelspeicher operieren. Diese Arbeitsweise orientiert sich nicht an der tatsächlich zugrundeliegenden Prozessorarchitektur und bietet daher Raum für Optimierungen. Eine einfache Optimierungsmöglichkeit ist die Eliminierung unnötiger Stapelzugriffe. Sie wird in diesem Abschnitt an einem kleinen Beispiel schematisch dargestellt.

```
tsp_Tagged globalControl(void)
{
   tsp_OID oidCode;
   Word ipCode, rValue;
   cfun pFunction;
   ...
   loadTVMRegisters();                    /* Register aus Objekt- */
   ...                                    /* speicher laden */
   except_TRY
      while(TRUE) {
         ...
         oidCode = activeFunction();      /* aktive Funktion laden */
         ipCode = IP;                     /* Befehlszähler laden */
         ...
again:
         pFunction = lookup(oidCode);     /* Funktionszeiger laden */
         rValue = (*pFunction)(ipCode); /* Funktion aufrufen */
         if(rValue == ILLEGAL_CONT)
            abort();
         ...
      }
   except_WHEN(exception_TL) {
      ...
      SP = *(EP++);                       // Programmfortsetzung aus
      FP = *(EP++);                       // dem Ausnahmepaket
      IP = *(EP++);                       // ermitteln
      ...
      oidCode = activeFunction();         /* aktive Funktion laden */
      ipCode = IP;                        /* Befehlszähler laden */
      goto again;
   }
   except_WHEN(exception_ESCAPE) {
      oidCode = activeFunction();
      ipCode = IP = 0;
      goto again;
   }
   except_END
   ...
}
```

Beispiel 5.4: Steuerschleife des dynamischen Codegenerators

Die folgenden Zeilen zeigen die TVM-Assemblerdarstellung der TL Anweisung $\{x + 3\} * 2$.

```
ldg x        // globale Konstante x auf den Stapel laden
ldc 3        // Ganzzahl 3 auf den Stapel laden
add          // Werte addieren und Ergebnis ablegen
ldc 2        // Ganzzahl 2 auf den Stapel laden
mul          // multipizieren
```

Die einfachste Möglichkeit der Transformation in C-Quelltext ist die Erzeugung zu den Byteinstruktionen äquivalenter C-Makros.

```
push(tsp_get(x));          // push(value) Wert auf Stapel legen
push(3);                   // pop() Wert von Stapel laden
push(pop()+pop());         // tsp_get(OID) Objekt aus Store laden
push(2);
push(pop()*pop());
```

Diese erste Variante benötigt zwar im Gegensatz zum Interpreter keine Befehlsdekodierung, enthält jedoch noch einige überflüssige und, da auf den Hauptspeicher zugreifend, kostenträchtige Stapelzugriffe. Die Entfernung unnötiger Speicherzugriffe und Zeigerarithmetik führt zu folgendem Code:

```
push(tsp_get(x)+3);
push(pop()*2);
```

In einem letzten Schritt können die primitiven Operationen '+' und '*' zusammengefaßt werden.

```
push((tsp_get(x)+3)*2);
```

Dieses mit einem geringen Aufwand zu implementierende Verfahren ist nicht nur in der Lage, die Performanz zu erhöhen, es reduziert auch die Größe des generierten Maschinencodes.

Voraussetzung für den erfolgreichen Einsatz dieser Optimierungsstrategie ist eine vorher im Compiler-Backend erfolgte Optimierung des Bytecodes. Arithmetische und logische Operationen werden vom TL-Übersetzer nicht direkt in die entsprechenden Byteinstruktionen übersetzt sondern über Basisfunktionen, welche die Instruktionen kapseln, angesprochen. Diese Funktionsaufrufe stellen eine Barriere für die Optimierung dar und müssen durch die Funktionsexpansion (*inlining*) des dynamischen TML-Optimierers entfernt werden.

