Virtuelle Maschinen

15. Dezember

2006

Georg Wächter

Besondere Lernleistung im Fach Informatik

Inhaltsverzeichnis

Allgemein ... 2
Begriffsklärung ... 2
Geschichte der virtuellen Maschinen ... 2
Typen virtueller Maschinen ... 5
Virtuelle Hardware Maschinen (Virtuelle Server) ... 5
Vollständige Emulation (Hardware Virtualization) ... 6
Gemischte Emulation (Hybrid Virtualization) ... 6
Paravirtualisierung ... 6
Virtualisierung auf Hardwareebene ... 6
Virtuelle Maschinen zur Ausführung von Anwendungen ... 7
Stapelbasierte und registerbasierte Maschinen ... 9
Zwischencode ... 11
Eigenschaften virtueller Maschinen ... 13
Plattformunabhängigkeit ... 13
Geschwindigkeit ... 14
Speicherverbrauch ... 16
Sicherheit ... 19
Flexibilität ... 21
Interoperabilität ... 22
Ayox Intermediate Language ... 24
Übersicht ... 24
Abstrakter Prozessor ... 24
Garbage Collector ... 25
Beispiel ... 25
Abschluss ... 26
Anhang ... 27
Entstehung der Arbeit ... 27
Literaturverzeichnis ... 28
Abschließende Klausel ... 29

Allgemein

Begriffsklärung

Eine virtuelle Maschine bezeichnet eine Software, die eine Ausführungsumgebung für Anwendungen oder Betriebssysteme zur Verfügung stellt.

Es kann prinzipiell zwischen zwei verschiedenen Arten von virtuellen Maschinen unterschieden werden. Sogenannte virtuelle Hardware Maschinen repräsentieren nicht nur einen abstrakten Prozessor, sondern zusätzlich auch die typischen Hardwarekomponenten eines PCs. Somit können Betriebssysteme in einer abgeschotteten Umgebung auf einer einzigen Maschine laufen. Solche Softwaresysteme werden oftmals auch als virtuelle Server oder virtuelle Hardware Maschinen bezeichnet.

Desweiteren existieren virtuelle Maschinen, die nicht das Emulieren einer gesamten Hardwarekonfiguration sondern nur das eines einzigen Prozessors als Ziel haben. Das populärste Beispiel für diese Art von virtuellen Maschinen ist die Java VM.

Geschichte der virtuellen Maschinen

Schon Mitte der 60'er Jahre waren die existierenden Prozessorarchitekturen so vielfältig, dass man mit einer Programmversion nicht mehr den gesamten Markt bedienen konnte. Für den Erfolg einer Software ist die Größe des Anwenderkreises jedoch besonders wichtig. Daher war man dazu gezwungen Software für möglichst viele Plattformen zu entwickeln, um noch mehr Abnehmer für das Programm zu finden.

Die einfachste, aber auch aufwendigste, Lösung war das manuelle Portieren auf jede Zielplattform. Da es damals noch keine plattformunabhängigen Programmiersprachen gab, mussten die Quelltexte im Extremfall für jedes System angepasst und anschließend neu kompiliert werden. Daher suchte man nach Lösungen, um Softwaresysteme portabler zu gestalten.

Etwa im Jahre 1966 begann Martin Richards an der University of Cambridge mit der Arbeit an der Basic Combined Programming Language (BCPL), welches die erste portable Programmiersprache wurde. Er entwickelte als Erster einen Compiler, der Zwischencode produzierte. Im konkreten Fall der BCPL war dies der sogenannte Object-Code, auch O-Code genannt. Durch diese neuartige Eigenschaft des Compilers wurde die Sprache sehr populär, da es nun relativ einfach war ein in BCPL geschriebenes Programm auf vielen verschiedenen Plattformen auszuführen. Der O-Code wurde nun normalerweise in Maschinencode transformiert oder per Software interpretiert.

Dieses Konzept hatte gleich zwei positive Aspekte. Zum einem waren die entwickelten BCPL Programme portabel und zum anderem war auch die Entwicklung neuer Compiler einfacher geworden. Bisherige Compiler mussten alles in einem Rutsch erledigen, der Quelltext musste also direkt in den Maschinencode umgewandelt werden. Richards Konzept hingegen sah eine Zweiteilung des Compilers in den Front End und den Back End vor. Der schwierigere Teil, das Kompilieren des Quelltextes in den Zwischencode musste nur einmal entwickelt werden. Im Gegensatz zu dem Back End der für jede Plattform existiert und den Zwischencode dann letztendlich in den nativen Code kompiliert.

