Migration von .NET-Programmen

André Seidelt
Matrikelnummer: 183234

18. Juli 2005

Zusammenfassung

Migration von nebenläufigen Applikationen durch Modifikation einer existierenden Laufzeitumgebung für .NET

Häufig gewünschte Eigenschaften von Anwendungen sind Verteilbarkeit und Ausfallsicherheit. Hierdurch sollen die verfügbaren Hardwareressourcen besser ausgenutzt werden. Dieses Ziel kann erreicht werden, wenn es Programmen ermöglicht wird, während der Laufzeit auf einen anderen Rechner zu migrieren und dort nahtlos die Ausführung fortzusetzen. Auch im Zusammenhang mit Agentensystemen wird versucht, durch mobile Agenten die Aufgabe zu den Daten wandern zu lassen und nicht, wie bisher üblich, die Daten zur Aufgabe.

Die vorliegende Arbeit untersucht die Möglichkeit der transparenten Anwendungsmigration auf Microsofts .NET Plattform, stellt eine Implementation vor, und bewertet die durch die Untersuchung der Implementation gewonnenen Ergebnisse.

Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 18. Juli 2005

....
(Andre Seidelt)

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Gründe für Anwendungsmigration 1
1.2 Aufgabenstellung 2
1.3 Das .NET Framework 3
1.4 Existierende Lösungsansätze 4
1.4.1 Entwicklungskontrollierte Migration 4
1.4.2 Übersetzungskontrollierte Migration 4
1.4.3 Laufzeitkontrollierte Migration 5
1.5 Problembereiche 5
1.6 Struktur der Arbeit 6

2 Die virtuelle Maschine **9**

2.1 Einführung 9
2.2 Die VM des .NET Frameworks 10
2.3 Auswahl der VM 14
2.3.1 Die MS-Rotor VM 15
2.3.2 Die Mono-VM 15
2.3.3 Die PNet VM 15
2.3.4 Zusammenfassung der Auswahlkriterien 17
2.4 Interner Aufbau der PNet VM 18
2.4.1 Übersetzungsstadien 18
2.4.2 Interpretermodi 20
2.4.3 Threading 23

3 Migration **25**

3.1 Problemraum 26
3.2 Vorstellung und Bewertung der Lösungsansätze 30
3.3 Vorstellung der ausgewählten Lösungen 37

4 Implementation **39**

4.1 Veränderungen an der VM 39
4.2 Datenstrukturen des Abbildes 42
4.3 Veränderungen an der Klassenbibliothek 45

5 Bewertung der Implementation **51**
5.1 Testszenario . . . 51
5.2 Overhead zur Laufzeit . . . 52
5.3 Messung des Migrationsvorganges . . . 57

6 Fazit **59**

A Literaturverzeichnis **60**

B Glossar **63**

C Benchmarks **66**
C.1 TakThreads . . . 66
C.2 TakMonitor . . . 67

D Übersicht über die CD-ROM **68**
D.1 Verzeichnisinhalte . . . 68
D.2 Übersetzen der VM . . . 68
D.3 Testfälle . . . 68
D.4 Nachvollziehen der Änderungen . . . 70

Abbildungsverzeichnis

1 Probleme der Anwendungsmigration 7
2 Berechnungsstack mit einem `int`, einem `long` und einem Strukturobjekt 11
3 Aufteilung des Berechnungsstack bei zwei Methoden 12
4 Der Ausführungsrahmen 13
5 Zustände eines Thread 13
6 Zusammenfassung der Vor- und Nachteile der VMs 17
7 Gegenüberstellung von C#, MSIL und CVM Code. 19
8 Der Switch-Mode 20
9 Der Token-Mode 21
10 Der Direct-Mode 22
11 Die Strukturen eines Threads 23
12 Zu migrierende Daten bei *starker* Migration 25
13 CVM Code mit eingefügter Migrationsinstruktion. 32
14 E/A-Proxy mit zwei Programmen 35
15 Klassendiagramm des Abbilds 42
16 Rechenzeitstatistik für Benchmarks 53
17 Speicherstatistik für Benchmarks 54
18 Rechenzeitstatistik für Fraktalberechnung 55
19 Speicherstatistik für Fraktalberechnung 56
20 Zeitmessung der Migration 58

1 Einleitung

Dieses Kapitel zeigt Gründe für Anwendungsmigration auf und stellt existierende Lösungen vor. Im weiteren gibt es eine Übersicht über die Problembereiche bei Anwendungsmigration.

1.1 Gründe für Anwendungsmigration

Die Migration von Daten ist in heutigen, verteilten Systemen ein weit verbreiteter Ansatz, um Ziele wie Ausfallsicherheit oder Lastverteilung zu verwirklichen. Soll eine Aufgabe bewältigt werden und ist ein Rechner überlastet oder nicht mehr verfügbar, wird die Aufgabe einem anderen System zugeteilt. Hierfür kommen häufig Techniken wie CORBA [10] oder *Remote Procedure Calls* (RPC) [14] zum Einsatz. Bei diesen Systemen wird ein *Arbeitsauftrag* (die Daten) in Form eines RPC-Aufrufes oder durch das Verschicken einer Nachricht an einen Knoten innerhalb des Rechnernetz erteilt, dieser arbeitet den Auftrag ab und liefert ein Ergebnis. Das Ergebnis kann dabei synchron oder asynchron zur Verfügung gestellt werden. Moderne Programmierumgebungen wie Java oder C# besitzen eigene Schnittstellen für diesen Zweck. Java verfügt über *Remote Method Invocation* (RMI), welche auf den vorhandenen Serialisierungsmechanismen aufsetzt. C# stellt eine ähnliche Technologie namens *Remoting* zur Verfügung. Neben den reinen RPC-Mechanismen unterstützt z. B. Java in der Java 2 Enterprise Edition die *Java Message Service* Spezifikation, welche dem Entwickler die Möglichkeit gibt, verteilte Komponenten über Nachrichten miteinander kommunizieren zu lassen.

Der Wunsch, eine Anwendung zur Laufzeit auf ein anderes System zu verlagern (Code-Migration), hat ähnliche Gründe, wie bereits für die Migration von Daten genannt. Hierbei ist das Ziel allerdings nicht, die Verarbeitung auf einem anderen Rechner durch einen RPC-Aufruf oder eine Nachricht anzustoßen, sondern die komplette Anwendung inklusive ihrer Laufzeitinformationen auf das Ziel zu verschieben. Die Ausfallsicherheit wird dadurch erreicht, daß die Applikation zur Laufzeit einen von Abschaltung bedrohten Rechner verlassen kann und nach ihrer Wiederherstellung auf einem anderen Rechner die Arbeit fortsetzt. Außerdem wäre es möglich, von einer laufenden Anwendung periodisch *Schnappschüsse* anzufertigen und im Falle eines Rechnerausfalles den letzten erfolgreichen Schnappschuß zu reaktivieren. Das Ziel der Lastverteilung kann einfach erreicht werden, indem Applikationen vom überlasteten Rechner auf weniger belastete Systeme verschoben werden. Des

Weiteren wird die Migration auch als Mittel gesehen, um in Agentensystemen die Mobilität von Agenten zu realisieren. Bei mobilen Agenten soll das Programm zu den Daten *wandern* und nicht die Daten zum Programm übertragen werden.

Bei der Migration wird zwischen *starker* und *schwacher* Migration unterschieden. Als *stark* wird eine Migration bezeichnet, wenn sie alle Bestandteile einer laufenden Anwendung umfaßt [11]. Dazu gehören:

Code Bei der Migration wird der ausgeführte Programmcode auf das Zielsystem kopiert und muß dort nicht bereits vorliegen.

Data Die Daten, auf denen das Programm arbeitet, werden auf das Zielsystem übertragen.

State Der Zustand des Programms wird auf das Zielsystem transferiert und dort genutzt, um das Programm nahtlos fortzusetzen.

Unter *schwacher* Migration versteht man jede Migration, die mindestens eines der oben genannten Kriterien einer *starken* Migration nicht erfüllt, also auch die reine Datenmigration.

1.2 Aufgabenstellung

In der vorliegenden Arbeit soll die Möglichkeit der transparenten Migration von nebenläufigen Anwendungen innerhalb einer .NET Laufzeitumgebung untersucht werden. Hierfür soll eine existierende Laufzeitumgebung um Migrationsfunktionalität erweitert werden.

Zusätzlich zu den oben genannten Zielen sollen die folgenden Anforderungen erfüllt werden:
Minimierung der Änderungen: Die Veränderungen an der Laufzeitumgebung sollten so gering wie möglich sein. Dies soll das Einpflegen der Änderungen in neue Versionen der Laufzeitumgebung erleichtern.
Portabilität: Soweit möglich sollten die Veränderungen an der Laufzeitumgebung, bzw. die Kontrolle des Extraktions- oder Restaurierungsvorganges in der C#-Klassenbibliothek und nicht nativ innerhalb der Laufzeitumgebung implementiert werden. Dies ermöglicht zum einen eine höhere Portabilität der Veränderungen, zum anderen vereinfacht es die Fehlersuche erheblich.

Die vorliegende Arbeit soll sich in zwei Punkten von bereits existierenden Migrationsimplementationen für die .NET Plattform [17] unterscheiden. Zum

einen sollen Anwendungen ohne Modifikation ihres Programmcodes migriert werden können, zum anderen soll es möglich sein, nebenläufige Programme zu migrieren.

1.3 Das .NET Framework

Das .NET Framework von Microsoft definiert die Rahmenbedingungen einer sprachunabhängigen Laufzeitumgebung für Applikationen. Die Spezifikation der Laufzeitumgebung wurde bei der ECMA (European Computer Manufacturers Association) zur Standardisierung eingereicht. Die Spezifikation umfaßt zwei Dokumente: ECMA-334 [4] und ECMA-335 [5]. ECMA-334 beschreibt die primäre Zielsprache C#, ohne näher auf die Ausführung von Applikationen einzugehen. In der ECMA-335 werden die *Common Language Infrastructure* (*CLI*) und damit die Rahmenbedingungen für die Ausführung der Programme festgelegt.

Die Definition der CLI setzt sich aus folgenden Teilbereichen zusammen:

- Der Definition eines Dateiformates zum Abspeichern von Programminformationen. Das Dateiformat ist dabei nicht auf reine Instruktionen beschränkt. Vielmehr speichert es neben dem eigentlichen Programmcode weitere Metainformationen wie die Sichtbarkeit von Programmelementen (Klassen, Methoden oder Variablen) oder vom Benutzer angegebene Annotationen (Attribute). Zusätzlich sind Sicherheitsmerkmale wie Signaturen vorgesehen, die es ermöglichen, nur Programmcode aus gesicherten Quellen auszuführen.

- Dem Common Type System (*CTS*). Um das Ausführen von Applikationen, die in verschiedenen Sprachen entwickelt wurden zu ermöglichen, definiert der Standard ein gemeinsames Typsystem. Die Wertebereiche und Eigenschaften der verschiedenen Datentypen werden definiert und Rahmenbedingungen für die Interoperation festgeschrieben.

- Einem System zum Beschreiben von Metainformationen. Diese ermöglichen nicht nur der Laufzeitumgebung die Ausführung, sie dienen auch beim Übersetzen von Programmen der Sicherstellung der semantischen Korrektheit.

- Der Definition eines Zwischencodes (Intermediate Language, *IL* oder *MSIL*), der von den verschiedenen Compilern generiert werden kann

und den Programmfluß beschreibt. Dieser Zwischencode darf laut Spezifikation nicht direkt von der Laufzeitumgebung interpretiert werden, sondern soll erst in den Maschinencode des Zielrechners übersetzt werden (Just-In-Time Übersetzung, *JIT*).

- Einem Modell einer virtuellen Maschine, welches Regeln und Rahmenbedingungen für die Ausführung des MSIL-Code definiert. Dies umfaßt auch eine Beschreibung, welche Regeln ein Programm erfüllen muß, um von der Maschine als gültig akzeptiert zu werden.
- Einem Regelwerk, welches die Konvertierungen von Parametern und Rückgabewerten für den Aufruf von Systemfunktionen und -bibliotheken der zugrundeliegenden Plattform beschreibt (*marshalling*).
- Einer maschinenlesbaren Beschreibung der Systembibliotheken, die einem Programm zugänglich sind.

1.4 Existierende Lösungsansätze

Für die Umsetzung von Anwendungsmigration existieren mehrere Lösungsansätze, die hier im einzelnen kurz vorgestellt werden sollen.

1.4.1 Entwicklungskontrollierte Migration

Bei der entwicklungskontrollierten Migration erfolgt die Implementation der Migrationsfunktionalität vollständig durch den Entwickler der Anwendung.

Der Vorteil dieser Form der Migration ist die größtmögliche Flexibilität und Kompatibilität, da der Anwendungsentwickler mögliche Migrationspunkte am besten identifizieren kann. Zusätzlich verspricht diese Lösung die geringsten Geschwindigkeitseinbußen, da migrationsspezifischer Code nur an den wirklich notwendigen Stellen existieren muß. Der Nachteil liegt im erhöhten Entwicklungs- und Pflegeaufwand der Anwendung.

1.4.2 Übersetzungskontrollierte Migration

Migrationsaspekte können auch während der Programmerzeugung in eine Anwendung eingebracht werden. Beispiele wären hier ein Präprozessor, der Modifikationen am Quelltext vornimmt [7], modifizierte Bibliotheken, die

zum Programm hinzugelinkt werden[1] oder Postcompiler, die das aus dem Übersetzungsprozeß hervorgegangene Binärimage modifizieren [18, 17].

Ein Vorteil dieses Verfahrens ist eine gesteigerte Transparenz für das Programm bzw. den Entwickler, allerdings müssen die Compiler in der Lage sein, mögliche Migrationspunkte zu identifizieren. Bei Verfügbarkeit des Quelltextes ist dies einfacher, als wenn nur das (möglicherweise sogar von Debuginformationen bereinigte) Binärimage vorliegt. Die Qualität der Migration hängt bei diesem Verfahren also stark von der Qualität ab, mit der der Compiler den Programmfluß analysieren kann.

1.4.3 Laufzeitkontrollierte Migration

Bei diesem Verfahren wird die Migration von der verwendeten Laufzeitumgebung durchgeführt, ohne daß die Anwendung verändert wird. Hierbei liegt es in der Verantwortung des Entwicklers der Laufzeitumgebung, die Migrationspunkte zu wählen und für das Sichern und Wiederherstellen des Programmzustands zu sorgen. Bei der Laufzeitumgebung kann es sich um eine modifizierte virtuelle Maschine [9] oder um ein komplettes Betriebssystem [1] handeln.

Dieser Ansatz vereint die Transparenz der Migration für die Anwendung mit der Flexibilität und Kompatibilität von Eigenentwicklungen. Der Nachteil ist hierbei ein größerer Ressourcenverbrauch der Laufzeitumgebung, um die für eine eventuelle Migration nötigen Informationen bereitzustellen. Dieses Verfahren ist in der vorliegenden Arbeit zum Einsatz gekommen.

1.5 Problembereiche

Die vorliegende Arbeit soll die Probleme und Lösungsmöglichkeiten für die folgenden Teilaspekte einer Migration behandeln:

- Die Auswahl einer geeigneten Laufzeitumgebung
- Die Steuerung der Migration
- Das Auslesen des Programmzustands
- Das Restaurieren des Programmzustands

[1] z. B. eine modifizierte `libc` bei C Applikationen.

Die *Auswahl einer geeigneten Umgebung* ist der erste Schritt zur Lösung der gestellten Aufgabe. Die Implementation einer Laufzeitumgebung ist eine komplexe und zeitintensive Aufgabe. Daher war die Modifikation einer existieren Laufzeitumgebung die einzige Möglichkeit, im Rahmen dieser Arbeit Ergebnisse zu erzielen. Von den verfügbaren Laufzeitumgebungen mußte deshalb eine geeignete ausgewählt und modifiziert werden.

Die im folgenden aufgezählten Probleme orientieren sich an Abbildung 1. Nach erfolgter Wahl ist das Problem der *Steuerung der Migration* zu bewältigen. Das Auslösen des Migrationsprozesses ist ein zentrales Problem bei der Migration. Die Migration soll an jeder beliebigen Stelle im Programmfluß ausgelöst werden können, es gibt allerdings Programmzustände, bei denen eine Migration nur äußerst schwer möglich, wenn nicht sogar unmöglich ist. Nun folgt das *Auslesen des Programmzustands* durch ein geeignetes Element innerhalb der Laufzeitumgebungen. In diesem Schritt werden die Laufzeitinformationen des *eingefrorenen* Programms ausgelesen und z. B. in eine Datei oder eine Netzwerkverbindung serialisiert. Hierbei müssen alle Zustandsinformationen gesichert werden, die die Weiterverarbeitung am Ziel der Migration ermöglichen. Bei dem auslesenden Element kann es sich nicht um einen normalen Thread der Laufzeitumgebungen handeln, da dieser bereits im vorherigen Schritt stillgelegt worden wäre.
Ist die Sicherung der Informationen vollzogen und sind die Informationen auf das Zielsystem übertragen worden, schließt sich das *Restaurieren des Programmzustands* an. Am Zielort der Migration muß es der Laufzeitumgebung möglich sein, den serialisierten Programmzustand und alle damit verbundenen Metainformationen wieder herzustellen und die Ausführung des Programms an der Unterbrechungsstelle fortzusetzen.