Weitergehende Optimierungen, wie in [LL95; KEH91] beschrieben, die den späten Übersetzungszeitpunkt und die damit verbundenen zusätzlich verfügbaren Informationen, z.B. Funktionsargumente, ausnutzen, können nur eingeschränkt erfolgen. Während des Übersetzungsprozesses von TL über TML hin zu Bytecode gehen zu viele semantische Informationen, vor allem die dringend erforderlichen Typinformationen, verloren.

6. Das Laufzeitverhalten

Nachdem in Kapitel 5 die Technik der dynamischen Maschinencodegenerierung vorgestellt wurde, erfolgt in diesem Kapitel die Betrachtung des Laufzeitverhaltens. Neben dem Vergleich der Ausführungsgeschwindigkeit von Interpreter und generiertem Maschinencode, werden auch der Aufwand der Codeerzeugung und die Veränderung der Codegröße bewertet.

6.1 Performanz

Für die Messungen wurden, wie bereits in [Kir94], zwei Standardtestprogramme verwendet, die *Standford Suite* und der *Richards' Benchmark*.

Richards' ist ein größeres Programm, das das Szenario eines Betriebssystems simuliert. Die *Stanford Suite* ist eine Sammlung einfacher Programme, die auf verschiedenen Sprachkonstrukten wie rekursiven Funktionen (*Perm*, *Towers*, *Queens*, *Tree*) sowie Schleifen und Feldbearbeitung (*Puzzle*, *FFT*) aufbauen:

Perm: Berechnung aller Permutationen über ein Ganzzahlenfeld mit 10 Einträgen.

Towers: Lösung des Problems der Türme von Hanoi.

Queens: 50 maliges Lösen des 8 Damen Problems.

Intmm: Multiplikation zweier 40x40 Ganzahlmatrizen.

Mm: Multiplikation zweier 40x40 Fließkommamatrizen.

Quick: Sortierung von 5000 Ganzzahlen mit dem Quicksort-Algorithmus.

Bubble: Sortierung von 5000 Ganzzahlen mit dem Bubblesort-Algorithmus.

Trees: Sortierung von 5000 Ganzzahlen mit dem Treesort-Algorithmus.

FFT: Fast Fourrier Transformation.

Puzzle: Iterative Feldbearbeitung.

Programm	Interpreter (s)	dyn. Maschinencode ohne Opt. (s)	dyn. Maschinencode mit Opt. (s)	$\frac{M}{M_{opt}}$	$\frac{Int}{M_{opt}}$
Perm	3.04 s	1.44 s	0.95 s	1.52	3.20
Towers	4.35 s	2.13 s	1.49 s	1.43	2.92
Queens	5.75 s	2.77 s	1.76 s	1.57	3.27
Intmm	3.55 s	1.30 s	0.86 s	1.51	4.13
Mm	5.37 s	2.86 s	2.24 s	1.28	2.40
Quick	4.67 s	1.91 s	1.38 s	1.38	3.38
Bubble	11.43 s	4.87 s	3.12 s	1.56	3.66
Trees	2.65 s	1.63 s	1.15 s	1.42	2.30
FFT	12.75 s	7.11 s	5.34 s	1.33	2.39
Puzzle	19.65 s	4.08 s	3.23 s	1.26	6.08
Richards'	23.58 s	19.14 s	13.69 s	1.40	1.72
Durchschnitt				1.42	3.22

Tabelle 6.1: Performanzmessungen an den Standardtestprogrammen der *Stanford Suite* und am *Richards' Benchmark*

Alle Messungen wurden auf einem Sun SPARCserver 1000 durchgeführt, wobei auf eine geringe Systemauslastung geachtet wurde, um die Ergebnisse nicht zu verfälschen. Der TVM-Interpreter wurde mit dem GNU C-Compiler bei maximaler Optimierung (-O2) übersetzt, der generierte Code sowohl mit als auch ohne Optimierungen. Zur Zeitmessung diente die Funktion *times* aus der Standardbibliothek des Betriebsystems, welche die von einem Prozess verbrauchte CPU-Zeit ermittelt.