Zur gleichen Zeit begann IBM mit der Arbeit an ersten virtuellen Maschinen. Das vorrangige Ziel war das einfache Testen von neuen Softwarefeatures. IBM suchte nach einer Möglichkeit einzelne Hardwarekomponenten flexibel an- und auszuschalten, um so die Auswirkungen auf die Software zu überprüfen. Damit die Maschinen nicht ständig umgebaut werden mussten, kam man im IBM Yorktown Research Center auf die Idee eine virtuellen Maschine zur Simulation eines PCs zu bauen.

In den früheren 70'er Jahren kam nun der Begriff des P-Codes auf. Damit war der Zwischencode gemeint, der von Pascal Compilern erzeugt wurde. Das Prinzip dieser Compiler war jedoch das gleiche wie das von BCPL. Die bekannteste Implementation eines Pascal Systems war UCSD Pascal (University of California, San Diego). Dieser Interpreter wurde so erfolgreich, dass er Ende der 70'er Jahre sogar als realer Prozessor umgesetzt und von Western Digital angeboten wurde. Der sogenannte Pascal MicroEngine Prozessor wurde aber nicht besonders lange hergestellt, da auch bald die in Software implementierten Interpreter eine ähnliche Geschwindigkeit erreichten und die Nachfrage somit schnell nachließ.

Auf Grund dieser Erfolge übernahmen auch bald andere Sprachen das gleiche Konzept. Es folgte beispielsweise „Warren's Abstract Machine" für die Programmiersprache Prolog(1983) oder auch andere Interpreter für Sprachen wie Perl (1987), Python (1990) und Lua (1993). Auch Microsoft verwendete eine P-Code ähnliche Zwischensprache für die Programmiersprache Visual Basic. Dort stand jedoch nicht die Plattformunabhängigkeit im Vordergrund, sondern die Verringerung der Codegröße. Denn Zwischencode, auch Bytecode genannt, ist in jedem Fall wesentlich kleiner als Maschinencode. Microsoft versuchte den Code auch dadurch klein zu halten, indem oft verwendete Funktionen wie die Windows API Funktion SysAllocString[1] (zur Allokation neuer Zeichenketten) als Instruktion der Zwischensprache hinzugefügt wurden.

Anfang der 90'er Jahre entstand nun die Sprache Java von Sun Microsystems. Ursprünglich wurde der Java Interpreter (damals noch Oak Interpreter genannt) für eine Software innerhalb interaktiver

---

[1] http://msdn2.microsoft.com/en-us/library/ms221458.aspx

Fernsehgeräte oder auch für kleinere Haushaltsgeräte entwickelt. Als die Software im Jahre 1992 erstmal lauffähig war, galten die Geräte schon als veraltet, sodass es nie zur Produktion kam. Stattdessen wurde die erfolgsversprechende Software unter dem Namen Java weiterentwickelt und im Jahre 1995 veröffentlicht. Die „Java Virtual Machine“ wurde somit die erste virtuelle Maschine. Alle vorher entwickelten Interpreter können als Vorläufer der JVM gesehen werden, da sie noch keine geschlossene Ausführungsumgebung für die Programme zur Verfügung stellten. Es waren einfache Abstraktionsschichten zwischen Hardware und Software, jedoch fehlte z.B. eine Schnittstelle zum Betriebssystem. Desweiteren hatte der Java Bytecode einen großen Unterschied zu bisherigen Zwischensprachen, denn alle Verweise innerhalb des Codes wurden nun symbolisch aufgelöst. Das bedeutet, dass alle Methodenaufrufe und Offsets für Objektvariablen nicht fest im Code gespeichert sind, sondern erst zur Laufzeit berechnet werden. Dafür sind zusätzliche Typinformationen nötig, die zusammen mit dem Code gespeichert werden müssen.

Als letztes folgte Microsoft im Jahre 2000 mit der .NET Initiative. Bill Gates verfolgte damit das Ziel eine einheitliche Zwischensprache für viele Programmiersprachen zu kreieren. So entstand die Common Intermediate Language (CIL), die dem Java Bytecode sehr ähnlich ist, aber doch einige signifikante Unterschiede aufweist. Microsoft wollte möglichst viele C++ Programmierer von der neuen Plattform überzeugen, daher war es zwingend notwendig, dass die CIL auch C++ Code wieder verwenden kann. Folglich wurde die CIL so angepasst, dass Zeigerarithmetik als auch Funktionszeiger realisierbar sind. Danach wurde eine neue Sprache geschaffen, die Managed C++ Language. Diese Weiterentwicklung von C++ wird direkt nach CIL kompiliert und ist somit kompatibel zu allen anderen .NET Sprachen. Die Hauptsprache für die .NET Plattform ist jedoch C#, eine von Microsoft entwickelte Sprache, die Java sehr ähnlich ist. Im Laufe der Zeit sind jedoch immer mehr Sprachen hinzugekommen, die auf der .NET Plattform laufen. Beispiele dafür sind: Visual Basic.NET, J#, IronPython, JavaScript.NET, #SmallTalk und NetRuby, um nur einige zu nennen.