1.6 Struktur der Arbeit

Im Folgenden soll die Struktur der vorliegen Arbeit beschrieben werden. Das folgende Kapitel 2 gibt zuerst eine allgemeine Einführung in das Konzept virtueller Maschinen. Danach geht es detaillierter auf die virtuelle Maschine des .NET Frameworks ein. Es folgt eine Beschreibung der Kriterien für die Auswahl einer geeigneten virtuellen Maschine für den praktischen Teil der Arbeit. Im Anschluß daran werden die internen Strukturen der gewählten Maschine erörtert.

In Kapitel 3 werden zunächst die Problemstellungen für eine Migration angesprochen. Nachfolgend werden mögliche Lösungsansätze vorgestellt und

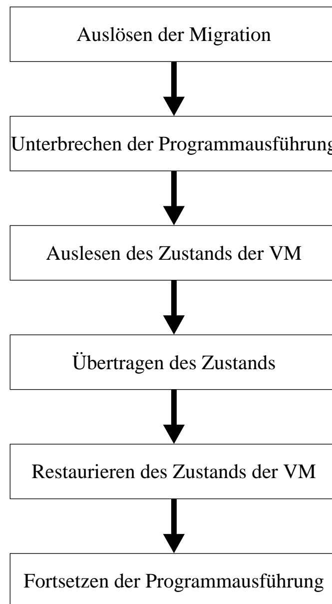

Abbildung 1: Probleme der Anwendungsmigration

eine erste Bewertung geliefert. Am Ende des Kapitels werden die gewählten Lösungsansätze aufgezählt und eine Begründung für die Auswahl gegeben. Der Aufbau des Kapitels orientiert sich dabei an den in Abschnitt 1.5 gegebenen Problembereichen.

Kapitel 4 erörtert die Details der Implementation die für diese Arbeit entstanden ist. Es präsentiert zunächst die Änderungen, die an der gewählten VM vorgenommen wurden. Danach gibt es einen Überblick über die für das Speichern des VM-Zustandes verwendeten Datenstrukturen und erklärt dann die Veränderungen an der Klassenbibliothek.

Eine Bewertung der Implementation findet in Kapitel 5 statt. Es werden die verwendeten Testfälle vorgestellt und die Testumgebung erläutert. Danach werden die gewonnenen Messergebnisse vorgestellt und bewertet.

Die Arbeit schließt mit Kapitel 6, wo die gewonnenen Ergebnisse noch einmal zusammengefaßt werden.

2 Die virtuelle Maschine

Dieses Kapitel gibt eine kurze Einführung in das Konzept von virtuellen Maschinen. Es erläutert den Entscheidungsprozeß zur Auswahl der für diese Arbeit verwendeten virtuellen Maschine und geht im Anschluß daran auf den internen Aufbau der benutzten Maschine ein.

2.1 Einführung

Viele Programmiersprachen werden nicht in Maschinencode für die Ausführung auf einer real existierenden Zielplattform (Prozessor und/oder Betriebssystem) übersetzt und dann gestartet, sondern die Ausführung des Programmes erfolgt in einer simulierten Umgebung. Hierbei wird der Programmtext vom Compiler in einen Zwischencode überführt, welcher dann interpretiert wird. Die Interpretation erfolgt durch die *Virtuelle Maschine* (VM). Beispiele für Sprachen mit Zwischencode sind Java [8] oder C# [4]. Um in diesen Sprachen erzeugte Programme ausführen zu können muß auf der Zielplattform des Programms eine lauffähige VM verfügbar sein.

Die virtuelle Maschine dient als Abstraktionsschicht zwischen der Anwendung und der verwendeten Plattform. Sie simuliert einen virtuellen Prozessor, der das Programm abarbeitet und stellt Interaktionsmöglichkeiten mit dem Betriebssystem wie z. B. Ein-/Ausgabeoperationen (E/A) bereit. Die Simulation des Prozessors kann dabei entweder durch Interpretation des Zwischencodes oder durch einen erneuten Übersetzungsschritt erfolgen. Im zweiten Fall wird der Zwischencode durch die VM vor der Ausführung in nativen, auf der Zielarchitektur lauffähigen Code übersetzt, um die Ausführungsgeschwindigkeit zu steigern. Diesen Vorgang bezeichnet man auch als Just-In-Time Übersetzung (JIT). Durch die Verwendung einer solchen Abstraktionsschicht ist es möglich, plattformunabhängige Programmdateien zu erzeugen. Ähnlich wie für reale Prozessoren existieren für die virtuellen Prozessoren der VMs detaillierte Spezifikationen. Diese beschreiben die verfügbaren Instruktionen (Opcodes) und das Programmiermodell des Prozessors.

Die VM stellt Applikationen ähnliche Funktionalitäten, wie sie von Betriebssystemen geboten werden, zur Verfügung. Dazu zählen neben den Eingangs genannten E/A-Funktionen auch die Speicherverwaltung und die Möglichkeit der nebenläufigen Programmierung. Die Funktionen, die eine VM einem Anwendungsprogrammierer zugänglich macht, werden im Allgemeinen über ein dazugehöriges *Application Programming Interface* (API), die Laufzeitbibliothek, angeboten.

2.2 Die VM des .NET Frameworks

Der Programmcode und Ressourcen für ein .NET Programm werden in *Assemblies* gespeichert. Bei Assemblies kann es sich entweder um Programme oder dynamische Bibliotheken handeln. Die Assemblies enthalten Klassendefinitionen, Metadaten und Programmressourcen. Der eigentliche MSIL-Programmcode befindet sich in den Methoden der Klassen.

Innerhalb der Laufzeitumgebung kommen unterschiedliche Ausführungsmodelle zum Einsatz. Als *Managed-Code* bezeichnet man MSIL-Code, der innerhalb der Laufzeitumgebung ausgeführt wird. Plattformabhängige Bibliotheken oder interne Funktionen der Laufzeitumgebung werden als *Native-Code* bezeichnet. Der Wechsel zwischen *Managed-* und *Native-Code*, sowie die dafür notwendige Konvertierung der Parameter und Rückgabewerte als *marshalling*.

Der virtuelle Prozessor der VM des .NET Frameworks verfügt nicht, wie viele reale Prozessoren, über Register, um Zwischenergebnisse von Berechnungen zu speichern. Die Operanden einer Instruktion werden auf einem Stapelspeicher (Berechnungsstack) abgelegt. Nach dem Ausführen der Instruktion befindet sich auch das Ergebnis auf dem Berechnungsstack. Eine weitere Aufgabe für diesen Stack ist das Speichern von Parametern bzw. Rückgabewerten beim Aufrufen von Methoden. Auch die lokalen Variablen einer Methode werden auf dem Berechnungsstack gespeichert.

Das .NET Framework kennt vier unterschiedliche Datentypen: primitive Typen, Objekte, Strukturobjekte und Zeiger. Zu den primitiven Datentypen zählen einfache numerische und logische Typen wie z.B. `bool` für Wahrheitswerte, `int` für ganzzahlige Werte oder `double` für Gleitkommawerte. Objekte sind komplexe Datentypen, deren Speicher auf dem Heap reserviert wird und die bei Methodenaufrufen als Referenz übergeben werden. Strukturobjekte hingegen werden auf dem Stack erzeugt und bei Methodenaufrufen als Kopie auf dem Parameterstack übergeben. Zeiger können auf alle vorher genannten Datentypen verweisen. In Abbildung 2 ist exemplarisch ein Berechnungsstack auf einem 32-Bit Rechner dargestellt. Er enthält einen Integerwert (benötigt einen Slot auf dem Stack), einem Longwert (benötigt zwei Slots) und ein 17 bis 20 Byte großes Strukturobjekt.

Jede ausgeführte Methode erhält auf dem Berechnungsstack einen logisch dreigeteilten Bereich zugewiesen. Dieser Bereich besteht aus den Parametern für die Methode, den lokalen Variablen und den Zwischenergebnissen der Berechnungen. Die Bereiche für die Berechnungsergebnisse der aufrufenden

Abbildung 2: Berechnungsstack mit einem `int`, einem `long` und einem Strukturobjekt

Methode und der Parameter der aufgerufenen Methode überschneiden sich dabei. In Abbildung 3 ist der Aufbau des Stack beispielhaft für zwei Methoden (A und B) aufgezeichnet. Der aktuelle Ausführungsrahmen wird auch als aktives Fenster bezeichnet. Zusammen mit Informationen wie der Rücksprungadresse und der assoziierten Methode bilden die drei Stackinhalte den Ausführungsrahmen (kurz Rahmen). Auch diese Ausführungsrahmen bilden einen Stack (Abbildung 4), den Aufrufstack. Das oberste Element des Aufrufstack enthält dabei immer den Rahmen der aktuell ausgeführten Methode.

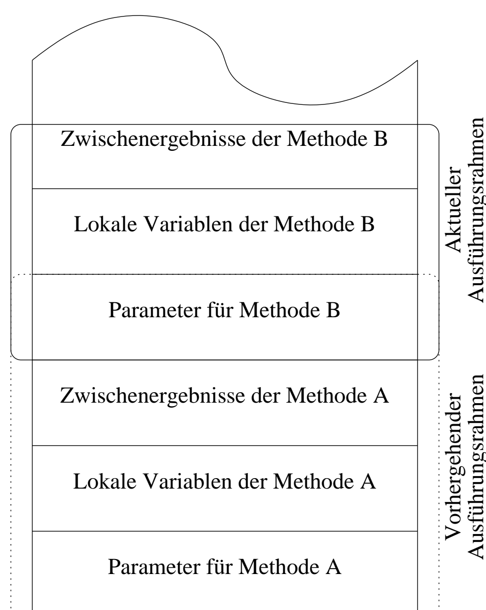

Abbildung 3: Aufteilung des Berechnungsstack bei zwei Methoden

Das .NET Framework unterstützt die nebenläufige Abarbeitung von Programmen. Bei start einer VM wird ein initialer Thread erzeugt (Mainthread), der am Eintrittspunkt des Programmes mit der Ausführung beginnt. Der Programmierer kann in seinen Programmen beliebig viele Threads starten. Diese laufen dann alle innerhalb des gleichen Kontexts, d. h. sie teilen sich den Arbeitsspeicher und damit die Daten, die sie bearbeiten.

Abbildung 5 gibt die verschiedenen Ausführungszustände eines Threads innerhalb der VM wieder[2]. Nach seiner Erzeugung kann ein Thread gest-

[2] Auf Zustände wie `SuspendRequested` oder `AbortRequested` wird hier nicht eingegan-

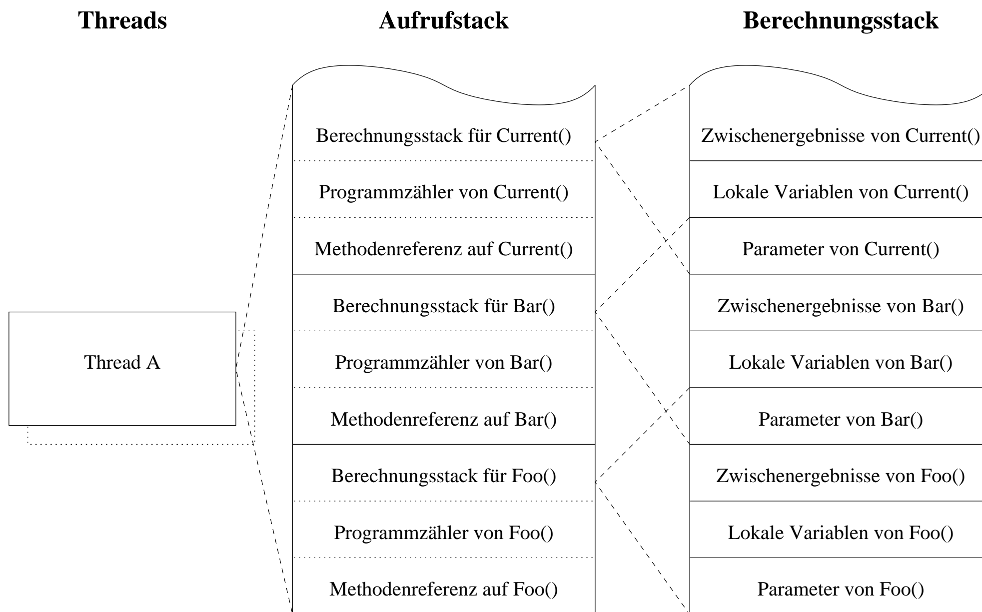

Abbildung 4: Der Ausführungsrahmen

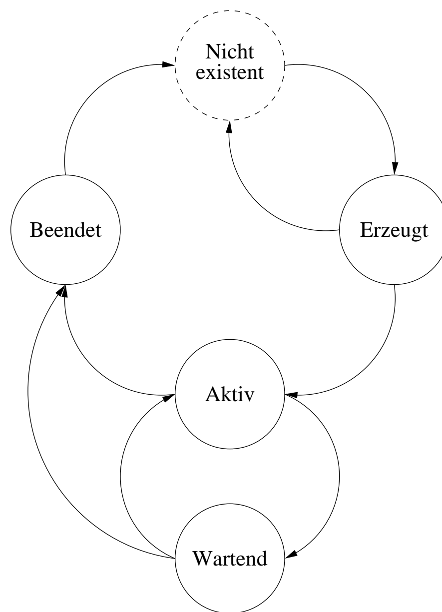

Abbildung 5: Zustände eines Thread

artet werden und beliebig lange laufen. Während seiner Laufzeit kann er sich selbst anhalten (`Wait()`, `Sleep()`) oder von außen angehalten werden (`Abort()`, bzw. durch Synchronisationsmechanismen). Ein einmal erzeugter Thread kann nur in den Zustand *nicht existent* übergehen, wenn die letzte Referenz auf das Threadobjekt gelöscht wurde. Ein Thread hat neben den für die Ausführung des Programmcode notwendigen Zustandsinformationen wie den Programmzähler (PC) oder die aktuelle Methode noch weitere assoziierte Informationen. Dazu zählen Attribute wie sein Name, die Priorität oder der Ausführungszustand. Zusätzlich zu der reinen Threadfunktionalität bietet die .NET VM Synchronisationsmechanismen, um konkurrierende Zugriffe bei der nebenläufigen Programmierung zu koordinieren. Der Programmierer kann kritische Programmstellen über Monitore synchronisieren [15]. Hat ein Thread einen Monitor belegt, so müssen alle anderen Threads bis zur Freigabe des Monitors warten, wenn sie diesen ebenfalls belegen wollen.

Die Freispeicherverwaltung wird in .NET durch einen *Garbage Collector* [23] genannten Mechanismus übernommen. Dieser überwacht den vom Anwendungsprogramm benutzten Heapspeicher und erkennt, wenn Teile nicht mehr benutzt werden. Diese unbenutzten Teile werden dann als wieder verfügbar markiert und bei nachfolgenden Speicheranforderungen erneut vergeben. In der .NET VM ist der Garbage Collector ein eigener Thread, der mit niedriger Priorität in der VM läuft.

Die Systembibliotheken des .NET Frameworks befinden sich in dynamische Bibliotheken wie z. B. der `mscorlib.dll` oder der `System.dll`. Diese Bibliotheken sind ein fester Bestandteil der Laufzeitumgebung.

2.3 Auswahl der VM

Für die Umsetzung der Migrationsfunktionalität mußte eine geeignete Laufzeitumgebung gewählt und erweitert werden. Von den verfügbaren Möglichkeiten kamen auf Grund der Komplexität der Veränderungen nur im Quelltext verfügbare VMs in Betracht. Der Quelltext sollte, wenn möglich, unter einer GPL [6] kompatiblen Lizenz vorliegen (*Freie Software*), da dies die Weitergabe des modifizierten Quellcodes erleichtert. Ein weiteres Kriterium, neben der Verfügbarkeit des Quelltextes, war die Ausführungsgeschwindigkeit. Die im folgenden aufgezählten Implementationen wurden auf ihre Eignung für das Projekt untersucht und sollen nun kurz vorgestellt, bzw. ihre

gen, da sie sehr spezielle VM-interne Informationen widerspiegeln.

Vorzüge, sowie Nachteile besprochen werden. Ein weiteres wichtiges Kriterium war die Komplexität der VM. Besitzt eine VM einen JIT, so wird die Fehlersuche mit Hilfe eines Debuggers stark erschwert, da zum Ausführungszeitpunkt des Programmes dynamisch Maschinencode generiert wird für den kein Quelltext vorliegt. VMs mit Interpreter lassen sich deshalb leichter zur Laufzeit untersuchen.

2.3.1 Die MS-Rotor VM

Die Rotor VM [22] ist eine frühe Beispielimplementation des .NET Frameworks von Microsoft und ist unter Microsofts *Shared Source* Lizenz verfügbar. Sie wurde schon in mehreren akademischen Projekten benutzt und setzt den ECMA-Standard relativ vollständig um. Es existiert ein JIT für x86 und PowerPC Prozessoren. Die VM ist in C++ implementiert.

Für Rotor spricht die relativ vollständige Umsetzung der ECMA Spezifikationen.