Als Objektspeicher kam in allen Fällen die auf dem Haupspeicher operierende und damit schnellste Variante *tymem* zum Einsatz, um den Einfluß der persistenten Datenhaltung auf die Messergebnisse möglichst gering zu halten.

Die Laufzeiten des interpretierten Bytecodes und des generierten Maschinencodes werden in Tabelle 6.1 gegenübergestellt. Es sind die besten Ergebnisse in einer Reihe von mindestests drei Messungen.

Klar zu erkennen ist die große Schwankung beim erreichbaren Geschwindigkeitszuwachs. Die Spanne reicht in der optimierten Version von 1,7 (*Richards*) bis hin zu 6 (*Puzzle*) mal schnellerer Ausführung. Bedingt ist dies durch die in den einzelnen Testprogrammen verwendeten

Sprachkonstrukte. Imperativ aufgebaute Programme, die Scheifen und Zustandsvariablen benutzen (z.B. *Puzzle*, *Intmm*), weisen eine deutlich höhere Performanzsteigerung auf als Programme mit funktionaler Struktur (*Quick*, *Trees*), deren überwiegendes Konstrukt die Rekursion ist. Haupverantwortlich hierfür ist das mit jedem TL-Funktionsaufruf notwendige und aufwendige Anlegen eines Ausführungsrahmens bzw. dessen Abbau mit Beendigung der Funktion. Auch der je nach Testprogramm unterschiedliche Anteil an Objektspeicherzugriffen hat einen Einfluß auf die Messwerte (siehe Abschnitt 6.2). Der Grund für das vergleichsweise schlechte Abschneiden der *Fast Fourrier Transformation* (*FFT*) und Matrixmultiplikation (*Mm*) ist in der fehlenden Unterstützung von Fließkommazahlen durch den TVM-Befehlssatz begründet. Fließkommaarithmetik muß über die langsame C-Aufrufschnittstelle abgewickelt werden.

Der generierte Maschinencode profitiert deutlich vom effizienten Optimierer des C-Übersetzers. Der Performanzgewinn liegt bei bis zu 50%, im Gegenzug steigen allerdings die Übersetzungszeiten zum Teil erheblich an (siehe Abschnitt 6.3).

Die benutzten Testprogramme wurden ursprünglich für konventionelle Hochsprachen (z.B. C) entwickelt und es stellt sich die Frage, inwieweit sich die Ergebnisse auf eine funktionale Sprache wie TL und für persistente Systeme typische Applikationen übertragen lassen. Um zu einer groben Abschätzung zu kommen, wurden in einem Versuch das gesamte in TL implementierte Compiler-Front- und Backend in die dynamische Codegenerierung mit einbezogen (siehe Abschnitt 7.1) und die Übersetzungszeiten für kleine TL-Ausdrücke gemessen. Der Geschwindigkeitszuwachs liegt mit ca. 75% bis 100% etwa im Bereich des *Richards' Benchmark*.

Die Auswirkungen des dynamischen TML-Optimierers [Kir94], vor allem der Funktionsexpansion, auf die Güte des generierten Maschinencodes konnten aufgrund von Fehlern im Zusammenspiel des dynamischen Optimierers mit dem restlichen Compiler Back-End (siehe Abschnitt 7.1) nicht genauer untersucht werden. Entsprechend von Hand optimierter Bytecode zeigte sowohl in der Ausführung durch den Interpreter als auch im Maschinencode eine um den Faktor 1,8 erhöhte Performanz. Die in Abschnitt 5.4 vorgestellte Optimierungsmöglichkeit für den Codegenerator ist sicherlich dazu geeignet, diese Spanne weiter zugunsten des Maschinencodes zu verschieben.