Erstaunlicherweise machte Microsoft die Spezifikationen der .NET Laufzeit und der C#-Sprache öffentlich, sodass mehre Open Source Projekte zur Portierung der sogenannten Common Language Runtime (CLR) entstanden. Ende des Jahres 2006 hat das größte Portierungsprojekt namens Mono die Version 1.2 fertiggestellt, womit nun .NET Programme nicht nur unter Windows, sondern auch unter Linux und Mac OS lauffähig sind.

Typen virtueller Maschinen

Wie bereits in der Begriffsklärung definiert, gibt es zwei verschiedene Arten von virtuellen Maschinen, die sich jedoch grundlegend voneinander unterscheiden. Virtuelle Server emulieren eine ganze Maschine wohingegen Laufzeitumgebungen für Programme lediglich einen abstrakten Prozessor zur Verfügung stellen. Dabei wird der PC, der als Plattform zu Emulation weiterer Maschinen dient, oft auch als Host-System bezeichnet. Dementsprechend wird der virtuelle Server als Gastsystem bezeichnet.

Beide Maschinentypen haben trotzdem ein gemeinsames Ziel. Ein Code in einer nicht näher definierten Sprache soll unabhängig von der zu verwendenden Hardware in einer weitgehend isolierten Umgebung ausgeführt werden.

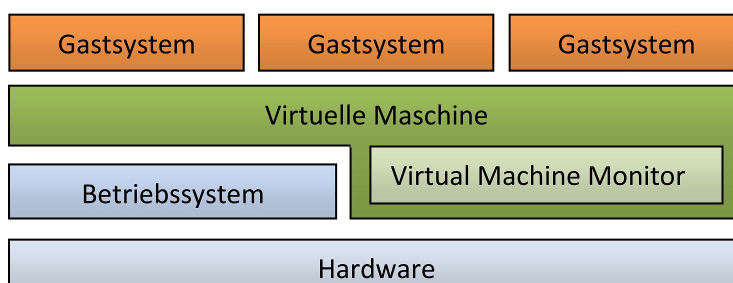
Virtuelle Hardware Maschinen (Virtuelle Server)

Virtuelle Hardware Maschinen emulieren in der Regel reale Prozessoren wie z.B. die PowerPC CPU mitsamt allen anderen Hardwarekomponenten (Laufwerke, Arbeitsspeicher, Sound – und Grafikkarten, etc.) der entsprechenden Maschine. Die Anwendung ist außerdem nicht auf PCs beschränkt. Es ist genauso möglich eine Spielekonsole oder ein Handy auf einem PC zu emulieren.

In der Praxis werden virtuelle Hardware Maschinen oft zum Testen von Software verwendet. Denn mit einer VM hat man ein kostengünstiges Mittel um Programme auf einer anderen Plattform laufen zu lassen. Außerdem kann man virtuelle Server nutzen, um einen echten Server in mehrere isolierte Bereiche aufzuteilen.

Die Realisierung einer solchen VM kann auf zwei unterschiedlichen Wegen geschehen. Entweder man entwickelt eine Software, welche die abstrakte Maschine zur Verfügung stellt oder aber man verwendet Hardwarekomponenten, die eine direkte Virtualisierungsunterstützung bieten.

Zur Realisierung solch einer VM gibt es vier unterschiedliche Methoden, die weiter unten erklärt werden. Die ersten drei Methoden basieren auf einer Software oder Software/Hardware Mix-Lösung. Nur die letzte Variante basiert vollständig auf einer Hardware-Lösung. Eine Gemeinsamkeit aller Softwarelösungen liegt darin, dass meist ein Virtual Machine Monitor verwendet wird. Dieser ist dafür verantwortlich alle Ressourcen gerecht zwischen allen Gastsystemen aufzuteilen.



Diese Abbildung soll noch einmal verdeutlichen wie die virtuelle Maschine im Gesamtkontext einzuordnen ist. Es ist ersichtlich, dass die Gastsysteme alle oberhalb

der VM ablaufen und dass die Virtuelle Maschine auf das Betriebssystem sowie auf die Hardware zugreift. Insbesondere der Virtual Machine Monitor, der ein Bestandteil der VM ist, agiert mit der Hardware. Bei einer Hardware basierten Lösung entfällt natürlich das Betriebssystem und die virtuelle Maschine wird zumindest teilweise durch die Hardware ersetzt.