Gegen Rotor spricht, das die VM keine *Freie Software* ist, Programme extrem langsam ausgeführt werden und die Plattformabhängigkeit des JIT. Zusätzlich macht der Einsatz eines JIT das Extrahieren und Restaurieren des Programmzustands schwieriger.

2.3.2 Die Mono-VM

Die Mono VM [16] ist eine freie Implementation der .NET Laufzeitumgebung, die von Novell gefördert wird. Sie verfügt über einen JIT für x86, PowerPC, Sparc und S390 Prozessoren. Neben dem JIT existiert ein Interpreter (`mint`). Dieser ist allerdings auf Grund seiner Geschwindigkeit nicht für den realen Einsatz bestimmt. Er dient den Entwicklern hauptsächlich zum Testen der nativen Bibliotheksfunktionen. Die VM ist in C implementiert.

Für Mono sprechen die hohe Ausführungsgeschwindigkeit, die große Entwicklergemeinde und die Tatsache, daß es sich um *Freie Software* handelt.

Gegen Mono sprechen der unstrukturierte Quellcode, die schlechte Dokumentation der VM-Interna und die von Rotor bekannten Probleme durch den JIT.

2.3.3 Die PNet VM

Die PNet VM [19] kommt im Rahmen des DotGNU Projektes zum Einsatz. Ihr primäres Ziel liegt auf einer möglichst breiten Unterstützung verschiede-

ner Prozessoren und Plattformen. Im Gegensatz zu den beiden vorgenannten VMs verfügt die PNet VM nicht über einen *echten* JIT. Der MSIL-Code der Programme wird in einen weiteren Zwischencode übersetzt und dieser dann von der *Coding Virtual Machine* (CVM) interpretiert [21]. Der hierbei eingesetzte Übersetzer (Coder) ist zum einen in der Lage bestimmte Methodenaufrufe einzubetten[3] (inlining), zum anderen ist er auch in der Lage andere Zwischencodes in CVM Instruktionen zu übersetzen. Neben MSIL-Code ist das Laden von Java-Bytecode bereits im Coder umgesetzt. Die VM ist in C implementiert. Für die PNet VM befindet sich ein JIT in Form einer dynamischen Bibliothek (libjit) in der Entwicklung, diese wird allerdings in der PNet VM noch nicht verwendet.

Für PNet sprechen eine ausreichende Ausführungsgeschwindigkeit, der gut strukturierte Quellcode und die Tatsache, daß es sich um *Freie Software* handelt. Der Hybridansatz einer re-kodierenden VM reduziert die Komplexität der VM und erleichtert somit Veränderungen.

Gegen PNet sprechen die schlechte Dokumentation, die kleine Kernentwicklergemeinde und die damit verbundenen Implementationslücken in Teilen des API.

[3] z. B. Math.Sqrt()

2.3.4 Zusammenfassung der Auswahlkriterien

In der nachfolgenden Tabelle sind die verschiedenen Vor- und Nachteile noch einmal zusammengefaßt:

	Rotor	**Mono**	**PNet**
ECMA Konformität	++	+	−
Geschwindigkeit	−−	++	+
Dokumentation	−	−−	−
Größe der Entwicklergemeinde	−−	++	+
Lizenz	−−	++	++
Komplexität der Änderungen	−−	−−	++

Abbildung 6: Zusammenfassung der Vor- und Nachteile der VMs

Unter der Prämisse, daß die Veränderungen an der VM möglichst einfach durchführbar sind, das ausgeführte Programme aber nicht zu langsam ablaufen dürfen, ist die Wahl auf die PNet VM gefallen. Von den oben aufgeführten Nachteilen wog das Problem der Implementationslücken am schwersten.

2.4 Interner Aufbau der PNet VM

Im folgenden soll kurz der Aufbau der PNet Laufzeitumgebung verdeutlicht werden. Dies soll helfen, die aufgetretenen Probleme und die an der Laufzeitumgebung durchgeführten Veränderungen nachzuvollziehen.

Die PNet Laufzeitumgebung besteht aus mehreren Teilen, die zusammen eine Plattform zum Erstellen und Ausführen von .NET Applikationen bilden. Zum PNet Projekt gehören unter anderem der C# Compiler `cscc`, die virtuelle Maschine `ilrun`, die Laufzeitbibliotheken (`mscorlib.dll`, `System.dll`, usw.). Weiterhin existieren Hilfsprogramme wie `ildasm` zum disassemblieren von Programmen oder `ilgac` zum Verwalten der Assemblies im Assemblycache.

Die PNet-VM besitzt, wie bereits in Kapitel 2.3.3 erwähnt, keinen *echten* Just-in-time-Compiler. Die vom *Loader* geladenen Images werden durch den sogenannten Coder in einen Zwischencode (CVM-Code) übersetzt, der auf schnelle Interpretation optimiert ist. Im Coder ist der *Verifier* integriert, welcher geladene Images auf die Einhaltung der Regeln der CLS überprüft. Neben der Typsicherheit von geladenen Programmen werden Rahmenbedingungen, wie die Gültigkeit von Sprüngen innerhalb von Methoden, als auch die Zulässigkeit von Operationen auf dem Stackinhalt überprüft. Dies ist die einzige Stelle innerhalb der VM in der die Typinformationen auf dem Stack bekannt sind.

2.4.1 Übersetzungsstadien

Der Geschwindigkeitsvorteil des PNet-Interpreters gegenüber anderen Interpretern wird unter anderem dadurch erreicht, daß statt der überladenen MSIL Instruktionen, wie z. B. `add` oder `mul`, typisierte Varianten (`iadd` und `fmul`) eingesetzt werden. So entfallen Typüberprüfungen und bedingte Konvertierungen zur Laufzeit des Programmes. Dem Coder ist es beim Überführen des MSIL-Codes in CVM-Code auch möglich, für eine MSIL-Instruktion mehrere CVM-Instruktionen zu generieren. So müssen wiederkehrende Funktionsblöcke nur einmal in C kodiert werden und können durch den Coder für unterschiedliche MSIL-Opcodes mit benutzt werden. Ein kurzes Beispiel für die verschiedenen Übersetzungsstadien eines C# Programmes ist in Abbildung 7 gegeben. Das Programm liegt einmal als C# Quelltext, einmal als MSIL-Opcodes und einmal als CVM-Opcodes vor.

C#

```
int i = 11;
i += 22;
double d = i * 33.0;
```

MSIL

```
[...]
ldc.i4.s   11
stloc.0
ldloc.0
ldc.i4.s   22
add
stloc.0
ldloc.0
conv.r8
ldc.r4     float32(0x42040000)
mul
conv.r8
stloc.1
ret

[...]
```

CVM

```
[...]
ldc_i4_s   11
istore_0
iload_0
ldc_i4_s   22
iadd
istore_0
iload_0
i2f
ldc_r4     33
fmul
f2d
waddr      1
dwrite_r
return
[...]
```

Abbildung 7: Gegenüberstellung von C#, MSIL und CVM Code.

2.4.2 Interpretermodi

Für jeden CVM-Opcode existiert innerhalb der Interpreterschleife der CVM ein Block, der die gewünschte Funktionalität implementiert. Die eigentliche Interpreterschleife innerhalb der Maschine kennt mehrere Ausführungsmodi. Diese werden zur Übersetzungszeit der VM anhand der Fähigkeiten des Prozessors und C-Compilers ausgewählt. Je nach verwendetem Modus ist die Bedeutung des Wertes, auf den der Programmzeiger (*PC*) verweist, unterschiedlich. Die folgenden Modi existieren:

Switched: Die Interpreterschleife besteht aus einem großen `switch`-Statement und der Coder überführt den MSIL-Code in Integerwerte. Diese Integerwerte werden durch das `switch` benutzt um den entsprechenden Funktionsblock durch ein `case` auszuwählen. (siehe Abb. 8). Der PC zeigt somit auf einen Integerwert der durch das Switch interpretiert wird. Dies ist der langsamste Modus.

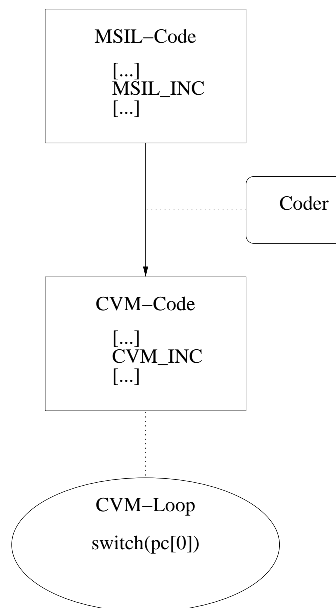

Abbildung 8: Der Switch-Mode

Token: Auch im Tokenmodus werden durch den Coder Integerwerte generiert. Diese werden allerdings nicht direkt durch ein `switch` interpretiert, sondern sie dienen als Offsets in eine Sprungtabelle. Diese Tabelle enthält für jeden CVM-Opcode die Startadresse des Funktionsblocks. Innerhalb der Interpreterschleife wird nun mit Hilfe von `goto`-Statements der nächste auszuführende Codeblock angesprungen (siehe Abb. 9). Hier zeigt der PC also auf Integerwerte welche ein Offset in die Sprungtabelle darstellen.

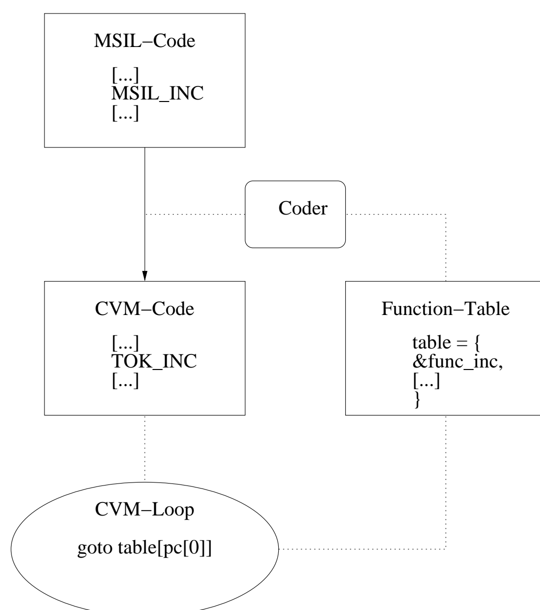

Abbildung 9: Der Token-Mode

Direct: Im Directmodus schreibt der Coder statt Integerwerten, die erst noch einer weiteren Interpretation bedürfen, direkt die Adressen der Funktionsblöcke in den Methodenkörper. In der Interpreterschleife wird direkt die Adresse angesprungen, welche im übersetzten Methodenkörper an nächster Stelle des PC steht (siehe Abb. 10). Hier zeigt der PC direkt auf die nächste anzuspringende Adresse. Dies ist der schnellste Modus.

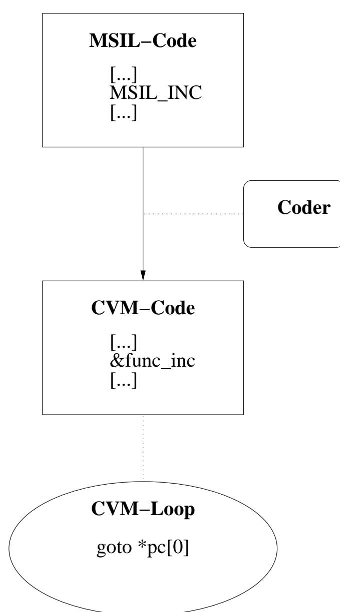

Abbildung 10: Der Direct-Mode

Parameter zu CVM-Instruktionen bleiben bei allen drei Verfahren erhalten und stehen direkt hinter den generierten Instruktionen im übersetzten Methodenkörper.

2.4.3 Threading

Für jeden Thread innerhalb eines laufenden Programms hält die VM mehrere Strukturen vor: `ILThread`, `ILExecThread` und `System_Thread`. Ein `ILThread` ist die abstrahierte Repräsentation eines Threads der verwendeten Plattform (Unix oder Win32). Ein `ILExecThread` enthält alle für die Ausführung innerhalb der Interpreterschleife notwendigen Informationen. `System_Thread` ist ein Objekt und definiert die VM-interne Repräsentation der Klasse `System.Threading.Thread` (STThread). Der Aufbau ist in Abbildung 11 noch einmal verdeutlicht.

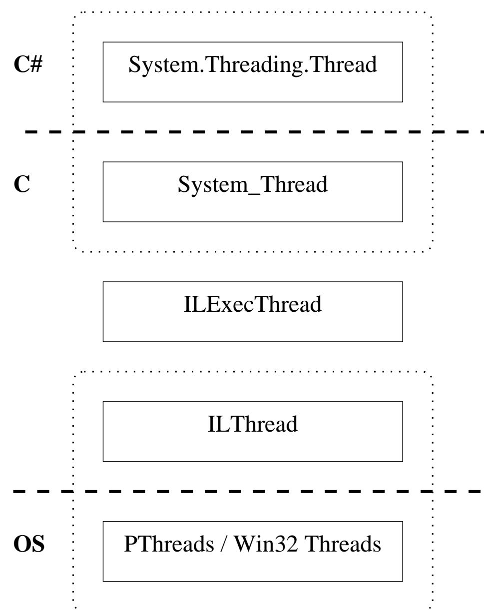

Abbildung 11: Die Strukturen eines Threads

In der Struktur `ILExecThread` sind neben dem Programmzeiger (*PC*), welcher die aktuelle Ausführungsposition innerhalb der aktiven Methode angibt, auch die Zeiger auf den Ausführungsstack enthalten. Auf diesem Stack finden die Berechnungen statt, werden Parameter und Rückgabewerte für Methodenaufrufe übergeben und alle lokalen Variablen angelegt (siehe auch Kapitel 2.2). Ruft Methode `A` jetzt Methode `B` auf, so setzt sie alle Parameter für `B` am Ende ihres Ausführungsstack und springt danach `B` an. Das aktive Fenster wird dann so verschoben, daß das Ende des Ausführungsstack von `A`

der Anfang des Stacks von `B` darstellt. Die Grenzen zwischen zwei Rahmen auf dem Ausführungsstack überschneiden sich also, wie in Abbildung 3, Seite 12 zu sehen ist. Neben dem Ausführungsstack existiert noch der Aufrufstack, auf dem die Aufrufhistorie für bereits aufgerufenen Methoden abgelegt ist. Hier wird jeweils die Position des Ausführungsrahmen auf dem Ausführungsstack, sowie die unterbrochene Methode und der PC innerhalb dieser Methode abgelegt. Diese Informationen sind in `ILCallFrame` beschrieben, welches einem Element des Aufrufstack aus Abbildung 4, Seite 13 entspricht.

Zusätzlich zu den oben beschriebenen threadspezifischen Strukturen existieren noch weitere, für das Zusammenspiel der verschiedenen Teile der VM wichtige Strukturen. So existiert pro VM Instanz ein `ILExecProcess`, welcher VM globale Daten wie Synchronisationspunkte, Zeiger auf geladene Images und existierende Threads hält. Weiterhin gibt es noch Verwaltungsstrukturen für geladene Assemblies (`ILImage`), Klassen (`ILType`) und Methoden (`ILMethod`).

3 Migration

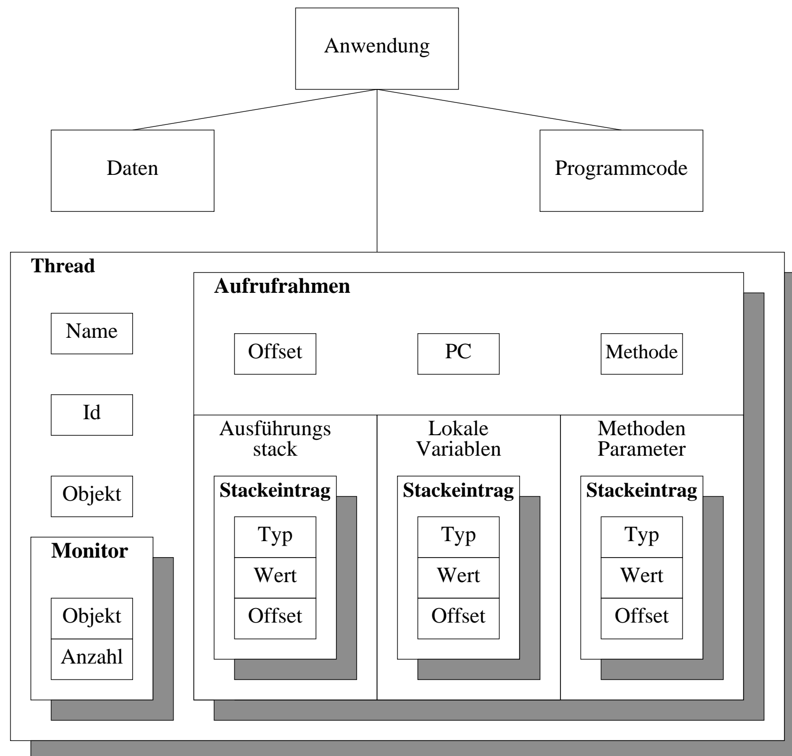

Abbildung 12: Zu migrierende Daten bei *starker* Migration

In diesem Kapitel sollen zuerst die zu lösenden Probleme für die Migration einer Anwendung beschrieben und im Anschluß daran deren mögliche Lösungen vorgestellt werden. Am Ende werden die implementierten Lösungen vorgestellt. Die Beschreibung geschieht in Anlehnung an die Reihenfolge aus Abschnitt 1.5, Seite 5. Ergänzend zu der Beschreibung der Funktionsweise der PNet VM soll Abbildung 12 noch einmal die verschiedenen Teile einer Applikation verdeutlichen. Graue Schatten hinter einer Entität deuten an, daß diese mehrfach vorhanden sein kann. Eine Anwendung besteht zur Laufzeit aus passiven und aktiven Komponenten. Zu den passiven Komponenten gehört der Programmcode und alle *erreichbaren* Daten des Programms; die Threads stellen die aktiven Komponenten dar. Als *erreichbare* Daten gelten hier alle Objekte, auf die von einem der laufenden Anwendungsthreads

zugegriffen werden kann[4].