6.2 Einfluß der Persistenz

Die Notwendigkeit zur persistenten Datenhaltung schafft mit dem Objektspeicher und der TSP-Schnittstelle einen die Geschwindigkeit von Byte- und Maschinencode limitierenden Faktor. Zur Ermittlung des Einfluß von Objektspeicherzugriffen auf die gesamte Ausführungszeit eines Programmlaufs wurden die in der Entwicklungsumgebung von Solaris [Sun94] bereitgestellten Werkzeuge Profiler und Analyzer benutzt. Der Profiler überwacht den Ablauf eines Programms und untersucht in kurzen Intervallen (im Bereich weniger ms) dessen Ausführungszustand, z.B. Programmzähler und Stapelspeicher. Die gewonnenen Daten

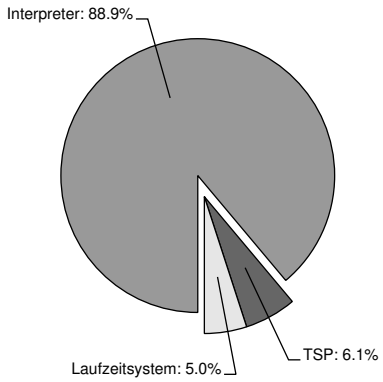

Abbildung 6.1: Profilerlauf für das Testprogramm *Queens*

können mit dem Analyzer nach verschiedenen Kriterien ausgewertet werden, u.a. läßt sich die Verweildauer für jede Funktion einzeln bestimmen.

Die Ergebnisse eines Interpreterlaufs für das Testprogramm *Queens* werden in den Abbildungen 6.1 und 6.2 dargestellt. Wie zu erkennen ist, nehmen die Objektspeicherzugriffe etwa 6% der Ausführungszeit in Anspruch. Hinzu kommen 5% von den Funktionen des Laufzeitsystems. Da der Maschinencode jedoch nur den Anteil des Interpreters ersetzt, ist ein Geschwindigkeitszuwachs auf mehr als das 10 fache nicht zu realisieren.

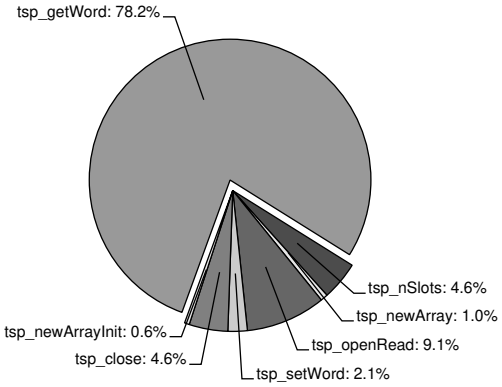

Abbildung 6.2: Verteilung des Objektspeicherzugriffe für das Testprogramm *Queens*

Programm	TL-Übersetzer (s)	C-Übersetzer ohne Opt. (s)	C-Übersetzer mit Opt. (s)	$\frac{C_{opt}}{C}$
Perm	3.49 s	7.30 s	14.90 s	2.04
Towers	8.91 s	15.32 s	38.14 s	2.49
Queens	5.00 s	10.94 s	26.91 s	2.46
Intmm	3.62 s	11.43 s	23.74 s	2.08
Mm	3.68 s	15.81 s	30.80 s	1.95
Quick	5.26 s	15.26 s	34.35 s	2.25
Bubble	4.18 s	13.12 s	30.82 s	2.35
Trees	6.24 s	16.05 s	32.10 s	2.00
FFT	20.29 s	30.59 s	124.97 s	4.09
Puzzle	33.24 s	41.03 s	322.34 s	7.86
Richards'	40.95 s	63.71 s	99.43 s	1.56
Durchschnitt				2.83

Tabelle 6.2: Übersetzungszeiten der Standardtestprogramme *Stanford Suite* und *Richards' Benchmark*

Eine Reduktion der überwiegend lesenden Zugriffe (*tsp_getWord*), z.B. durch die Zwischenspeicherung bereits gelesener Daten in lokalen Variablen, kann nur innerhalb kurzer Programmabschnitte (*basic blocks*) stattfinden. Da kein Mechanismus existiert, um die Auslösung der Freispeicherverwaltung zu überwachen, besteht mit Abgabe des Kontrollflusses die Gefahr, daß lokal gesicherte Werte (OIDs) ihre Gültigkeit verlieren.