Vollständige Emulation (Hardware Virtualization)

Bei der vollständigen Emulation wird der komplette Befehlssatz eines realen Prozessors simuliert. Dies funktioniert genau so wie bei den P-Code Interpretern mit dem Unterschied, dass kein Zwischencode, sondern z.B. x86 Maschinencode interpretiert wird. Eine weitere Möglichkeit zur vollständigen Emulation ist das Umwandeln in Maschinencode, der auf dem Host-System lauffähig ist. Da die Prozessorarchitekturen zum Teil sehr unterschiedlich ausfallen, ist diese Variante sehr aufwendig und weniger verbreitet. Ein Beispiel für solch eine virtuelle Maschine ist der Emulator Bochs[2], der in der Lage ist x86 Code auf vielen verschiedenen Systemen auszuführen.

Gemischte Emulation (Hybrid Virtualization)

Die gemischte Emulation stellt dem Gastsystem einen Teil der physischen Ressourcen direkt zur Verfügung, sodass nur noch wenige Komponenten emuliert werden müssen. Die Prozessorarchitekturen des Host- und des Gastsystems müssen außerdem identisch sein, da der Code direkt ausgeführt wird. Die bekannteste Implementierung solch einer Software ist VMWare[3]. Durch die eben genannte Beschränkung ist es jedoch nicht möglich mit solch einer Software ein PowerPC System auf einem Intel Prozessor laufen zu lassen. Trotzdem ist diese Art der Emulation sehr beliebt, da es nur zu geringen Geschwindigkeitseinbußen kommt.

Paravirtualisierung

Die Paravirtualisierung ist streng genommen keine echte Virtualisierung einer Maschine, da keine Hardware emuliert wird. Stattdessen wird nur ein Betriebssystem emuliert, was aber speziell an die virtuelle Maschine angepasst sein muss. Alle Hardware-Zugriffe müssen direkt über den Virtual Machine Monitor an das Host-System weitergeleitet werden. Dazu muss das Gastsystem sogenannte Hypercalls senden, diese sind die einzige Möglichkeit zur Kommunikation zwischen Host- und Gastsystem. Der Nachteil liegt hier natürlich darin, dass das Betriebssystem erst modifiziert werden muss, um zusammen mit der VM zu funktionieren. Dafür erhält man eine hohe Performance, die den Aufwand wieder rechtfertigen könnte.

Virtualisierung auf Hardwareebene

Der Vollständigkeit halber sei hier außerdem die Virtualisierung auf Hardwareebene erwähnt. Hierfür ist keine spezielle Software nötig, weil die Hardware bereits entsprechende Funktionen besitzt.

[2] http://bochs.sourceforge.net/
[3] http://www.vmware.com/

Daher unterscheidet sich diese Variante vollständig von den ersten drei Methoden, denn das Host-System muss nicht einmal ein Betriebssystem haben.

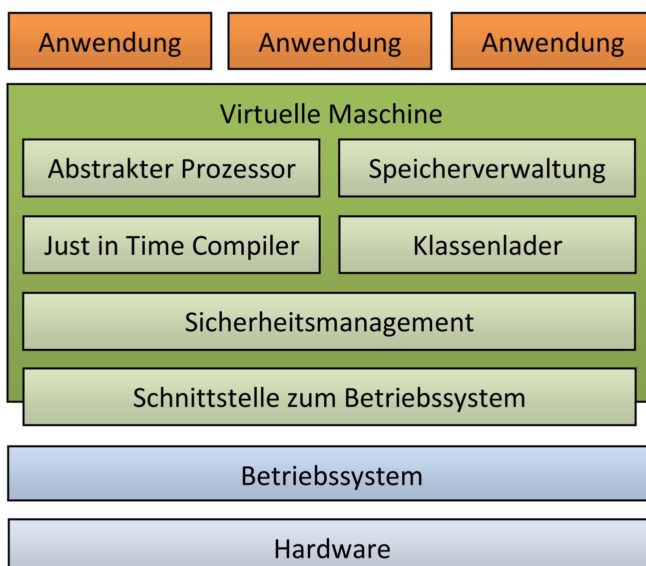
Ein Prozessor, der dies erlaubt, ist beispielsweise der Transmeta Prozessor. Im Unterschied zu den meisten anderen Prozessoren ist sein Befehlssatz nicht fest auf dem Chip verankert, sondern kann flexibel programmiert werden. Die Maschinensprache der Transmeta CPU kann somit nicht nur erweitert, sondern auch komplett ausgetauscht werden.

Weiterhin gibt es die Prozessorerweiterungen Pacifica von AMD und Vanderpool von Intel. Auch mit diesen ist die Virtualisierung auf Hardwareebene möglich, jedoch besitzen erst neuere Prozessoren im Jahre 2006 diese Fähigkeiten.