Die Gesamtheit der Daten und Zustandsinformationen, die nötig sind um ein Programm zu restaurieren, werden im Folgenden als *Abbild* bezeichnet.

3.1 Problemraum

Das Auslösen des Migrationsprozesses ist ein Problem, das zuerst gelöst werden muß. Hierbei ist zu entscheiden, wie die Migrationsanforderung an die VM kommuniziert wird. Ist z. B. gefordert, die VM von einem anderen Rechner unterbrechen zu können, so muß die Migrationsanforderung über Netzwerkfunktionen angeboten werden. Reicht es, die VM lokal unterbrechen zu können, so kann der Mechanismus einfacher gehalten werden. Möglicherweise darf die Migration nicht von jedem Benutzer gestartet werden, so daß Authentifizierungsmechanismen nötig sind.

Erhält die VM einen Migrationsauftrag, so muß überprüft werden, ob eine Migration zum aktuellen Zeitpunkt möglich ist. Wie bereits in Kapitel 2.4 beschrieben besitzt die PNet-VM keinen echten JIT. Statt dessen findet die Ausführung innerhalb einer optimierten Interpreterschleife statt. Aus mehreren Gründen ist es trotz dieses Aufbaus nicht einfach möglich, nach der Interpretation jeder MSIL- bzw. CVM-Instruktion die VM anzuhalten und die Migration zu starten. Einerseits kostet die Überprüfung der Migrationsanforderung bei jeder Instruktionsausführung Zeit, zum anderen ist nicht garantiert, daß der Inhalt des Stacks nach jeder CVM-Instruktion auch gültig ist. Es können sich aber durchaus ungültige Werte auf dem Stack befinden, wenn eine MSIL-Instruktion durch mehrere CVM-Instruktionen simuliert wird. Da für die Migration bestimmte Bedingungen an einen Thread gestellt werden, wurde der Begriff *sicherer Punkt* definiert. An *sicheren Punkten* ist es möglich die Migration erfolgreich durchzuführen. Als *sicher* gilt ein Punkt, wenn die folgenden zwei Eigenschaften erfüllt sind:

- Der Thread darf zum Zeitpunkt der Suspendierung keine nativen Methoden ausführen. Dies gilt nicht nur für den aktuellen Ausführungsrahmen, sondern auch für alle auf dem Aufrufstack befindlichen Methoden.

- Für jeden Eintrag auf dem Ausführungsstack muß der Typ bekannt sein.

[4] D. h. auf diese Objekte existieren noch Referenzen und sie dürfen somit beim nächsten Lauf des Garbage Collectors nicht aus dem Speicher entfernt werden.

Die erste Bedingung soll sicherstellen, daß sich der Thread unterbrechen läßt. Hierfür muß er unter der Kontrolle der Interpreterschleife der VM abgearbeitet werden. Befindet sich der Thread innerhalb einer nativen Funktion, so ist es möglich, das sich nicht alle Zustandsinformationen in den Strukturen der VM befinden. Hierdurch kann dieser Zustand nicht in das Abbild integriert und übertragen werden. Diese Bedingung bewirkt, daß laufende Threads nur im Zustand *Aktiv*[5] migriert werden können. Noch nicht gestartete Threads, bzw. bereits beendete Threads enthalten keine Ausführungsinformationen und zählen somit zu den passiven Daten. Die zweite Bedingung ermöglicht es, alle auf dem Stack befindlichen Daten korrekt in das Abbild zu integrieren. Dies ist wichtig, da unterschiedliche Datentypen unterschiedlich viel Platz auf dem Stack benötigen.

Wurde ein Zeitpunkt als *sicher* identifiziert, müssen alle Threads in ihrer Ausführung unterbrochen werden. Hierfür müssen zuerst alle *migrationsfähigen* Threads der VM identifiziert werden können. Als *migrationsfähig* gelten alle Anwendungsthreads. VM eigene Threads wie der Garbage Collector müssen von der Migration ausgeschlossen werden. Zusätzlich ist zu beachten, daß ein Thread, der noch nicht unterbrochen wurde, jederzeit in der Lage ist, weitere Threads zu starten. Es ist also ein geeigneter Mechanismus zu finden um einzelne oder alle Threads anzuhalten.

Wenn alle Threads angehalten wurden, muß ihr Zustand ausgelesen und in das Abbild geschrieben werden. Die für das Abbild gewählte Repräsentation muß geeignet sein, alle auftretenden Zustandsdaten zu speichern. Hierzu zählen für aktive Threads alle in Abbildung 12 gezeigten Werte. Das Auslesen der Zustandsdaten und das Erzeugen des Abbildes muß durch eine geeignete Entität innerhalb der VM gesteuert werden. Damit Methoden wie `Join()` oder `Start()` nach der Migration auch weiterhin funktionieren, müssen für noch nicht gestartete bzw. bereits beendete Threads entsprechende Maßnahmen ergriffen werden. Zusätzlich müssen alle durch die Anwendung erreichbaren Daten in das Abbild integriert werden. Diese Daten setzen sich aus zwei Teilen zusammen: Zum einen existieren *lokale*, über Objektreferenzen erreichbare Daten. Diese können über die auf dem Stack befindlichen Objekte erreicht werden. Zum anderen gibt es noch *global* erreichbare Daten, die über statische Felder innerhalb von Klassen referenziert werden. Die Daten können aus allen in Kapitel 2.2 erwähnten Datentypen bestehen. Die Migration der verschiedenen Typen erzeugt unterschiedliche Aufwandsklassen.

[5] Siehe Abbildung 5, Seite 13 für die möglichen Zustände.

Zu den *leicht migrierbaren Datentypen* gehören primitive Datentypen und Objekte. Primitive Typen haben auf dem Stack eine feste Größe, die zum Zeitpunkt der Entwicklung einer modifizierten VM bekannt sind. Objekte befinden sich auf dem Heap, Referenzen der Objekte auf dem Stack haben, genau wie die primitiven Typen, eine feste Größe.
Die Strukturobjekte zählen zu den *schwer migrierbaren Datentypen*. Sie nehmen auf den Stacks unterschiedlich viel Platz ein und erhöhen deshalb die Anforderungen an das Auslesen und Zurückschreiben der Stackinhalte.
Zeiger bilden die Klasse der *nicht migrierbare Datentypen*. So lange ein Zeiger auf Datenstrukturen verweist, die innerhalb der VM erzeugt wurden ist es theoretisch noch möglich ihn als Referenz auf diese Struktur zu identifizieren und zu migrieren. Wird der Zeiger allerdings im Zusammenhang mit nativen Systembibliotheken verwendet, so lassen sich die referenzierten Daten nicht in das Abbild integrieren. Bei der Entscheidung für eine der Aufwandsklassen müssen die Fähigkeiten des gewählten Speicherformates für das Abbild berücksichtigt werden.

Die dynamische, vom Typ des Objektes abhängige Größe von Strukturobjekten stellt (siehe Beispiel in Abbildung 2, Seite 11) komplexere Anforderungen an die Migration bezüglich der Offset- und Größenberechnung auf dem Ausführungsstack.

Die Zustandsinformationen eines Threads, wie PC oder aktuelle Methode, liegen direkt in einer der Threadstrukturen aus Abschnitt 2.4.3, Seite 23 vor. Die einzige Ausnahme bilden die von einem Thread gehaltenen Monitore. Welcher Thread einen bestimmten Monitor hält wird innerhalb der Verwaltungsstrukturen der Objekte protokolliert. Diese Informationen müssen also in geeigneter Art zugreifbar gemacht werden.

Neben den in der VM gespeicherten Zustandsinformationen, wie Threadstatus und Programmdaten, existieren weitere Abhängigkeiten eines Programmes von seiner Umgebung. Beispiele hierfür sind die Schreib- /Leseposition innerhalb einer geöffneten Datei oder eine bestehende Socketverbindung, die nicht auf einen anderen Rechner verschoben werden kann. Eine weitere Ressource ist die für das Programm sichtbare Systemzeit, da sich Programme oft darauf verlassen, das die Zeit keine Sprünge (evtl. sogar in die Vergangenheit) macht [6]. Für diese Ressourcen sind also andere Vorgehensweisen als für Zustandsinformationen aus der VM nötig.

Wurde eine Anwendung erfolgreich suspendiert, so müssen alle Bestand-

[6] z. B. wenn der Zielrechner über eine andere Zeitzoneneinstellung verfügt

teile der Anwendung auf den Zielrechner übertragen werden.

Auf dem Migrationsziel muß eine neue VM erzeugt werden. Diese muß die im Abbild gespeicherten Informationen auslesen und ihren internen Zustand entsprechend anpassen. Hierfür muß ein Mechanismus in der VM existieren, der die Restauration steuert.

Der erste Schritt bei der Restauration ist das Erzeugen des Objektbaumes mit allen im Abbild vorhandenen Objekten. Als nächstes müssen die Threads der Applikation neu erzeugt werden. Durch Restauration erzeugte Threads sind keine vollständigen Threads, wie sie bei normaler Programmausführung entstehen würden. Sie sind vielmehr leere *Threadhüllen*, die mit den aus Abbildung 12 bekannten Werten gefüllt werden müssen.

Nach der Restauration der Daten und der Threadzustände müssen die Threads an der Stelle fortgesetzt werden, wo sie auf der Quellmaschine unterbrochen wurden. Damit ist die Migration der Applikation abgeschlossen.

3.2 Vorstellung und Bewertung der Lösungsansätze

Um die Unterbrechungsanforderung an die VM zu senden, sind mehrere Vorgehensweisen möglich. Zum einen Wege der Interprozeßkommunikation, wie das klassische System V IPC [12], welches nur lokal funktioniert. Zum anderen ist eine Socketschnittstelle denkbar, welche auch von anderen Systemen ansprechbar wäre. Beide Möglichkeiten bieten eine große Funktionalität, da das Versenden komplexer Kommandos und das Gewinnen detaillierter Informationen aus der VM umsetzbar sind. Gleichzeitig ist dies auch der Nachteil bei diesen Ansätzen: Um die beschriebenen Funktionen verfügbar zu machen, ist ein hoher Arbeitsaufwand nötig. Eine weitere, leichter zu implementierende Möglichkeit bietet die Verwendung von UNIX Signalen [13]. Dieser Mechanismus ist einfach zu verwenden und bietet zudem einen gewissen Schutz gegen mißbräuchliche Benutzung durch Dritte. Die Migration kann somit nur von am Rechner angemeldeten Benutzern (z. B. mit Hilfe des Unix-Kommandos *kill*) gestartet werden. Neben diesen, VM-externen Möglichkeiten ist es denkbar, eine Migrationsmethode in der Laufzeitbibliothek zu implementieren. Dieser Mechanismus ist allerdings nicht völlig transparent, da ein Programm, das diese Methode benutzt, nur noch gegen die veränderte Bibliothek kompiliert werden kann. Hierdurch kann allerdings eine einfache Möglichkeit geboten werden die Implementation reproduzierbar zu testen.

Als nächstes müssen alle Threads auf ihre Migrationsfähigkeit geprüft und es muß entschieden werden, wie mit nicht migrierbaren Threads verfahren wird. Befindet sich einer der Anwendungsthreads in einer nativen Methode, so muß mit der Durchführung der Migration gewartet werden, bis er diese Methode verlassen hat. Das Warten kann auf *globaler* Ebene geschehen, d. h. kein Thread wird angehalten, bis nicht für alle Threads eine Migration möglich ist. Es ist allerdings auch möglich auf *lokaler* Ebene zu warten, das bedeutet: alle Threads, die die Migrationsbedingung erfüllen werden unterbrochen, alle anderen Threads dürfen weiter laufen, bis sie migrierbar sind.
Lokales Warten ist einfach zu implementieren, da Threads immer nur einzeln, losgelöst von der restlichen Anwendung betrachtet werden müssen. Es ist allerdings möglich, das die VM sich verklemmt. Dies würde auftreten, wenn Thread **A** einen Monitor hält und bereits angehalten wurde, Thread **B** zum Verlassen der nativen Methode aber auf diesen Monitor angewiesen ist. Da Thread **A** nie mehr die Möglichkeit hat den Monitor freizugeben, Thread **B** aber nicht angehalten werden kann, entsteht eine Verklemmung und die Migration schlägt fehl.

Globales Warten hat den Nachteil, daß bereits migrierbare Threads möglicherweise native Methoden betreten und somit zu nicht migrierbaren Threads werden. Geschieht dies mit hinreichend hoher Häufigkeit, so kann die Migration auch bei *globalem Warten* fehlschlagen.

Zur Identifikation unterbrechbarer Threads kann ein Flag in einer der Threadstrukturen herangezogen werden. Nicht migrierbare Threads wie der Garbage Collector können hier beim Erzeugen markiert werden. Für das Unterbrechen der Abarbeitung der Threads ist es nötig, migrationsspezifischen Programmcode in die Interpreterschleife der VM einzufügen. So könnte z. B. vor der Ausführung eines CVM-Opcodes jedes Mal geprüft werden, ob eine Migration gewünscht ist. Somit könnte ein Thread jederzeit unterbrochen werden. Dieses Vorgehen würde die VM allerdings stark verlangsamen, da für jede interpretierte Instruktion zusätzliche Abfragen ausgeführt würden. Ein besserer Ansatz wäre also die Überprüfung nur innerhalb bestimmter Instruktionen (z. B. Methodenaufrufe) durchzuführen. Eine der Bedingungen an *sichere Punkte* für die Unterbrechung eines Threads ist, daß der Typ jedes Elements auf dem Stack bekannt sein muß. Das bedeutet allerdings, daß der Verifier diese Informationen aufbewahren und der Interpreterschleife zur Verfügung stellen muß. Eine der besonderen Eigenschaften der PNet-VM ist allerdings, daß der MSIL-Code in eine eigene Repräsentation überführt wird. Diese wird durch den Coder beim Laden von Klassen durchgeführt. Der Verifier der PNet-VM ist in den Coder integriert. Daher bietet es sich an, eine zusätzliche CVM-Instruktion zu definieren. Diese Instruktion wird durch den Coder bei ausgewählten MSIL-Instruktionen (z. B. bei Methodenaufrufen) generiert und erhält als Parameter die aktuell auf dem Stack befindlichen Typinformationen. Dieses Vorgehen ist in Abbildung 13 beispielhaft dargestellt. Die zusätzliche Instruktion kann dann drei Aufgaben erfüllen: Das Überprüfen, ob eine Migration durchgeführt werden soll, gegebenenfalls das Anhalten des aktuellen Threads und zu guter Letzt das Verfügbarmachen der Typinformationen. Dieses Vorgehen erfüllt also alle Bedingungen an einen *sicheren* Punkt. Ein weiterer Vorteil dieser Lösung ist, daß die Generierung der neuen Instruktion unterbunden werden kann, sollte die VM nicht im Migrationsmodus laufen. Dies erhöht die Verarbeitungsgeschwindigkeit, wenn nicht migriert werden soll.

Wurden die migrierbaren Threads angehalten, so muß ein in der VM enthaltener Dienst die Zustände der Threads auslesen und in das Abbild schreiben. Es wäre beispielsweise möglich, diese Aufgabe durch den letzten aktiven Thread durchführen zu lassen. Hierfür müsste allerdings der Zustand

C#

```
class Test {
  void Main() {
    Add(12,34);
  }

  void Add(int a, int b) {
    int i = a+b;
  }
}
```

CVM

```
[...]
ldc_i4_s       12
ldc_i4_s       34
op_mig         0x10316cf0
call           void Test::Add(int32 a, int32 b)
set_num_args   2
ckheight
mk_local_1
iload_0
iload_1
iadd
istore_2
return
[...]
```

Abbildung 13: CVM Code mit eingefügter Migrationsinstruktion.

des Thread verändert werden. Dies birgt die Gefahr, daß Veränderungen für das ausgeführte Programm sichtbar werden. Eine andere Möglichkeit besteht darin, einen zusätzlichen Thread zu erzeugen, welcher die Steuerung der Migration übernimmt. Dieser Thread muß dann allerdings (ähnlich dem Garbage Collector) von der Migration ausgenommen werden. Der Vorteil dieses Verfahrens ist, daß Mechanismen um die Migration eines Threads zu verhindern bereits für den Garbage Collector existieren müssen.