6.3 Aufwand der Codegenerierung

Neben der Zunahme an Performanz ist der Generierungsaufwand ein für die Beurteilung der Effektivität des dynamischen Codegenerators entscheidendes Kriterium. In Tabelle 6.2 werden die Laufzeiten des C-Übersetzers für die einzelnen Testprogramme ohne und mit Optimierungen denen des TL-Übersetzers gegenübergestellt. Die Messungen erfolgten mit dem Unix Kommando *time* bzw. der Bibliotheksfunktion *times*.

Im Schnitt verdoppeln sich die Übersetzungszeiten mit zugeschalteten Optimierungen, teilweise liegen sie sogar noch höher (*Puzzle*). Die Zahl der einzelnen Funktionen und damit

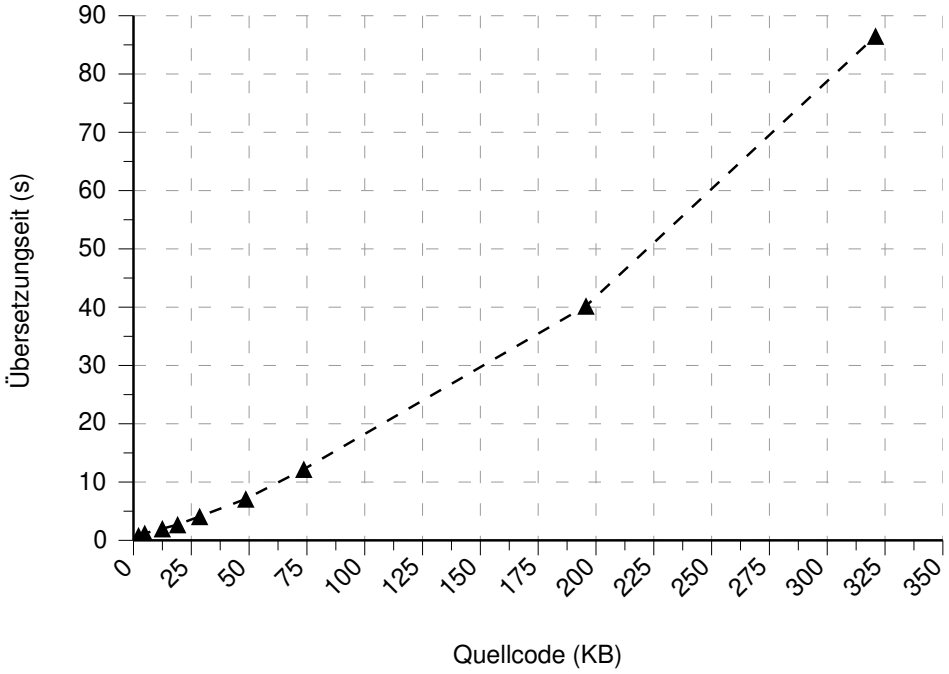

Abbildung 6.3: Anstieg der Übersetzungszeiten mit steigender Quellcodegröße

der Overhead, der durch das Starten des C-Übersetzers entsteht, spielt hierbei eine untergeordnete Rolle. Vielmehr steigen die Zeiten mit der Größe des Quelltextes nicht linear an (Abbildung 6.3). Die Dauer für die Erzeugung des Quelltextes liegt im Bereich der Meßtoleranz.

Der in Abschnitt 5.3.2 beschriebene Laufzeitcache für generierten Code trägt wesentlich dazu bei, die Auswirkungen des externen C-Übersetzers auf die Systemperformanz zu minimieren, da Bytecodefunktionen während eines Systemlaufs in der Regel nur ein einziges Mal in Maschinencode transformiert werden müssen.

6.4 Entwicklung der Codegröße

Wie bereits aus den in [BDBV94] gemachten Erfahrungen zu erwarten, ist Maschinencode gegenüber dem kompakten Bytecode mit einer drastischen Erhöhung des Speicherbedarfs verbunden.