Virtuelle Maschinen zur Ausführung von Anwendungen

Virtuelle Maschinen zur Ausführung von Anwendungen emulieren einen abstrakten Prozessor und stellen gleichzeitig eine komplette Laufzeitumgebung für Programmezur Verfügung. Dazu gehört nicht nur das simple Ausführen des Programmcodes, sondern auch die Speicherverwaltung, eine Schnittstelle zum Betriebssystem, ein Klassenlader, Sicherheitsmanagement und eventuell noch ein Just in Time Compiler.

Eine sehr bekannte VM ist die DOS-Box von Windows, die seit Windows NT zur Emulation einer DOS-Umgebung genutzt werden kann. Da die Prozessorarchitektur seit DOS im Wesentlichen gleich geblieben ist, wird kein abstrakter Prozessor benötigt. Die CPU muss aber in den sogenannten Real Mode gesetzt werden, in dem man nur maximal 1 Megabyte adressieren kann. Außerdem werden andere Hardwareteile eines typischen IBM PCs, wie er zu DOS Zeiten anzutreffen war, emuliert.



Auf der linken Seite ist nun ein Schema einer virtuellen Maschine zu sehen. Ganz oben sind die Anwendungen, die basierend auf der VM ausgeführt werden und darunter sind alle Komponenten aufgeführt. Dabei ist ersichtlich, dass die virtuelle Maschine eine Schnittstelle zum Betriebssystem bereitstellt, welche meist durch eine Klassen- bzw. Methoden-bibliothek realisiert ist. Diese Kapselung ist erforderlich, da alle hardwarenahen

Funktionen nicht plattformunabhängig sind. Dazu zählen alle Dateioperationen, das Erzeugen von Threads, die Kommunikation mit anderen Prozessen oder auch andere Zugriffe auf vom Betriebssystem verwaltete Ressourcen.

Die Programme können entweder interpretiert oder direkt ausgeführt werden. Für die Interpretationen ist der abstrakte Prozessor zuständig. Charakteristisch für einen abstrakten Prozessor ist der Sachverhalt, dass er keinen realen Maschinencode, sondern immer nur einen Zwischencode ausführt. Eine genauere Erläuterung zum Zwischencode wird weiter unten nach dem Abschnitt zu register – und stapelbasierten Maschinen gegeben.

Um einen Geschwindigkeitszuwachs zu erhalten, verwendet man seit einigen Jahren zusätzlich einen sogenannten Just in Time (JIT) Compiler, der den Zwischencode in Maschinencode umwandelt, wenn er das erste Mal verwendet wird. Die Vor- und Nachteile eines solchen Compilers werden in dem zweiten Abschnitt der Arbeit ausführlicher erläutert.

Der Klassenlader darf natürlich auch nicht vergessen werden, er ist für das Laden des Codes und der zugehörigen Typinformationen zuständig. Bei nicht objektorientierten Sprachen können die Typinformationen zum Teil auch wegfallen. Weiterhin muss der Lader auch alle Verweise in der Datei auslesen und referenzierte Pakete ebenfalls einlesen. Anschließend werden die eingelesenen Codes miteinander verknüpft (Linking). Alternativ kann dieser Schritt auch im Just in Time Compiler ausgeführt werden. Als Beispiel sei hier einmal die String Klasse aus Java genannt, welche sich im `java.lang` Paket befindet. Verwendet man diese Klasse in einem Java Programm, so muss auch dieses Package vor der Ausführung geladen worden sein.

Während der Ausführung sorgt oftmals auch ein spezielles Sicherheitsmanagement dafür, dass die Anwendungen nur Dinge tun, für die sie auch berechtigt sind. In modernen Laufzeitumgebungen ist es beispielsweise möglich einem Programm den Zugriff auf das Dateisystem oder die Registrierung[4] zu verweigern. Außerdem weist die virtuelle Maschine den Anwendungen auch einen bestimmten Satz an Berechtigungen zu, abhängig davon aus welcher Quelle die Software stammt. Programme aus dem Internet sollten z.B. standardmäßig nicht als vertrauenswürdig eingestuft werden.

Die letzte wichtige Komponente innerhalb der Ausführungsschicht ist die Speicherverwaltung. Alle Speicherreservierungen und Freigaben laufen über diese Schnittstelle. Falls die auszuführende Sprache einen Garbage Collector (GC) benötigt, übernimmt dieser das Freigeben des Speichers. Dieser verwaltet dann automatisch alle Objekte, daher nennt man Sprachen mit einem GC auch verwaltete (managed) Sprachen.