Für das Abbild kann ein spezielles Dateiformat definiert werden. Dies müsste Definitionen für alle möglichen Datentypen enthalten. Alternativ kann die Serialisierung des .NET Frameworks benutzt werden. Diese ermöglicht es, beliebige Objektbäume in einer Datei zu speichern. Der Vorteil der Serialisierung ist, daß alle Datentypen automatisch korrekt gehandhabt werden und Abhängigkeiten eines Objektes mit gespeichert, bzw. beim Einlesen vom System restauriert werden. Da für jede Instanzmethode eines Objektes sichergestellt ist, daß auch die `this`-Referenz des Objektes in den Parametern enthalten ist, wird durch die Serialisierung der drei Stackelemente der gesamte erreichbare Objektbaum der Anwendung gesichert. Ein Nachteil der Serialisierung ist, daß diese nur explizit als serialisierbar markierte Klassen speichert. Dieser Mechanismus soll sicherstellen, das der Programmierer einer Klasse sich bewußt ist, das die Klasse durch Serialisierung persistent gemacht werden kann. Dieser Schutzmechanismus muß für die Migration umgangen werden.

Die Funktionsfähigkeit von Threadmethoden wie `Join()` oder `Start()` nach der Migration kann durch Veränderungen in der VM sichergestellt werden. Alternativ kann die Implementation der notwendigen Programmlogik auch in der Klasse STThread erfolgen.

Um die von einem Thread gehaltenen Monitore zu erhalten, gibt es zwei Möglichkeiten. Entweder muß diese Information aus den erreichbaren Objekten extrahiert werden, oder die Information wird in einer leicht auszulesenden Form in einer der Threadstrukturen mitprotokolliert. Die erste Variante hat den Vorteil, daß zur Laufzeit kein zusätzlicher Overhead zum Pflegen einer redundanten Information entsteht. Dafür ist der Aufwand während der Suspendierung sehr groß, da erst alle erreichbaren Objekte identifiziert und danach die gehaltenen Monitore der Threads ausgelesen werden müssen. Die zweite Variante ermöglicht zum Suspendierungszeitpunkt einen schnellen Zugriff auf die gehaltenen Monitore eines Thread, dafür entsteht zur Laufzeit ein Overhead zum pflegen der redundanten Monitorinformationen.

Eine Anwendung hängt durch Ressourcen, die sie verwendet, immer auch

von ihrer Umgebung ab. Bei Dateioperationen liegen Teile des E/A-Zustands nicht in der VM, sondern in der darunterliegenden Betriebssystemfunktionalität. Als Beispiel sei hier der Schreib-/Lesezeiger innerhalb einer Datei genannt. Um Dateioperationen vollständig transparent zu machen, müsste dieser entweder aus dem Betriebssystem in die VM verlagert werden, was die Geschwindigkeit der E/A-Operationen verlangsamen würde, oder der Schreib-/Lesezeiger müsste mit Hilfe geeigneter Funktionen ausgelesen werden (z. B. `lseek()`). Dies ist aber im Falle von Pipes oder anderen speziellen Dateien nicht immer möglich. Zusätzlich zu den vorgenannten Einschränkungen müssen alle vom migrierten Programm benutzten Dateien auch auf der Zielmaschine unter einem identischen Pfad erreichbar sein. Dies ließe sich mit Hilfe von Netzwerkdateisystemen wie NFS oder SMB erreichen.

Ein weiteres Problem sind Netzwerkverbindungen, welche an einen bestimmten Rechner gebunden sind und sich deswegen nicht wie Dateien beliebig auf mehreren Rechnern verfügbar machen lassen. Diese Problematik könnte durch den Einsatz eines Proxy umgangen werden. Die Kommunikation nach außen würde für das Programm vollständig durch den Proxy übernommen und die so gewonnene virtuelle Verbindung könnte dann durch den Proxy von der Migrationsquelle auf das Ziel umgeleitet werden. Da dieser Lösungsansatz auch für Dateioperationen funktionieren würde, ließe sich über diesen Proxy die gesamte Ein-/Ausgabe transparent für das Anwendungsprogramm umleiten. Da bei der PNet VM sämtliche Ein-/Ausgabeoperationen über einige wenige Klassen (`Platform.FileMethods`, bzw. `Platform.SocketMethods`) vom Managed- zum Native-Code abgebildet werden, lassen sich E/A-Operationen relativ leicht durch Proxy-Implementationen dieser Klassen umleiten.

In Abbildung 14 ist schematisch aufgezeichnet, wie ein solcher E/A-Proxy arbeiten könnte. Programm 1 und 2 enthalten statt der eigentlichen Implementation für E/A-Operationen nur noch Stubs. Diese Stubs stellen eine Verbindung zum E/A-Proxy her und reichen jede E/A-Operation direkt zum Proxy durch. Dieser enthält somit alle Zustandsinformationen über die benutzten Ressourcen. Sind die Stubs entsprechend fehlertolerant implementiert, so können sie nach einer Migration erneut die Verbindung zum E/A-Proxy herstellen und so die E/A-Kanäle für die Anwendung transparent weiterleiten. Der Nachteil einer Proxylösung ist, das ein Ausfall des Proxys einen Verlust sämtlicher E/A-Kanäle bedeutet.

Die in Kapitel 3.1, Seite 28 angesprochenen Zeitprobleme lassen sich verhindern, in dem die Zeitbasis der an der Migration beteiligten Rechner ab-

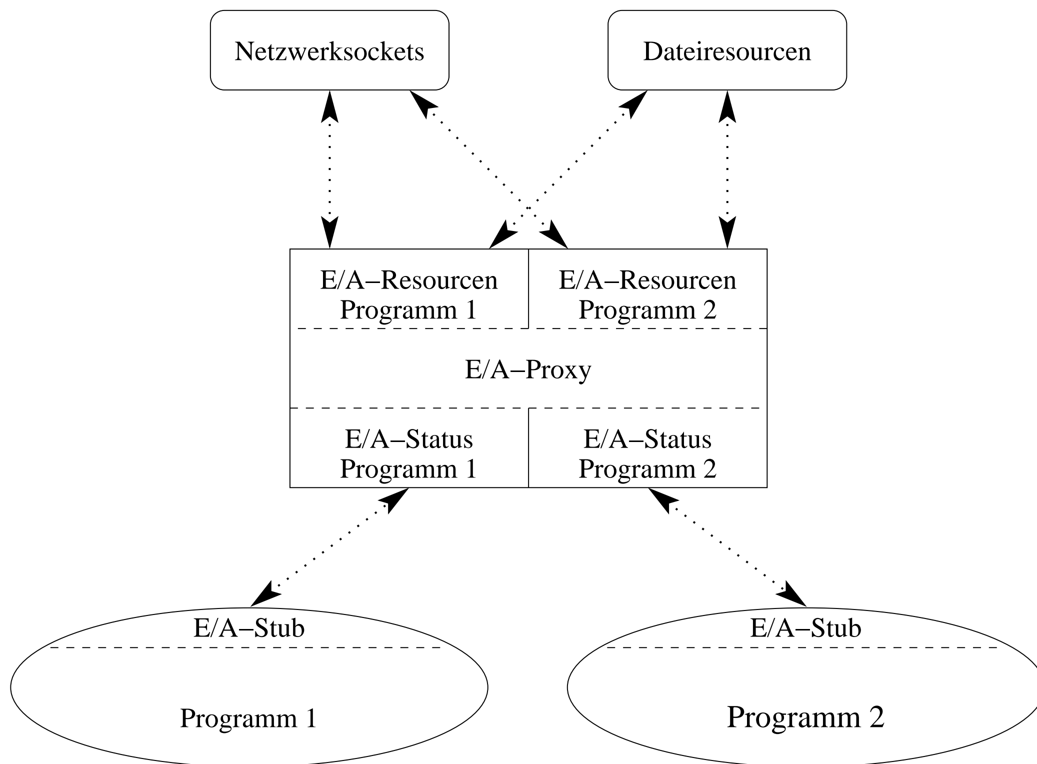

Abbildung 14: E/A-Proxy mit zwei Programmen

geglichen wird, die Zeitzonen der Rechner identisch sind und die VM intern nur mit *Universal Coordinated Time* (UTC) arbeitet.

Wurde eine Anwendung erfolgreich suspendiert, so müssen alle Bestandteile der Anwendung (Abbild und Programmcode) auf den Zielrechner übertragen werden. In einer einfachen Variante kann ein existierendes Netzwerkdateisystem verwendet werden. Die Quell-VM legt das Abbild auf einem Netzwerklaufwerk ab von welchem die Ziel-VM es einliest.

Diese Lösung wäre einfach zu verwenden, da das Laufwerk neben dem Abbild auch den Programmcode enthalten kann. Eine andere Variante wäre das Abbild über eine Socketverbindung direkt an eine wartende VM zu übertragen. Hier müsste der Programmcode mit in das Abbild integriert werden.

Auch in der Ziel-VM muß eine Kontrollinstanz für den Restaurationsprozeß existieren. Hierfür kann, wie bei der Suspendierung, ein zusätzlicher Thread erzeugt werden. Alternativ übernimmt der beim Starten der VM erzeugte Mainthread die Steuerung der Restauration. Da der Mainthread in der Ziel-VM nach der Restauration keine Aufgabe hat, seine Aufgabe wird ja durch den restaurierten Mainthread aus dem Abbild übernommen, steht dieser zur Verfügung.

3.3 Vorstellung der ausgewählten Lösungen

Von den oben vorgestellten Lösungsansätzen wurden die folgenden implementiert:

Die *Migrationsanforderung* wird entweder über das Unix-Signal `SIGUSR1` oder über einen Methodenaufruf (`Thread.FreezeVm()`) an die VM kommuniziert. Hierdurch ist es berechtigten Benutzern leicht möglich eine laufende VM zu migrieren, andererseits kann die Implementation der Migration leicht und wiederholbar getestet werden.

Ist ein Thread *nicht migrationsfähig*, so wird durch *lokales Warten* versucht die Migration durchzuführen. Die Wahrscheinlichkeit, daß eine Verklemmung in der VM entsteht, wird als zu niedrig angesehen, als daß ein erhöhter Implementationsaufwand für *globales Warten* gerechtfertigt wäre.

Nicht migrierbare Threads werden beim Erzeugen markiert und somit bei einer Migration nicht berücksichtigt.

Die *Unterbrechung der Threads* wurde durch die Implementation einer zusätzlichen CVM-Instruktion gelöst. Diese Lösung bietet alle für einen *sicheren Punkt* notwendigen Bedingungen und ermöglicht gleichzeitig eine hohe Arbeitsgeschwindigkeit der VM. Es wird nur bei Methodenaufrufen migriert. Zum Zeitpunkt eines Methodenaufrufs ist sichergestellt, daß sich keinerlei CVM spezifische Zwischenergebnisse auf dem Stack befinden. Zudem sind die Inhalte der drei Stacks zu diesem Zeitpunkt wohldefiniert. Der Inhalt der Stacks entspricht dem gestrichelten Bereich in Abbildung 3. Die Typen der Parameter der aktuellen Methode sind durch die Signatur der Methode definiert und liegen, wie die Typen der lokalen Variablen, in den Metadaten der Methode vor. Die Parameter der aufzurufenden Methode sind somit auch bekannt. Einzig die Typen der Zwischenergebnisse auf dem Berechnungsstack liegen in der VM nicht direkt vor und müssen über die zusätzliche Instruktion gewonnen werden.

Das *Auslesen des Zustandes und Erzeugen des Abbildes* erfolgt mit Hilfe eines gesonderten Migrationsthread. Diese Lösung setzt auf bereits existierenden Mechanismen auf, ist einfach zu implementieren und bietet eine hohe Flexibilität.

Für das *persistente Speichern des Abbildes* wird die im .NET Framework enthaltene Serialisierung benutzt. Dadurch wird die Klasse der unterstützten Datentypen auf die *leicht migrierbaren Datentypen* beschränkt, dafür ist keinerlei zusätzlicher Implementationsaufwand für das Erzeugen des Abbildes notwendig. Weiterhin werden im Abbild nur *lokal* erreichbare Daten und

kein Programmcode gespeichert. Für den Nachweis der Funktionsfähigkeit der Migration sind die *globalen* Daten nicht zwingend erforderlich.

Sonderbehandlungen für Threadmethoden wurden in der Klasse STThread der Laufzeitbibliothek implementiert.

Die *Monitorinformation* der Threads wird in Form einer assoziativen Hashtabelle in der Threadstruktur `ILExecThread` gepflegt und steht somit zum Suspendierungszeitpunkt direkt zur Verfügung. Das aufwendige und fehleranfällige *Aufsammeln* der Informationen entfällt. Die einhergehenden Geschwindigkeitseinbußen werden in Kauf genommen.

Um im Rahmen dieser Arbeit eine funktionsfähige Lösung zu erreichen werden die *Ressourcenabhängigkeiten eines Programmes* in der Implementation nicht berücksichtigt.

Die *Übertragung* des Abbilds erfolgt entweder durch den Benutzer, oder Programmcode und Abbild befinden sich auf einem Netzlaufwerk und sind somit auf allen beteiligten Rechnern verfügbar.

Für die *Kontrolle der Restauration* wird der Mainthread verwendet, da es für die Applikation unerheblich ist, ob der restaurierte Mainthread und der erste gestartete Thread einer VM identisch sind.

Zum Erzeugen *fortsetzbarer Threads* wurden Mechanismen in der VM vorgesehen, die leere Threadhüllen erzeugen und mit Zustandsinformationen füllen können. Beim Neustart werden die Initialisierungsmechanismen der Interpreterschleife umgangen.

4 Implementation

In diesem Kapitel soll auf Details der Implementation der Migration eingegangen werden.

4.1 Veränderungen an der VM

Die Kommandozeilenschnittstelle der VM wurde um zwei Parameter erweitert. Wird die VM mit dem Parameter `--migration-vm` (kurz `-X`) gestartet, so wird ein zusätzlicher Thread erzeugt, der die Steuerung der Migration übernimmt. Zusätzlich wird die VM in der Prozeßstruktur als migrierbar markiert, hierdurch können die verschiedenen, an der Migration beteiligten Komponenten der VM feststellen, daß es sich um eine migrationsfähige VM handelt. Zum Schluss wird eine Behandlungsroutine für das Unix-Signal `SIGUSR1` installiert. Diese Routine setzt beim Auftreten des Signals das Flag für die Migration.

Die internen Strukturen der VM wurden um mehrere Felder erweitert:

- Die Prozeßstruktur wurde um einen VM-Typ (normale VM, Migrations-VM) und einen VM Status (normaler Betrieb, Migration gestartet) ergänzt.
- Der Ausführungsrahmen erhielt zusätzlich die letzten gesicherten Typinformationen und den Startzeiger des Rahmens.
- Zur Threadstruktur wurde ein Hash mit gehaltenen Monitoren, ein Flag, welches die Migrationsfähigkeit angibt, der Migrationszustand des Thread und der Zeiger auf die zuletzt gefundene Typinformation hinzugefügt.
- Die Methodenstruktur wurde um die Typinformationen der lokalen Variablen und der Parameter vergrößert.

Da die Signaturinformationen nach dem Laden nicht mehr direkt verfügbar sind, hat der Coder beim Übersetzen zusätzlich die Aufgabe erhalten, die Typinformation der Methodensignaturen und lokalen Variablen in dem gleichen Format wie für den Ausführungsstack aufzubereiten und in der Methodenstruktur `ILMethod` zu vermerken.

Wie bereits in Kapitel 3.3 erwähnt, wurde im Interpreter der VM ein zusätzlicher Opcode implementiert (`COP_MIG`). Dieser erhält als Parameter

einen Zeiger auf eine Beschreibung der Typinformationen für seine aktuelle Position innerhalb der Methodenausführung. Hierfür mußte sowohl der Coder als auch die Interpreterschleife angepaßt werden. Der Coder erzeugt bei jedem Methodenaufruf diesen neuen Opcode, die Interpreterschleife wurde um die Behandlung dieses Opcodes erweitert. Der zusätzliche Opcode erhält einen Zeiger auf eine Struktur mit den Typinformationen als Parameter. Der Verifier innerhalb des Coders stellt in der modifizierten Form die Typinformationen für den Ausführungsstack zur Verfügung und vermerkt die Typen der lokalen Variablen, sowie die Methodensignaturen in der Struktur ILMethod. In der Interpreterschleife mußte die Struktur für die Ausführungsrahmen um die zusätzlichen Typinformationen erweitert, sowie die Makros zur Verwaltung des Aufrufstack angepaßt werden. Die Interpreterschleife wurde so modifiziert, daß sie zur Fortsetzung eines Programmes innerhalb des neuen Opcodes erneut betreten werden kann. Die vollständige Typinformation für den Ausführungsstack ist nötig, um einerseits den Inhalt aus C# heraus auslesen und serialisieren zu können. Andererseits ist es nur so möglich, nicht migrierbare Datentypen wie Zeiger zu erkennen. Stößt die VM zur Laufzeit auf diesen Opcode, so überprüft sie das Migrationsflag in der Prozeßstruktur und im Falle eines Migrationswunsches hält sie den Thread an. Neben dem Unterbrechen der Ausführung bei Migrationswunsch erfüllt die zusätzliche CVM-Instruktion noch den Zweck, die Typinformationen beim Aufruf einer Methode im erzeugten Aufrufrahmen zu speichern. Nur so ist sichergestellt, daß für jeden Rahmen auf dem Aufrufstack auch immer die Typinformation des Stackinhaltes vorliegt.