Die in Tabelle 6.3 angegebenen Werte beziehen sich auf die Gesamtgröße der für die Testprogramme erzeugten Bibliotheken. Diese beinhalten neben dem eigentlichen Programmcode

Programm	Funktionen	Bytecode (Byte)	Maschinencode (Byte)	$\frac{M}{B}$
Perm	7	378	37704	99.75
Towers	13	1009	77752	77.06
Queens	11	719	61144	85.04
Intmm	12	621	61416	98.90
Mm	16	732	83520	114.10
Quick	15	991	77864	78.57
Bubble	13	823	70488	85.65
Trees	17	841	85816	102.04
FFT	19	2557	147016	57.50
Puzzle	12	4810	184560	38.37
Stanford	55	9817	607428	61.88
Richards'	53	3323	434532	130.76
Durchschnitt				85.80

Tabelle 6.3: Größenvergleich zwischen Bytecode und dynamischem Maschinencode

auch Symbolinformation für den dynamischen Binder, deren Anteil bei kleinen Bibliotheken (<= 10 KB) mit etwa 60 – 70%, bei großen Bibliotheken (> 100 KB) mit etwa 30% anzusetzen ist.

Die Diskrepanz der Werte aus den Einzelsummen bzw. dem Gesamtpaket der *Stanford Suite* ist in der Eigenschaft des TL-Übersetzers begründet, mathematische und logische Operationen nicht mit festen TVM-Byteinstruktionen zu verbinden, sondern in Basisfunktionen zu kapseln. In den einzelnen Testprogrammen beträgt der Anteil dieser Funktionen bis zu 50% der gesamten Dateigröße, im Paket fällt er erheblich niedriger aus.

Es existieren verschiedene Methoden, den Speicherverbrauch zu verringern:

- Die in Abschnitt 5.4 vorgestellte Optimierung erhöht nicht nur die Performanz, sondern reduziert gleichzeitig die Codegröße.
- Die Funktionsexpansion ersetzt einen aufwendigen Funktionsaufruf durch ein Primitiv.
- Voluminöse oft genutzte Programmblöcke können in neue Funktionen des Laufzeitsystems verlagert werden.

Eine Reduktion der Codegröße ist auch aus Geschwindigkeitsgründen wünschenswert. Moderne Prozessorfamilien (PowerPC, Intel i86, SPARC, DEC Alpha etc.) besitzen interne, schnelle Cachespeicher, die einen Großteil der Speicherzugriffe abfangen können und so den Prozessor vom langsamen Haupspeicher entkoppeln. Aufgrund der geringen Größe dieser Zwischenspeicher (16 – 64 KB) wird ihre Effizienz, neben der Anzahl und Verteilung von Datenzugriffen, maßgeblich von der Programmgröße bestimmt.

7. Zusammenfassung

Abschließend werden in den beiden folgenden Abschnitten der Stand der Implementierung vorgestellt und die in den vorangegangenen Kapiteln beschriebene Technik der dynamischen Maschinencodeerzeugung und das Laufzeitverhalten bewertet sowie ein Ausblick auf die weitere Entwicklung gegeben.

7.1 Stand der Implementierung

Ein Prototyp des dynamischen Codegenerators wurde auf SunSPARC Workstations unter den Betriebssystemen SunOS 4 und Solaris 2.x implementiert. Die Entwicklung wurde allerdings durch Probleme erschwert, deren Ursachen teilweise außerhalb des Codegenerators liegen und die noch nicht befriedigend gelöst werden konnten:

- Mit zunehmender Anzahl dynamisch gebundener Maschinencodefunktionen wird das Laufzeitverhalten des Tycoon-Systems instabil. Dies äußert sich in Systemabstürzen, die durch das Überschreiben wichtiger Speicherbereiche ausgelöst werden. Als Ursache konnte das Zusammenspiel mit dem dynamischen Linker des Betriebssystems lokalisiert werden. Ob es sich hierbei um eine prinzipielle Schwäche oder fehlerhafte Implementierung des Linkers handelt oder die Probleme auf Wechselwirkungen mit dem Laufzeitsystem, eventuell mit der Ausnahmebehandlung auf C-Ebene (Abschnitt 5.3.6), zurückzuführen sind, konnte nicht festgestellt werden.
  Umfangreiche Testmessungen zur Abschätzung des zu erzielenden Geschwindigkeitszuwachses bei großen persistenten Applikationen (Abschnitt 6.1) konnten aus diesem Grund nicht bzw. im Fall des TL-Übersetzers nur in Ansätzen durchgeführt werden.

- Die hohen Laufzeiten des externen C-Übersetzers wirken sich direkt auf die Entwicklungsarbeit aus. Testläufe zur Fehlerfindung bzw. -bereinigung, insbesondere in Verbindung mit den Instabilitäten des Systems, nahmen mit einer Dauer von jeweils 1 bis 1,5 Stunden über die Hälfte der Erstellungszeit des Prototypen in Anspruch.

- Effizienter Maschinencode durch Optimierungen des Codegenerators in der Transformationsphase von Bytecode zu C-Quellcode setzt in hohem Maß bereits optimierten Bytecode voraus. Als wesentlicher Faktor erweist sich hierbei das *inlining* von Funktionen durch den dynamischen Optimierer des Compiler-Backends. Da dessen Funktionalität seit einer Neustrukturierung des Backends jedoch stark eingeschränkt ist, mußte von einem verbesserten Codegenerator Abstand genommen werden.

7.2 Bewertung und Ausblick

Das persistente und modulare Tycoon-System ist eine persistente Programmierumgebung für datenintensive Applikationen. Sein portables und plattformunabhängiges Laufzeitsystem ermöglicht den Einsatz auf verschiedenen Rechnerarchitekturen, bedingt durch die interpretierende Ausführung von Programmen jedoch eine im Vergleich zu Maschinencode geringe Performanz. Die dynamische Maschinencodeerzeugung kann die Geschwindigkeit des Tycoon-Systems steigern, ohne dabei an Portabilität und Flexibilität zu verlieren. Sie weist dabei gegenüber einem statischen Codegenerator folgende Vorteile auf:

- Ihre Integration in das Gesamtsystem erfordert nur geringe Modifikationen am Laufzeitsystem.
- Da sie nur eine Ergänzung, aber keinen Ersatz des Interpreters darstellt, ist ihre Realisierung für jede Plattform optional. Einzelne Implementierungen können auch unabhängig voneinander erfolgen.
- Der Einsatz eines dynamischen Codegenerators ist auf höheren Systemebenen transparent.
- Maschinencode wird nur für Funktionen erzeugt, die auch zur Ausführung gelangen. Dies führt, da seine Lebensdauer zudem auf einen Systemlauf beschränkt ist, im Gegensatz zu statisch erzeugtem Code, der persistent im Objektspeicher gehalten werden muß, zu einer deutlichen Reduktion des Speicherverbrauchs.

Durch die Entscheidung, TVM-Bytecode als Ausgangsbasis für die Codegenerierung zu verwenden und mit dem Maschinencode dem Ausführungsmodell der virtuellen Maschine zu folgen, muß nicht auf die innovativen Konzepte, die auf der Plattformunabhängigkeit des Tycoon-Systems beruhen, wie z.B. die Migration persistenter Threads, verzichtet werden. Die besonderen Anforderungen, die von einer persistenten Umgebung gestellt werden, insbesondere die Bindung von dynamisch generiertem Code an ein laufendes System, die Unterstützung von persistenten Sicherungspunkten und einer Freispeicherverwaltung, lassen sich, wie in Abschnitt 5.3 beschrieben, mit geringem Aufwand lösen.