---

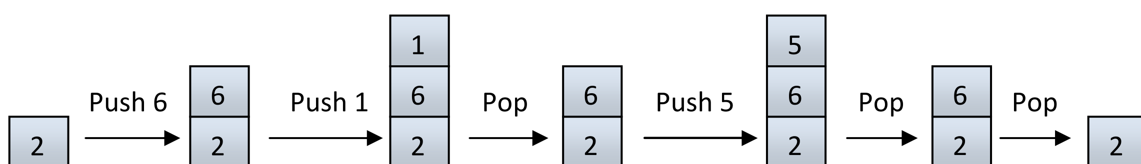
[4] Hierarchische Datenbank zur Konfiguration von Anwendungen und Komponenten innerhalb von Microsoft Windows

Sobald der Speicher knapp wird, springt der Garbage Collector an und sucht nach Objekten, die nicht mehr referenziert werden. Dafür wird die Ausführung unterbrochen. Alle nicht mehr benötigen Objekte werden aussortiert, sodass wieder neuer Platz für andere Daten entsteht. Sein Arbeitsbereich ist der sogenannte Freispeicher (auch Heap genannt), der für jede Anwendung einmal existiert. Dort werden alle Objekte abgelegt, die über eine längeren Zeitraum leben wollen. Im Falle von Java müssen sogar alle Objekte auf dem Heap landen, egal wie oft und wie lange man sie verwenden will. In anderen Sprachen wie Managed C++ und C# ist es jedoch auch möglich Objekte auf dem Stack zu erzeugen.

Stapelbasierte und registerbasierte Maschinen

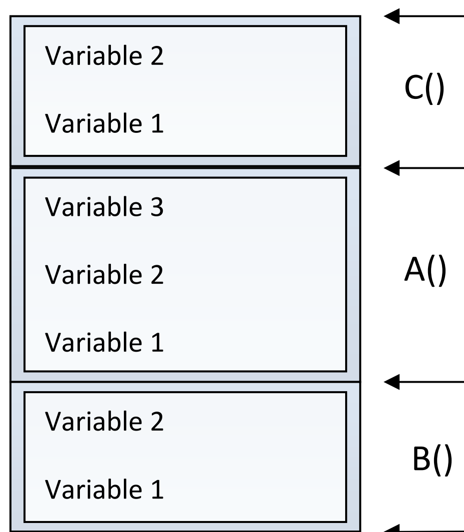
Der abstrakte Prozessor kann auf zwei verschiedenen Wegen realisiert werden. Entweder man verwendet eine stapelbasierte Maschine (Stackmaschine) oder eine Registermaschine. Um zu verstehen wo der Unterschied zwischen beiden liegt, muss als erstes erklärt werden was überhaupt ein Stapel ist.

Ein Stapel (auch Stack oder Keller genannt) ist eine Datenstruktur, die nach dem Last In First Out Prinzip arbeitet. Es gibt drei Operationen, die man auf einen Stack anwenden kann: Das Ablegen (push), das Runternehmen (pop) und das Auslesen von Werten. Man verwendet den Keller, um lokale Variablen, Parameter sowie andere zu einer Methode gehörige Daten abzuspeichern. Die Datenstruktur wird von vielen Prozessoren direkt unterstützt, indem Register zur Abgrenzung des Stacks vorhanden sind. Zusätzlich stehen bei diesen CPUs auch Maschineninstruktionen bereit, mit denen man Werte auf den Stack pushen und poppen kann. Die folgende Abbildung verdeutlicht die Funktionsweise der Struktur:



Stapelbasierte Maschinen zeichnen sich dadurch aus, dass sie zum Speichern und Berechnen von Werten nur Stacks verwenden und keinerlei andere Datenstrukturen. Normalerweise würde zur Ausführung eines Bytecodes ein einziger Stack genügen, wenn man davon ausgeht, dass alle Befehle synchron nacheinander ausgeführt werden sollen. Java als auch die .NET Zwischensprache verwenden jedoch einen zusätzlichen Stapel, welcher ausschließlich zur Speicherung von temporären Werten verwendet wird. Somit sind die lokalen Variablen strikt von den Werten aus den aktuellen Berechnungen getrennt.

Desweiteren unterteilt man den Stapel in sogenannte Stackframes, diese geben einen Abschnitt an, der zu einem Methodenaufruf gehört. Jedes Mal wenn man eine Methode aufruft wird solch ein Rahmen angelegt, beim Verlassen wird er wieder zerstört. Der Rahmen sorgt dafür, dass man die lokalen Variablen der aktuellen Methode mit einem festen Index adressieren kann. Daher benötigt man immer dann einen Stackframe, wenn man lokale Variablen mit einem Index im Code ansprechen will. In folgendem Bild ist zu sehen wie der Stapel nach mehren Methodenaufrufen aussieht.