Der PC wird bei der Extraktion nicht als Zeiger, sondern als Offset in die vom Coder erzeugten Daten gesichert. Dies ist nötig, da nicht garantiert ist, daß der Coder auf dem Zielsystem den Code der Methode an die gleiche Stelle im Speicher generiert wie auf der Quellmaschine. Es ist allerdings sichergestellt, daß er für die selben Quelldaten den gleichen CVM-Code generiert, weswegen sich aus dem gespeicherten Offset und dem Methodenbeginn der korrekte PC auf der Zielmaschine bestimmen läßt. Wurden alle Zustandsinformationen eines Ausführungsrahmen extrahiert, folgen die dem Rahmen zugeordneten Stackinhalte. Zuerst werden alle Berechnungsergebnisse eines Rahmens extrahiert. Danach werden die lokalen Variablen und die Parameter gesichert. Da nicht alle Objekte durch das *Serializable*-Attribut als serialisierbar ausgewiesen sind, mußte die Laufzeitbibliothek so modifiziert werden, daß sie im Migrationsmodus dieses Attribut nicht abprüft.

In der vorliegenden Implementation wird das Abbild im Programmver-

zeichnis unter dem Namen der gestarteten Applikation erzeugt. Dem Applikationsnamen wird dabei die Endung `.vmstate` hinzugefügt. Ist eine Applikation also von einem Netzlaufwerk gestartet worden, so ist das Abbild nach erfolgter Suspendierung auf diesem Laufwerk verfügbar.

Der Parameter `--restore-vm` (kurz `-Y`) startet die VM im Restaurationsmodus. Hier betritt der Mainthread der VM nicht die Startmethode der auf der Kommandozeile angegebenen Assembly, statt dessen liest er das zu der Assembly gehörende Abbild ein, restauriert es und übergibt dann die Kontrolle den restaurierten Threads.

Zuerst deserialisiert der Migrationsthread das Abbild. Hierbei wird der Objektbaum der im Abbild gespeicherten Anwendung wieder hergestellt. Als nächstes werden alle Threads der Anwendung restauriert. Die STThreads werden beim Deserialisieren über einen speziellen Konstruktor erzeugt. Dieser Konstruktor bekommt ein `SerializationInfo` Objekt übergeben, welches bei der Extraktion gesichert wurde. Nach dem Erzeugen werden alle Threads durch den Migrationsthread mit Zustandsinformationen gefüllt. Zuerst werden die öffentlich zugänglichen Attribute wie Threadname und Hintergrundausführung gesetzt. Danach wird die nötige Anzahl von Ausführungsrahmen erzeugt. Nun werden für jeden Rahmen zuerst die Rahmeninformationen wie die ausgeführte Methode oder der PC gesetzt. Zum Schluß werden für jeden Frame die Inhalte seiner Berechnungsstacks (Parameter, lokale Variablen und Zwischenergebnisse) restauriert. Sind die Stackinhalte wieder hergestellt, werden alle dem Thread zugewiesenen Monitore wieder belegt und die Restauration ist abgeschlossen.

Wurden alle Threads erfolgreich restauriert, so startet der Migrationsthread die Threads über eine gesonderte Methode (`ReStart()`) neu. Die `ReStart()` Methode stellt sicher, das die Interpreterschleife ihre Initialisierungen unterdrückt und direkt hinter die eingefügte Migrationsinstruktion verzweigt. Hierdurch wird die Ausführung direkt an der Unterbrechungsstelle wieder aufgenommen.

Die Restauration der internen VM Zustände ist nicht vollständig umgesetzt. Deshalb wird die mehrfache Migration nicht unterstützt. Eine Applikation, die wiederhergestellt wurde, kann dadurch nicht erneut migriert werden. Die Initialisierung der fehlenden Zustandsinformationen wurde aus Zeitgründen nicht implementiert. Aus diesem Grund ist die vorliegende Implementation für den Einsatz z. B. in Agentensystemen noch nicht geeignet.

4.2 Datenstrukturen des Abbildes

Wie in Kapitel 3.3 beschrieben wird das Abbild durch normale Serialisationsmechanismen der Klassenbibliothek erzeugt. Neben dem Objektbaum der Anwendung werden auch alle Zustandsinformationen in dafür geschaffene Objekte gespeichert und diese dann serialisiert. Die dafür genutzten Objekte sollen hier kurz beschrieben werden.

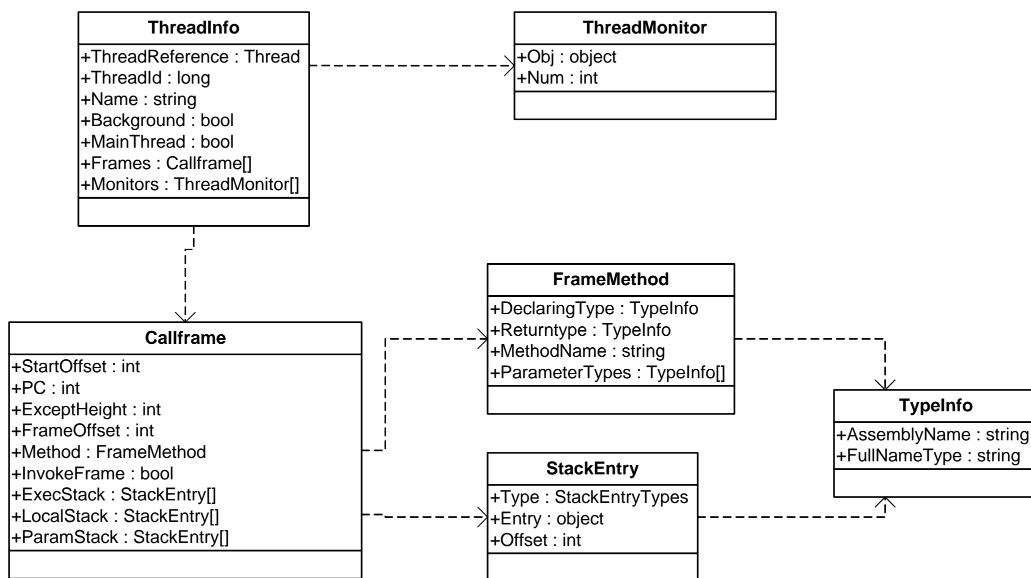

Abbildung 15: Klassendiagramm des Abbilds

Ein Array von `Remote.ThreadInfo` Objekten bildet die oberste Hierarchieebene beim Sichern des VM Zustands. Diese Objekte enthalten alle Informationen die mit einem Thread assoziiert sind:

- `System.Threading.Thread ThreadReference`: Das Threadobjekt welches den aktuellen Thread innerhalb der original VM repräsentiert hat.
- `long ThreadId`: Eine eindeutige Id für diesen Thread.
- `String Name`: Der Name des Thread.
- `bool Background`: Enthält `true`, wenn es sich um einen Hintergrundthread gehandelt hat.
- `bool MainThread`: Enthält `true`, wenn es sich um den initialen Thread der original VM handelt.
- `CallFrame[] Frames`: Ein Array von Ausführungsrahmen.
- `System.Threading.ThreadMonitor[] Monitors`: Ein Array mit allen Monitoren, die dieser Thread gehalten hat.

Ein `System.Threading.ThreadMonitor` enthält die Informationen, die nötig sind, um auf der Ziel VM alle Synchronisationszuständen herzustellen:

- `Object Obj`: Das Objekt, auf dem der Thread einen Monitor hält.
- `int Num`: Die Anzahl, wie oft der Thread den Monitor belegt hat.

Ein `CallFrame` beinhaltet alle Informationen eines Ausführungsrahmen:

- `int StartOffset`: Ein Offset in Stackworten, welches den Abstand vom Anfang des Stacks beschreibt.
- `int PC`: Der aktuelle PC innerhalb dieses Rahmen als Offset in die übersetze Methode.
- `int ExcHeight`: Ein Wert aus der internen VM Struktur, welcher zum Behandeln von Ausnahmen dient.
- `int FrameOffset`: Das Offset in den aktuellen Rahmen.

- `FrameMethod Method`: Ein Objekt, welches die aktuell ausgeführte Methode beschreibt.
- `bool InvokeFrame`: Enthält `true`, wenn es sich bei diesem Rahmen um den initialen, nativen Rahmen beim Starten eines Thread handelt. Alle anderen Informationen in diesem Objekt sind dann ungültig.
- `StackEntry[] ExecStack`: Ein Array von Stackeinträgen für den Berechnungsstack.
- `StackEntry[] LocalStack`: Ein Array von Stackeinträgen für den Stack der lokalen Variablen.
- `StackEntry[] ParamStack`: Ein Array von Stackeinträgen für den Stack der übergebenen Parametern.

Eine `FrameMethod` beschreibt die Klasse, aus der eine Methode stammt. Diese Klasse ist nötig, da das Serialisieren von `System.Reflection.MethodInfo` Objekten in der PNet Laufzeitbibliothek noch nicht funktionsfähig ist:

- `TypeInfo DeclaringType`: Die Beschreibung der deklarierenden Klasse.
- `TypeInfo ReturnType`: Die Beschreibung des Rückgabewertes der Methode.
- `String MethodName`: Der Name der Methode.
- `TypeInfo[] ParameterTypes`: Ein Array mit Typbeschreibungen der Parameter.

Die Klasse `TypeInfo` beschreibt einen Datentyp. Dies ist nötig, da das Serialisieren von `System.Type` Objekten in PNet nicht implementiert ist:

- `String AssemblyName`: Der Name der Assembly, die die Klasse enthält.
- `String FullTypeName`: Der voll qualifizierte Name der Klasse.

In der Klasse `StackEntry` ist ein einzelner Eintrag auf dem Stack gespeichert:

- `StackEntryTypes Type:` Eine Aufzählung, welche den Typ des Eintrags bezeichnet.
- `Object Entry:` Der Stackeintrag, native Datentypen sind gekapselt.
- `int Offset:` Das Offset des Eintrages in dem Rahmen.

4.3 Veränderungen an der Klassenbibliothek

Die .NET Klassenbibliothek wurde an zwei Stellen verändert: Der `mscorlib.dll` und der `System.dll`. In der `mscorlib.dll` wurden so wenig Änderungen wie möglich durchgeführt. Es wurden einige Enumerations mit Definitionen von Konstanten aus der VM und die Klasse `System.Threading.ThreadMonitor` hinzugefügt. Alle anderen neuen Klassen wurden in dem neuen Package `Remote` in der `System.dll` definiert. Siehe hierzu auch Kapitel 4.2.

Wie bereits in Kapitel 2.4 erwähnt, wird ein Thread innerhalb der VM durch mehrere Datenstrukturen beschrieben. Für die Migration ist die Zuordnung zwischen dem STThread und den dazugehörigen VM-Strukturen nur während der Extraktion und der Restaurierung wichtig. Der STThread wird zusammen mit allen anderen erreichbaren Objekten serialisiert. Um die internen Variablen eines Threadobjektes migrieren zu können wurde die Klassen STThread um das Interface *ISerializable* erweitert. Durch diese Erweiterung und die Verlagerung des Threadstatus aus der VM in das Threadobjekt ermöglicht der Methode `Join()` auch nach der Migration noch das gewünschte Verhalten zu liefern.

Alle Schnittstellen zum Steuern der Migration wurden innerhalb der Klasse STThread implementiert. Im Folgenden werden die hinzugefügten oder veränderten Methoden kurz mit ihrer Signatur und einer Beschreibung aufgeführt.

Allgemeine Methoden zum Feststellen des VM Zustand oder zum Steuern der Migration innerhalb von STThread:

- `public void Start(bool val):` Diese Methode existiert neben der normalen `Start()` Methode und dient zum Erzeugen eines nicht migrierbaren Threads. Hierfür muß als Parameter `false` übergeben werden. Wird `true` übergeben, so verhält sie die Methode wie die normalen `Start()` Methode.

- `public Thread(SerializationInfo info, StreamingContext context)`: Erzeugt einen STThread beim Deserialisieren.
- `public void GetObjectData(SerializationInfo info, StreamingContext context)`: Dient zum Serialisieren des Objektes.
- `extern public void ReStart()`: Startet einen Thread nach der Restauration erneut.
- `extern public void InitializeThread()`: Initialisiert einen deserialisierten Thread für den Neustart.
- `extern public void Dump()`: Dient zum Ausführen von nativem Code in der VM zum Suchen von Fehlern.
- `extern public long GetId()`: Liefert eine eindeutige Id für diesen Thread.
- `extern public bool IsFreezeable()`: Liefert `true` wenn dieser Thread angehalten werden darf.
- `extern public static Thread[] GetThreadList()`: Liefert ein Array aller laufenden Threads der VM.
- `extern public ThreadStates GetFreezeState()`: Liefert den Migrationsstatus des Thread. Mögliche Werte sind `Run`, `Freezing` und `Frozen`.
- `extern public bool IsMainThread()`: Liefert `true` wenn es sich um den Hauptthread der VM handelt.
- `extern public static VmMode GetVmMode()`: Liefert den Betriebsmodus der VM. Mögliche Modi sind `Default`, `Migration` und `Restore`.
- `extern public static void FreezeVm()`: Löst die Migration aus. Diese Methode kann benutzt werden, um die Migration an bestimmten Stellen einer Anwendung zu starten.
- `extern public static bool AllFrozen()`: Liefert `true` wenn alle migrierbaren Threads angehalten wurden.

- `extern public static void ForceExit()`: Beendet die VM nach Abschluß des Extrahierens.

Methoden zum Auslesen des Threadzustandes:

- `extern public int NumStackFrames()`: Liefert die Anzahl von Ausführungsrahmen dieses Threads.
- `extern public int GetPcOffset(int frame)`: Liefert den PC für den angegebenen Rahmen.
- `extern public int GetExceptionHeight(int frame)`: Liefert den ExceptionHeight Wert des angegebenen Rahmen.
- `extern public int GetFrameOffset(int frame)`: Liefert das Offset in den aktuellen Rahmen.
- `extern public int GetStartOffset(int frame)`: Liefert das Offset in Stackworten, welches den Abstand vom Anfang des Stacks beschreibt.
- `extern public ClrMethod GetMethod(int frame)`: Liefert die aktuell ausgeführte Methode des angegebenen Rahmens.
- `extern public ThreadMonitor[] GetMonitors()`: Liefert alle Monitore, die dieser Thread hält.
- `extern public int Exec_Num(int frame)`: Liefert die Anzahl der Elemente auf dem Ausführungsstack.
- `extern public StackEntryTypes Exec_EntryType(int frame, int pos)`: Liefert den VM internen Typ des Stackeintrages. Mögliche Typen sind Int32, Int64, NativeInt, Float, Double, NativeFloat oder Object.
- `extern public` *type* `Exec_EntryAs`*Type*`(int frame, int pos, out int offset)`: Liefert einen bestimmten Stackeintrag vom Ausführungsstack typisiert zurück. *Type* kann Int32, Int64, NativeInt, NativeFloat oder Object sein.
- `extern public int Locals_Num(int frame)`: Liefert die Anzahl der Elemente auf dem Stack der lokalen Variablen.

- `extern public StackEntryTypes Locals_EntryType(int frame, int pos)`: Liefert den VM internen Typ des Stackeintrages. Mögliche Typen sind Int32, Int64, NativeInt, Float, Double, NativeFloat oder Object.

- `extern public` *type* `Locals_EntryAs`*Type*`(int frame, int pos, out int offset)`: Liefert einen bestimmten Stackeintrag vom Stack der lokalen Variablen typisiert zurück. *Type* kann Int32, Int64, NativeInt, Float, Double oder Object sein.

- `extern public int Param_Num(int frame)`: Liefert die Anzahl der Elemente auf dem Parameterstack.

- `extern public StackEntryTypes Params_EntryType(int frame, int pos)`: Liefert den VM internen Typ des Stackeintrages. Mögliche Typen sind Int32, Int64, NativeInt, Float, Double, NativeFloat oder Object.

- `extern public` *type* `Params_EntryAs`*Type*`(int frame, int pos, out int offset)`: Liefert einen bestimmten Stackeintrag vom Parameterstack typisiert zurück. *Type* kann Int32, Int64, NativeInt, Float, Double oder Object sein.

Methoden zum Setzen des Threadzustandes:

- `extern public void SetMonitor(Object obj, int num)`: Restauriert eine Monitorbelegung für den aktuellen Thread.

- `extern public void CreateStackFrames(int numFrames)`: Erzeugt die passende Anzahl leerer Stackeinträge für den aktuellen Thread.

- `extern public void SetFrameData(int frame, int pc, int ExceptionHeight, int FrameOffset, int StartOffset, ClrMethod method)`: Setzt alle übergebenen Werte für den angegebenen Rahmen.

- `extern public void sExec_EntryAs`*Type*`(int frame, int pos, int offset,` *type* `val)`: Setzt den übergebenen Wert typisiert als Eintrag auf dem Ausführungsstack.

- `extern public void sLocals_EntryAs`*Type*`(int frame, int pos, int offset,` *type* `val)`: Setzt den übergebenen Wert typisiert als Eintrag auf dem Stack der lokalen Variablen.

- `extern public void sParams_EntryAs`*Type*`(int frame, int pos, int offset,` *type* `val)`: Setzt den übergebenen Wert typisiert als Eintrag auf dem Parameterstack.