Die Programmiersprache C als Zwischenrepräsentation unter Verwendung eines externen C-Übersetzers erweist sich für die Portabilität des Systems als vorteilhaft. Auch die Güte

des erzeugten Maschinencodes wird durch Optimierer auf Seiten des Übersetzers deutlich verbessert.

Der dynamisch generierte Maschinencode weist selbst in dem in dieser Arbeit vorgestellten einfachen Ansatz einen Geschwindigkeitsvorteil gegenüber der Ausführung von Bytecode durch den TVM-Interpreter auf. Für die Beurteilung der Effizienz des dynamischen Codegenerators ist jedoch neben der Geschwindigkeitssteigerung durch den Maschinencode die Performanz des Codegenerators selbst von Bedeutung. Hier erweist sich die gewählte Lösung als problematisch, da der verursachte Performanzverlust aufgrund der Übersetzungszeiten erst bei einer entsprechend langen Systemlaufzeit durch den Maschinencode wieder aufgeholt und ein effektiver Geschwindigkeitsvorteil erreicht wird.

Für die Zukunft, auch im kommerziellen Umfeld von Tycoon, ist die Implementierung von speziellen, der jeweiligen Rechnerplattform angepaßten dynamischen Codegeneratoren vorgesehen. Neben vollständigen Eigenentwicklungen könnten hier auch wieder bestehende, schnelle Werkzeuge [EP94] verwendet werden.

Literaturverzeichnis

[ASU88] A. Aho, R. Sethi and J. Ullman. *Compilerbau.* Addison-Wesley, 1988.

[BDBV94] S.J. Bushell, A. Dearle, A.L. Brown and F.A. Vaughan. *Using C as a Compiler Target Language for Native Code Generation in Persistent Systems.* In: *Proceedings of the Sixth International Workshop on Persistent Object Systems*, Tarascon, Provence, France, September 1994.

[Car90] L. Cardelli. *The Quest Language and System (Tracking Draft).* Digital Systems Research Center, DEC SRC Palo Alto, 1990. shipped as part of the Quest V.12 system distribution.

[DCBM89] A. Dearle, R. Connor, F. Brown and R. Morisson. *Napier88 - A Database Programming Language?* In: *Proceedings of the Second International Workshop on Database Programming Languages*, Salishan, Oregon, June 1989.

[DS84] P. Deutsch and A.M. Schiffman. *Efficient Implementation of the Smalltalk-80 System.* In: *11th Annual Symposium on Principles on Programming Languages*, January 1984.

[EHK96] D. Engler, W. Hsieh and M. Kaashoek. *C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation.* In: *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[EP94] D. Engler and T. Proebsting. *DCG: An Efficient, Retargetable Dynamic Code Generator.* In: *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[GM95] J. Gosling and H. McGilton. *The Java Language Environment.* Vorlesungsskript, Sun Microsystems, 1995.

[KEH91] D. Keppel, S.J. Eggers and R.R. Henry. *A Case for Runtime Code Generation.* Technical report, Department of Computer Science and Enginnering, University of Washington, 1991.

[Kir94] P. Kiradjiev. *Dynamische Optimierung in CPS-orientierten Zwischensprachen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

[LL95] M. Leone and P. Lee. *Optimizing ML with Run-Time Code Generation.* Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[Mat92] B. Mathiske. *Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

[MMS95] B. Mathiske, F. Matthes and J.W. Schmidt. *On Migrating Threads.* In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, June 1995.

[MSS96] F. Matthes, G. Schröder and J.W. Schmidt. *Tycoon: A Scalable and Interoperable Persistent System Environment.* Technical report, Hamburg University, 1996.

[Mül91] R. Müller. *Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung.* Diplomarbeit, J.W. Goethe-Universität, Frankfurt/Main, 1991.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

[SM92] J.W. Schmidt and F. Matthes. *The Database Programming Language DBPL - Rational an Report.* Technical report, Hamburg University, 1992.

[Sun94] Sun Microsystems. *SPARCworks/ProWorks 3.0.1 Performance Tuning an Application*, 1994.