In dem Fall startete die Ausführung in der Methode B(). Dort wurde die Methode A() aufgerufen, welche dann zu C() sprang.

Damit Prozessoren wissen an welche Stelle sie sich momentan im Code befinden, sichern sie die aktuelle Codeposition in einem speziellen Register. Bei einem Methodensprung muss diese Position gespeichert werden, sodass die Ausführung wieder dort weitergeführt werden kann, wo sie zuletzt stoppte.

Die meisten heutigen Computer gehen auf die Neumann Architektur zurück, welche auch besagt, dass die Rücksprungadresse auf dem Stackframe gespeichert wird. Dies ist eine sehr effiziente Methode, weil somit alle benötigten Daten für eine Methode im Speicher eng aneinander liegen. Auf der anderen Seite erscheint es aus heutiger Sicht nicht sehr praktikabel Daten und Rücksprungadresse an einem Ort aufzubewahren, da man somit Gefahr läuft die Adresse zu überschreiben.

Die zweite Möglichkeit zur Implementation eines abstrakten Prozessors ist die Verwendung einer registerbasierten Maschine. Dieses Modell ist identisch mit dem eines realen Prozessors, wie beispielsweise der IA-64 CPU[5]. Anstatt die Berechnungen mittels eines Stapels durchzuführen, werden alle Operationen in sogenannten Registern ausgeführt. Ein Register kann immer genau einen Wert speichern, daher kann man sich ein Register also wie eine lokale Variable vorstellen. Der aktuelle Zustand aller Register wird als Konfiguration der Registermaschine bezeichnet, jede Operation verändert diese Konfiguration. Der Vorteil dieses Modells liegt darin, dass die Zwischensprache schon relativ nahe an der echten Maschinensprache ist. Dadurch vereinfacht sich der Kompiliervorgang im Back End, bei dem der Zwischencode in den Maschinencode kompiliert wird.

[5] Ein nicht abwärts kompatibler 64 Bit Prozessor von Intel mit insgesamt 256 Registern

Registermaschinen haben entweder eine unendliche oder endliche Anzahl an Registern. Im zweiten Fall muss man zwangsläufig auf eine weitere Datenstruktur zurückgreifen, um Daten zu speichern, die nicht in die Register passen.

Ein Beispiel für eine Registermaschine ist die momentan in der Entwicklung befindliche VM Parrot[6].

Zwischencode

Der Zwischencode einer virtuellen Maschine stellt eine Abstraktionsschicht zwischen der Software und der Hardware dar. Diese zusätzliche Schicht ist unter anderem dafür verantwortlich, dass die Plattformunabhängigkeit möglich ist. Würde man die virtuelle Maschine als einen realen Prozessor betrachten, so wäre der Zwischencode der native Maschinencode dieser Maschine.

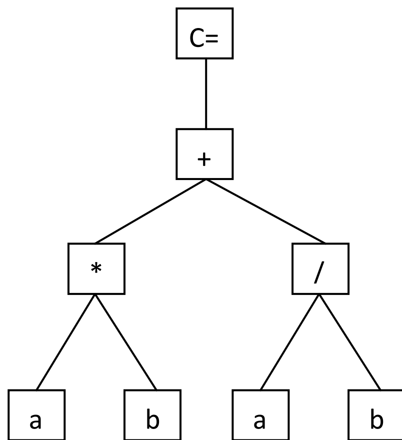
Genau wie beim Maschinencode, so setzt sich der Zwischencode aus einer Folge von unterschiedlichen Instruktionen zusammen. Eine Instruktion wird immer durch einen Opcode (Operation Code) eingeleitet und durch null oder mehr Operanden bzw. Parameter abgeschlossen. Wie der Name bereits vermuten lässt, bestimmt der Operation Code welche Aktion auszuführen ist. Oftmals wird der Zwischencode auch als Bytecode bezeichnet, da die meisten Opcodes genau ein Byte groß sind. Alle Instruktionsarten zusammen bilden den Befehlssatz des Prozessors. Zur Identifikation der einzelnen Instruktionen in der Zwischensprache werden alphanumerische Abkürzungen verwendet, die sogenannten Mnemoniks.

Die Größe des Befehlssatzes hängt stark von dem Typ der VM ab. Stackmaschinen brauchen in der Regel wesentlich weniger Befehle als Registermaschinen, weil nicht angegeben werden muss auf welches Register sich die Aktion bezieht. Daher kommen z.B. Java Bytecode und die CIL mit weniger als 256 Befehlen aus, sodass ein Byte in jedem Fall zur Kodierung der Operation reicht. Reale Prozessoren und auch virtuelle Registermaschinen brauchen jedoch oftmals mehrere tausend Instruktionen. Besonders Prozessoren wie der x64 Prozessor haben einen sehr großen Befehlssatz, da stets Abwärtskompatibilität zu x86 Systemen gewährleistet sein muss. Da sind Stackmaschinen im Vorteil, da die Hardware dort stets abstrakt betrachtet werden kann, wodurch die Sprache auf einer höheren Ebene arbeitet und so weniger Instruktionen benötigt.