Neben den oben aufgezählten Methoden wurde bei gesetztem Migrationsflag die Überprüfung des `Serializable` Attributes beim Schreiben von Objekten in der Klasse `BinaryValueWriter` außer Kraft gesetzt.

5 Bewertung der Implementation

Um abschätzen zu können, ob die vorgenommenen Modifikationen eine für den realen Betrieb benutzbare VM hervorbringen, wurden Zeit und Speicherverbrauch der modifizierten Version mit denen einer Referenzversion verglichen. Zum Schluss wurden noch exemplarisch zwei unterschiedliche Migrationen ausgemessen, um einen Eindruck von der Geschwindigkeit des Verfahrens zu vermitteln.

5.1 Testszenario

Die Testfälle waren im Einzelnen (siehe auch Anhang C für die Quelltexte der Tak-Benchmarks):

TakThreads: Dieses Testprogramm arbeitet parallel mit zwei Threads den TAK Algorithmus [2] ab. Dieser rekursive Algorithmus zeichnet sich durch eine große Menge an Rekursionen bei geringem Stackverbrauch aus. Die Startwerte für den Aufruf waren 18, 16 und 6. Die Werte wurden willkürlich gewählt.

TakMonitor: Eine modifizierte Version des vorherigen Testfalles. Hier sind beide Threads zusätzlich über einen Monitor synchronisiert.

PrimeNumbers: Dieses Programm berechnet in einem einzelnen Thread die ersten zwei Millionen Primzahlen mit einem Brute-force Verfahren.

Linpack: Ein zum Testen der PNet VM eingesetzter Benchmark aus [20]. Hierbei handelt es sich um eine modifizierte Variante des Java Linpack von [3]. Dieser Benchmark berechnet Lösungen von linearen Gleichungssystemen mit Hilfe des Gauß-Algorithmus.

Mandel: Ein frei parametrisierbares Fraktalberechnungsprogramm, welches das Mandelbrotfraktal mit einer einstellbaren Menge paralleler Threads berechnet. Das Programm wurde mehrfach mit einem, zwei, vier, acht, 16 und 32 Threads gestartet. Es wurde jeweils ein Fraktal mit 1000x1000 Bildpunkten berechnet.

Jeder der obigen Testfälle wurde in folgenden VMs gemessen:

- Auf einer Referenz VM mit gleichem CVS-Stand wie die modifizierte VM von PNet (*REF*).
- Auf der modifizierten Version der VM mit deaktiviertem Migrationsfeature (*MIG*).
- Auf der modifizierten VM mit aktiviertem Migrationsfeature, allerdings ohne eine Migration durchzuführen (*ACT*).

Alle Testergebnisse wurden auf einem Intel Pentium 4 mit 2,6 GHz unter FreeBSD 5.2.1 gewonnen. Jeder Testfall wurde zehnmal ausgeführt um Schwankungen durch Hintergrundlast auf dem Testsystem auszugleichen. Das Betriebssystem wurde ohne ACPI Unterstützung im normalen Multibenutzermodus gestartet. Beide VMs wurden mit GCC 3.3.3 ohne jegliche Optimierungen (-O0) übersetzt. Das Methoden-inlining der VM wurde bei der Kompilation nicht aktiviert. Das Deaktivieren der Optimierungen von GCC und VM sollte unvorhersehbare Effekte durch Optimierungen ausschließen.

5.2 Overhead zur Laufzeit

Da die modifizierte VM mehr Überprüfungen zur Laufzeit durchführen muß und auch beim Übersetzen der MSIL Opcodes in CVM Opcodes Mehrarbeit anfällt, ist zu erwarten, daß die migrationsfähige VM langsamer arbeitet als die Referenz VM. Zusätzlich ist zu vermuten, daß die Veränderungen den Speicherverbrauch der VM erhöhen, da zusätzliche Daten für den Fall einer Migration vorgehalten werden müssen.

Im Falle der Fraktalberechnung sollte überprüft werden, inwieweit die Anzahl der aktiven Threads Auswirkungen auf die Laufzeit und den Speicherverbrauch hat. Erwartet wurde hier, daß mit steigender Threadzahl auch die Ausführungszeiten steigen.

In allen Abbildungen wurde der Beginn der Y-Achse von Null auf einen Wert knapp unterhalb des niedrigsten Meßwert verschoben um die Unterschiede in den Meßwerten möglichst gut sichtbar zu machen.

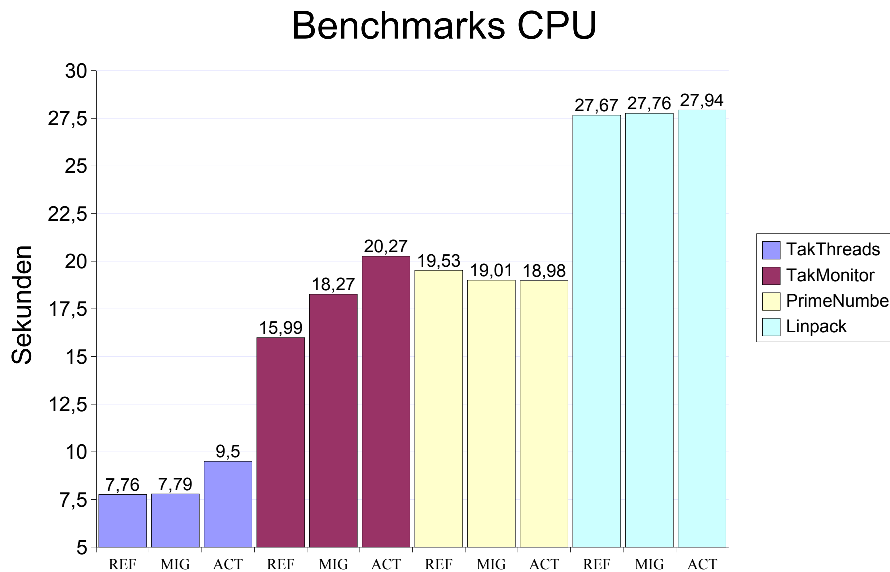

Abbildung 16: Rechenzeitstatistik für Benchmarks

Die Laufzeiten für die ersten vier der fünf Testprogramme lassen sich aus Abbildung 16 ablesen. Es ist zu sehen, daß zwei der Testfälle genau die erwarteten Werte liefern (dunkel hinterlegt), die anderen beiden Fälle sich nicht wie erwartet, bzw. genau gegenteilig verhalten (hell hinterlegt). Die im Vergleich zu den anderen Testfällen großen Unterschiede bei TakThreads und TakMonitor lassen sich zum einen damit erklären, daß der Tak Algorithmus sehr viele Methodenaufrufe ausführt (und somit auch sehr häufig den Migrationsopcode) und zum anderen die Verwaltung der Monitorinformation bei TakMonitor relativ teuer ist. Der Unterschied zwischen REF und ACT beträgt hier 22,4 %, bzw. 26,7 %. Im Fall von der Primzahlberechnung ist die Migrations VM sogar minimal (2,9 %) schneller als die Referenz VM. Dieser Effekt kann mit der durch die Testumgebung möglichen Meßungenauigkeit, den praktisch nicht vorhandenen Methodenaufrufen während der Berechnung und dem Fehlen von Synchronisation erklärt werden. Für den Linpack Benchmark steigt die Laufzeit um 0,9 %.

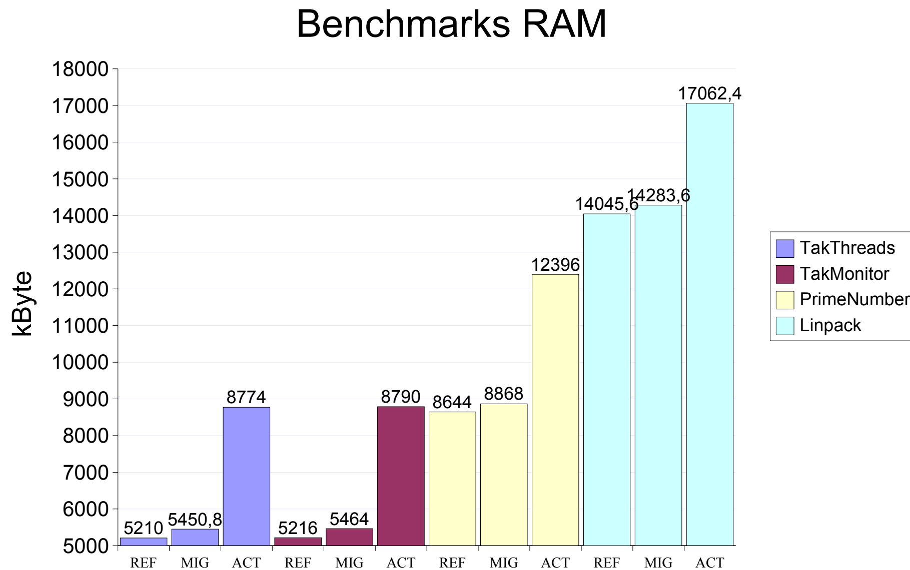

Abbildung 17: Speicherstatistik für Benchmarks

Analog zur Betrachtung der Laufzeiten wurde auch die Veränderung des Speicherverbrauch gemessen. Die Meßergebnisse lassen sich aus Abbildung 17 ablesen. Im Gegensatz zu den Laufzeiten ist das Bild hier einheitlicher. Der Anstieg im Speicherverbrauch beträgt zwischen Referenz VM und Migrations VM in etwa 240 KByte. Mit aktivierter Migrationsunterstützung steigt der Speicherverbrauch stärker an, hier sind es bis zu 3,5 MByte. Zum einen wird jetzt zusätzlich zur `mscorlib.dll` noch die `System.dll` geladen, die den Migrationscode enthält. Zum anderen wächst die Größe der übersetzten Methoden durch die eingefügten Migrationsinstruktionen und die damit assoziierten Typinformationen an. Die Prozentuale Steigerung des Speicherverbrauchs betrug 68,4 % für TakThreads und 21,5 % für Linpack.

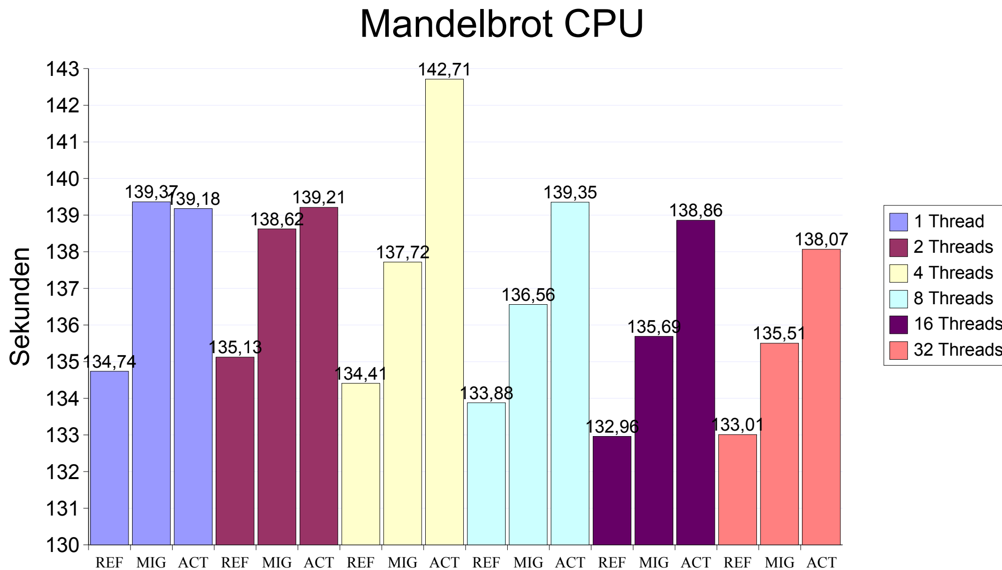

Abbildung 18: Rechenzeitstatistik für Fraktalberechnung

Mit Hilfe der Fraktalberechnung sollte die Auswirkung von mehreren Threads auf den Ressourcenverbrauch der verschiedenen VMs untersucht werden. Hierfür wurde eine nebenläufige Implementation des Mandelbrotfraktals benutzt. Die Anzahl der zu verwendenden Threads kann auf der Kommandozeile angegeben werden. Die Laufzeiten sind in Abbildung 18, die Speicherverbrauchswerte in Abbildung 19 aufgeführt. Die Laufzeiten zeigen hier die Tendenz, daß mit zunehmender Anzahl von Threads die Ausführungszeiten sinken. Dieses Verhalten kann mit der Zuteilung des Prozessors durch den Scheduler erklärt werden. Behandelt der Scheduler bei der Zuteilung des Prozessors Threads genau wie eigenständige Prozesse, so bekommt eine Applikation mit vielen Threads den Prozessor in der Summe öfter zugeteilt als eine Applikation mit wenigen Threads. Gleichzeitig ist aus der Sicht des Prozessorcache der ausgeführte Programmcode lokaler, da sich die Threads einer Anwendung den gleichen Speicher teilen. Die Verlangsamung der Berechnung liegt für diesen Testfall zwischen 3-6 %.

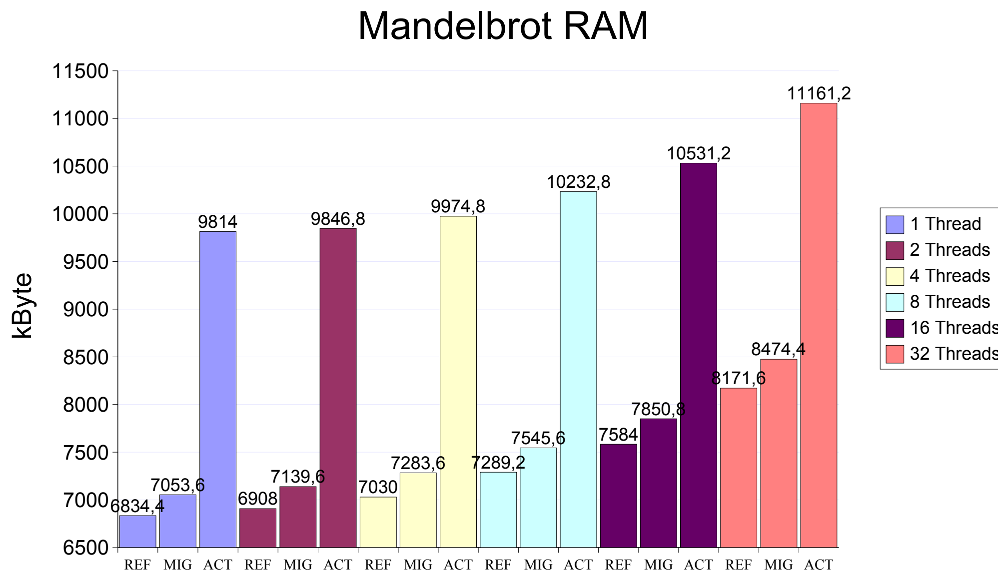

Abbildung 19: Speicherstatistik für Fraktalberechnung

Auch für die Fraktalberechnung wurde die Veränderung des Speicherverbrauchs gemessen. Hier lag der mittlere Speicheranstieg zwischen REF und MIG mit 250 KByte geringfügig höher als bei den anderen Testfällen, zwischen REF und ACT betrug er in allen Fällen genau 2900 KByte. Der Anstieg beträgt somit 43,6 % für einen und 36,6 % für 32 Threads. Dies zeigt, das der Speicherverbrauch so gut wie nicht von der Anzahl der Threads abhängt.

5.3 Messung des Migrationsvorganges

Um die folgenden Zahlen zu gewinnen wurde der Migrationscode innerhalb der `System.dll` um eine Zeitmessung erweitert. Danach wurden zwei Testfälle mit unterschiedlichen Datenmengen migriert: Eine Fraktalberechnung mit `100x100` Pixeln bearbeitet durch zwei Threads, danach der gleiche Bildausschnitt mit `5000x5000` Pixeln und acht Threads. Es wurden die folgenden Zeiten gemessen:

- Start der Migration bis zum Extrahieren der Liste der existierenden Threads.
- Auslesen der Threadzustände und Erstellen des Abbildes im Speicher.
- Schreiben des Objektbaumes durch Serialisierung.
- Lesen und Deserialisieren des Objektbaumes.
- Restaurieren des Threadzustandes.
- Neustart aller Threads.

Zusätzlich wurde noch die Größe des Abbildes auf der Festplatte festgehalten.

Anhand der Tabelle ist deutlich zu sehen, das die De-/Seralisierung am meisten Zeit benötigt. Dies ist im vorliegenden Testfall dadurch zu erklären, daß es sich um große Arrays primitiver Datentypen handelt. Der Arrayinhalt wird von den Serialisationsmechanismen elementweise in die Datei geschrieben, bzw. daraus gelesen. Dies verursacht einen großen Overhead, welcher sich in der Ausführungsgeschwindigkeit niederschlägt.

Das Neustarten der Threads wird wiederum durch die bereits laufenden Threads verlangsamt. Dies führt dazu, das mit zunehmender Threadzahl das Neustarten immer langsamer wird.

	100x100 Pixel 2 Threads	**5.000x5.000 Pixel 8 Threads**
Threadliste auslesen	1 ms	1 ms
Abbild erstellen	63 ms	207 ms
Abbild schreiben	185 ms	1.802 ms
Abbild lesen	63 ms	22.496 ms
Threads restaurieren	18 ms	85 ms
Threads neu starten	68 ms	537 ms
Summe	398 ms	25.128 ms
Dateigröße	15.788 Byte	25.014.957 Byte
Nutzdaten	10.000 Byte	25.000.000 Byte
Overhead	5788 Byte	14.957 Byte
Durchsatz	39.668,34 Bytes/s	995.501,31 Bytes/s

Abbildung 20: Zeitmessung der Migration

6 Fazit

Moderne Softwaresysteme werden häufig mit Hilfe von Sprachen, die auf virtuellen Maschinen ablaufen implementiert. Populäre Vertreter dieser Sprachen sind z.B. C# und Java. Gleichzeitig wird von diesen Systemen immer öfter gefordert, daß sie skalierbar sind (z.B. durch Hinzufügen von weiteren Rechnern), bzw. die Ausfallsicherheit gewährleistet ist. Eine Möglichkeit dies zu erreichen ist die Migration von ganzen Anwendungen zur Laufzeit. Die vorliegende Arbeit hat die Möglichkeit der Migration von laufenden C# Programmen analysiert, eine Beispielimplementation vorgestellt und die praktische Verwendbarkeit dieser Implementation untersucht.

Die Migration sollte dabei nicht durch Veränderungen an dem zu migrierenden Programm, sondern durch einen Eingriff in eine bestehende VM ermöglicht werden. Für die Untersuchung der Migration wurden dabei Kriterien wie Transparenz für das migrierte Programm und Portabilität sowie das Minimieren der Änderungen an der verwendeten VM definiert. Es wurde beispielhaft eine Implementation einer VM erstellt. Diese Implementation wurde in ihrer Funktionsweise vorgestellt und anschließend untersucht. Die Untersuchungsergebnisse zeigen, daß eine Migration von .NET Anwendungen möglich ist, ohne daß die Geschwindigkeit der ausgeführten Programme stark abnimmt. Neben der Geschwindigkeit wurde der Speicherverbrauch für die Migration jeweils zur Laufzeit und zum Migrationszeitpunkt untersucht.

Die meisten noch existierenden Limitationen der vorliegenden Implementation sind nicht grundsätzlicher Natur und lassen sich mit zusätzlichem Arbeitsaufwand beseitigen. So wurde z. B. für die E/A-Umleitung ein Lösungsvorschlag unterbreitet, der eine weitestgehende Transparenz für E/A-Operationen sicherstellen würde.

A Literaturverzeichnis

[1] Barak, Amnon ; La'adan, Oren: The MOSIX Multicomputer Operating System for High Performance Cluster Computing. In: *Journal of Future Generation Computer Systems (13) 4-5* (1998), March, S. 361–372

[2] Coffee, Peter: Tak test stands the test of time. In: *PCWeek* (1996), September, S. 91

[3] Dongarra, Jack ; Wade, Reed ; McMahan, Paul: *Linpack Benchmark - Java Version.* – URL http://www.netlib.org/benchmark/linpackjava/

[4] ECMA (Veranst.): *Standard ECMA-334 - C# Language Specification.* December 2002. – URL http://www.ecma-international.org/publications/standards/Ecma-334.htm

[5] ECMA (Veranst.): *Standard ECMA-335 - Common Language Infrastructure (CLI).* December 2002. – URL http://www.ecma-international.org/publications/standards/Ecma-335.htm

[6] Free Software Foundation, Inc.: *GNU General Public License.* – URL http://www.gnu.org/copyleft/gpl.html

[7] Fünfrocken, Stefan: *Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs).* S. 26–37. In: *Kurt Rothermel, Fritz Hohl (Eds.), Proceedings of the Second International Workshop on Mobile Agents (MA'98).* Springer-Verlag, September 1998

[8] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification.* Third Edition. Addison-Wesley Professional, Juni 2005. – ISBN 0-321-24678-0

[9] Malabarba, Scott: *A Java Virtual Machine for Mobile Programs.* – URL http://pdclab.cs.ucdavis.edu/projects/ariel/migrate_vm.html

[10] Object Management Group, Inc.: *Welcome To The OMG's CORBA Website.* – URL http://www.corba.org

[11] ROTHERMEL, Kurt ; SCHWEHM, Markus: *MOBILE AGENTS*. In: KENT, A. (Hrsg.) ; (EDS.), J. G. W. (Hrsg.): *Encyclopedia for Computer Science and Technology*, New York: M. Dekker Inc., 1998

[12] STEVENS, W. R.: *Advanced Programming in the UNIX Environment*. Addison Wesley Longman, Inc., 1993. – 449–474 S. – ISBN 0-201-56317-7

[13] STEVENS, W. R.: *Advanced Programming in the UNIX Environment*. Addison Wesley Longman, Inc., 1993. – 263–324 S. – ISBN 0-201-56317-7

[14] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. 2., überarbeitete Auflage. Pearson Studium, München, 2003. – 576–578 S. – ISBN 3-8273-7019-1

[15] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. 2., überarbeitete Auflage. Pearson Studium, München, 2003. – 131–137 S. – ISBN 3-8273-7019-1

[16] THE MONO GROUP: *What is Mono?*. – URL http://www.mono-project.com/Mono:About

[17] TRÖGER, Peter: *Aspect-oriented object and component migration*, Humboldt-Universität Berlin, Diplomarbeit, September 2002

[18] TRUYEN, Eddy ; ROBBEN, Bert ; VANHAUTE, Bart ; CONINX, Tim ; JOOSEN, Wouter ; VERBAETEN, Pierre: Portable Support for Transparent Thread Migration in Java Departement Computerwetenschappen, K.U.Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium (Veranst.), URL http://www.cs.kuleuven.ac.be/~eddy/BRAKES/MA2000Final.pdf, 2000

[19] WEATHERLEY, Rhys: *DotGNU Portable.NET*. – URL http://www.southern-storm.com.au/portable_net.html

[20] WEATHERLEY, Rhys: *PNetMark benchmarking tool*. – URL http://www.southern-storm.com.au/download/pnetmark-0.0.6.tar.gz

[21] Weatherley, Rhys ; V, Gopal: *Design of the Portable.NET Interpreter*. January 2003. – URL http://www.southern-storm.com.au/download/pnet-engine.pdf

[22] Whittington, Jason: Rotor - Shared Source CLI Provides Source Code for a FreeBSD Implementation of .NET. In: *MSDN Magazine* (2002), July. – URL http://msdn.microsoft.com/msdnmag/issues/02/07/SharedSourceCLI/

[23] Wilson, Paul R.: Uniprocessor Garbage Collection Techniques. In: *Proceedings of the International Workshop on Memory Management* Bd. 637. Springer-Verlag, 1992, S. 1–42

B Glossar

.NET Framework Eine von Microsoft entwickelte Programmier- und Laufzeitumgebung für plattform- und sprachunabhängige Programme.

Abbild Das Abbild enthält alle Zustandsinformationen und alle Daten einer zu migrierenden Applikation. In der vorliegenden Implementation ist hier allerdings kein Programmcode enthalten.

Aktives Fenster Der für die aktuelle Methode sichtbare Bereich auf dem Ausführungsstack.

API Application Programming Interface - Die Schnittstelle, über die eine Applikation auf Funktionen einer Funktionsbibliothek zugreifen kann.

Aufrufstack Wird eine Methode aufgerufen, so werden die zur Ausführung nötigen Informationen der aktuellen Methode auf dem Aufrufstack gesichert um bei einer `return` Anweisung oder einer Exception in der aufgerufenen Methode die Ausführung der ursprünglichen Methode fortsetzen zu können.

Berechnungsstack Das Maschinenmodell der .NET VM kennt keine Prozessorregister wie reale Prozessoren sie besitzen. Sämtliche Berechnungen finden auf dem Berechnungsstack statt. Beim Aufrufen einer Methode wird der Berechnungsstack zusätzlich zur Parameterübergabe und zum Erzeugen der methodenlokalen Variablen benutzt.

CLI Common Language Infrastructure - Die Definition einer Laufzeitumgebung basierend auf einer VM.

Coder Eine Komponente in der PNet VM, die MSIL Opcodes in CVM Opcodes übersetzt. Sie ist vergleichbar mit dem JIT anderer VMs.

CORBA Common Object Request Broker Architecture - Ein Standard zur Implementation von verteilten Systemen.

CTS Common Type System - Ein gemeinsames Typsystem für Sprachen, die auf der CLI lauffähigen MSIL Code erzeugen wollen.

CVM Converting Virtual Machine - Die virtuelle Maschine der PNet Plattform.

DotGNU Die DotGNU Plattform besteht aus der PNet VM und DGEE, welches eine Ausführungsumgebung für Webservices anbietet.

E/A Eingabe- oder Ausgabeoperationen wie Datei- oder Netzwerkoperationen.

ECMA European Computer Manufacturers Association - Ein Standardisierungsgremium wo neben der CLI und C# auch JavaScript und andere Standards eingereicht wurden.

Extraktion Das Auslesen des VM Zustand und aller Daten einer laufenden Applikation und das Erzeugen des Abbildes.

Garbage Collector Ein Mechanismus in der VM, der unreferenzierten Speicher erkennt und ihn zur erneuten Verwendung freigibt.

Heap Auf dem Heap wird der Speicher für Objekte angefordert. Die Freigabe des Heapspeichers erfolgt durch den Garbage Collector.

JIT Just in Time Compiler - Eine Komponente einer VM, die den Zwischencode eines Programms in Instruktionen der ausführenden Plattform übersetzt.

Managed Code MSIL Code, der innerhalb und unter Kontrolle der VM ausgeführt wird.

MigVM Die modifizierte PNet VM mit Migrationsunterstützung.

MSIL Microsoft Intermediate Language - Eine Zwischensprache ähnlich dem Java Bytecode.

Native Code Maschinenabhängiger Code, der außerhalb der VM ausgeführt wird. Dies umfaßt z. B. nativ implementierte Methoden der Laufzeitbibliothek als auch externe Bibliotheken.

PC Program Counter - Ein Zeiger, der die Position der nächsten auszuführenden Prozessor oder VM Instruktion im Speicher angibt.

PNet Portable DotNet - Eine freie Implementation der ECMA Standarts 334 und 335.

Rahmen Die Rahmeninformation liegt auf dem Aufrufstack und enthält für jede Methode auf dem Aufrufstack Werte wie den PC, der bei einer `return`-Anweisung angesprungen werden soll.

Restaurierung Das Erzeugen eines VM Zustand aus einem Abbild und das nachfolgende Fortsetzen der Ausführung der Applikation.

RMI Remote Method Invocation - Java spezifische Implementation von RPC.

RPC Remote Procedure Calls - Der Aufruf von Prozeduren über Rechnergrenzen hinweg, z. B. über eine Netzwerkverbindung.

Scheduler Der Teil in einem Betriebsystem, der einem Prozeß den Prozessor zuteilt oder entzieht.

Serialisierung Das Abspeichern eines Objektgraphen in einem Datenstrom in einer Form, wo sich der Originalgraph mit allen seinen Referenzen wieder herstellen läßt.

STThread Eine Instanz von *System.Threading.Thread*.

VM Virtuelle Maschine - Ein Programm, welches einen Rechner simuliert.

C Benchmarks

Im folgenden sind kurz die selbsterstellten Benchmarks TakThreads und Tak-Monitor im Quelltext angefügt.

C.1 TakThreads

```
[...]
  // Der Eintrittspunkt des Benchmark. Dieser startet einen
  // weiteren Thread und bearbeitet dann mit beiden Threads
  // den Tak-Algorithmus.
  public static void Main() {
    ThreadStart ts = new ThreadStart(doTak);
    Thread t = new Thread(ts);
    t.Name = "tak 2";
    t.Start();
    doTak();
  }

  // Der Eintrittspunkt des Tak-Algorithmus.
  public static void doTak() {
    tak(18,16,6);
  }

  // der Tak-Algorithmus
  public static int tak(int x, int y, int z) {
    if(!(y < x)) {
      return y;
    } else {
      return tak(
        tak(x-1, y, z),
        tak(y-1, z, x),
        tak(z-1, x, y)
        );
    }
  }
[...]
```

C.2 TakMonitor

Dieser Benchmark ist identisch zu dem oben angegebenen mit folgender Ausnahme:

```
[...]
  // Ein Objekt ueber das synchronisiert werden kann.
  private static Object lockObject = new Object();

  public static int tak(int x, int y, int z) {
    if(!(y < x)) {
      return y;
    } else {
      // Synchronisation ueber das Lockobjekt bei jedem Aufruf.
      lock(lockObject) {
        return tak(
          tak(x-1, y, z),
          tak(y-1, z, x),
          tak(z-1, x, y)
          );
      }
    }
  }
[...]
```

D Übersicht über die CD-ROM

Die beigefügte CD-ROM enthält die Implementation der migrationsfähigen VM, die verwendeten Benchmarks sowie die Testfälle.

D.1 Verzeichnisinhalte

treecc Dieses Verzeichnis enthält den vom PNet Projekt verwendeten Compilergenerator. Dieser wird zwingend zum Übersetzen des `cscc` Compilers benötigt.

pnet In diesem Verzeichnis befinden sich die Quellen der VM, des Compilers und weiterer Hilfsprogramme.

pnetlib In diesem Verzeichnis befinden sich die Quellen der Laufzeitbibliotheken.

testcases Dieses Verzeichnis enthält die während der Entwicklung eingesetzten Testfälle für die Migration. Diese Programme lassen sich nur gegen die modifizierte Laufzeitbibliotheken übersetzen.

benchmarks Die in Kapitel 5 eingesetzten Testfälle befinden sich in diesem Verzeichnis.

compile.sh Dieses Script übersetzt die Laufzeitumgebung und -bibliothek und installiert eine funktionsfähige VM in das Verzeichnis `install/`.

D.2 Übersetzen der VM

Die VM kann auf einfache Art und Weise übersetzt werden. Hierfür existiert das Script `compile.sh`, welches alle beteiligten Bestandteile übersetzt und installiert. Das Script funktioniert auf Unix-ähnlichen Systemen inkl. Cygwin. Es benötigt die üblicherweise vorhandenen Entwicklungswerkzeuge wie Automake, Autoconf, Libtool, Gcc, Bison, usw.

D.3 Testfälle

Für die im Verzeichnis `testcases/` vorliegen Testfälle existieren Scripte die ein einfaches Ausführen und Auswerten sicherstellen sollen. Alle diese Scripte lassen sich ohne Kommandozeilenparameter, bzw. mit `-X` oder `-Y` zum

Migrieren oder Restaurieren starten. Sie erzeugen Logdateien der Konsolenausgaben im Verzeichnis `/tmp/`.

test1.sh: Startet den Testfall `threads.cs`. Dieser enthält einen langlaufenden Thread, welcher den TAK-Algorithmus abarbeitet. Der Mainthread ruft eine Reihe Methoden mit unterschiedlichen Parametern, Rückgabewerten und lokalen Variablen auf. Während dieser Aufrufe löst er die Migration über die in 4.3 beschriebene Methode `Thread.FreezeVm()` aus.
Dieser Testfall sollte die korrekte Berechnung der Stackoffsets für verschiedene Datentypen sicherstellen.

test2.sh: Startet den Testfall `MandelFreeze.cs`. Dieser enthält den Fraktalalgorithmus aus 5. Hat einer der Berechnungsthreads die Hälfte der Pixel berechnet, so löst er die Migration aus. Dieser Testfall verwendet `100x100` Pixel und zwei Threads und sollte eine *normale* Anwendung darstellen.

test3.sh: Identisch zu `test2.sh`, der Testfall berechnet hier allerdings `2000x2000` Pixel mit acht Threads und sollte `test2.sh` um mehr Threads und eine größere Datenmenge erweitern.

test4.sh: Dieser Testfall berechnet die ersten 1000 Primzahlen. Ähnlich wie bei `test2.sh` wird die Migration über einen Methodenaufruf ausgelöst, wenn die Hälfte der Ergebnisse vorliegt.
Dieser Testfall sollte die Migration für ein Programm mit wenigen Methodenaufrufen testen.

test5.sh: Dieser Testfall ist eine modifizierte Version von `test1.sh`. Hier wird die Migration ausgelöst, wenn der erste Thread seine Berechnung beendet hat und der Mainthread ein `Join()` auf diesem ausgeführt hat. Durch diesen Test sollte die korrekte Funktion von `Join()` sichergestellt werden.

test6.sh: Identisch zu `test1.sh`, die Migration wird hier allerdings vor dem Starten des zweiten Threads ausgelöst.
Dieser Test sollte sicherstellen, daß auch noch nicht gestartete Threads migriert werden können.

D.4 Nachvollziehen der Änderungen

Alle Änderungen an existierenden Quelldateien der Laufzeitumgebung und -bibliothek wurden durch die Kommentarzeilen „`// MIG`“ und „`// endMIG`“ eingeschlossen. Die Verwaltungsinformationen des CVS sind intakt, somit lassen sich alle Änderungen auch mit Mitteln der Versionsverwaltung identifizieren. Alle hinzugefügten Dateien der Laufzeitumgebung enthalten im Dateinamen das Kürzel „`mig`“.