Eine Instruktion `mul` könnte z.B. dafür verantwortlich sein zwei Werte zu multiplizieren. Auf einer Registermaschine oder einem realen Prozessor mit n Registern, müssten nun mindestens zwei Operanden folgen, die definieren welche Register zu multiplizieren sind, Beispiel: `mul` r_{n-1}, r_{n-2}. Betrachtet man nun eine stapelbasierte Maschine, so fällt auf, dass keine Operanden benötigt werden. Dies folgt aus der einfachen Annahme, dass alle Operationen auf die oberen Werte des

[6] http://www.parrotcode.org/

Stacks angewendet werden sollen. Abhängig von der verwendeten Sprache werden die Operanden eventuell auch direkt in den Opcode integriert. In dem Fall werden `n * n` Opcodes statt einem einzigen benötigt. Dies führt dann dazu, dass die Anzahl der Opcodes stark ansteigt, wie schon weiter oben beschrieben.



Der Zwischencode entsteht normalerweise im Front End des Compilers, nachdem der abstrakte Code Baum erzeugt wurde. Zur Demonstration soll hier der folgende Ausdruck in einem Baum dargestellt und anschließend in Zwischencode umgewandelt werden: `c = (a * b) + (a / b).` Dieser kleine Codeschnipsel bildet die Summe aus dem Produkt der Variablen a und b und dem Quotient der Variablen a und b. Das Ergebnis soll am Ende in die Variable c geschrieben werden. Auf der linken Seite ist nun die Repräsentation des Ausdrucks in Baumform zu sehen.

Um den Baum nun in Zwischencode zu kompilieren, wird der Graph post-order traversiert und jeder Knoten in eine Instruktion umgewandelt. Als Liste dargestellt, ergibt sich nun folgendes Ergebnis: `{a, b, *, a, b, /, +, c=}`. Beispielhaft soll hier nun eine Implementation des Codes in der Common Intermediate Language von Microsoft gezeigt werden. Der folgende Abschnitt wurde direkt von einem C# Compiler erzeugt:

```
ldloc.0        // ‘a‘ auf den Stack schieben
ldloc.1        // ‘b‘ auf den Stack schieben
mul            // die beiden oberen Werte auf dem Stack multiplizieren
ldloc.0        // ‘a‘ auf den Stack schieben
ldloc.1        // ‘b‘ auf den Stack schieben
div            // die beiden oberen Werte auf dem Stack dividieren
add            // die beiden oberen Werte auf dem Stack addieren
stloc.2        // Resultat in Variable ‘c‘ speichern
```

Der entsprechende Java Bytecode würde genauso aussehen mit der Ausnahme, dass der Java Code andere Mnemoniks verwendet.

Eigenschaften virtueller Maschinen

Dieser Abschnitt soll die Eigenschaften virtueller Maschinen für Anwendungen genauer beleuchten. Dabei wird auch insbesondere auf bereits existierende virtuelle Maschinen von Microsoft und Sun eingegangen.

Plattformunabhängigkeit

Der Wunsch nach Plattformunabhängigkeit war der ursprüngliche Grund zur Entwicklung einer virtuellen Maschine. Und auch heutzutage ist diese Eigenschaft sicher die wichtigste von allen. Daher soll zuerst geklärt werden, wodurch die Plattformunabhängigkeit überhaupt möglich ist.

Nicht alle Bestandteile einer modernen virtuellen Maschine werden auch benötigt, um die Unabhängigkeit von einer Plattform zu gewährleisten. Man benötigt im Prinzip nur zwei Komponenten: Den abstrakten Prozessor mitsamt seiner Zwischensprache plus der gekapselten Schnittstelle zum Betriebssystem.

Der abstrakte Prozessor sorgt dafür, dass man alle Plattformen wie eine einzige betrachten kann. Der Zwischencode ist dabei der Maschinencode des Prozessors und somit die Zielsprache aller Anwendungen. Doch allein der Prozessor genügt noch nicht, um plattformunabhängige Programme zu entwickeln. Jedes einigermaßen komplexe Programm muss auf Ressourcen des Betriebssystems zugreifen. All diese Funktionalitäten sind in den Betriebssystemen verankert, aber jeweils anders implementiert. Daher benötigt man quasi ein virtuelles Betriebssystem, was durch die Schnittstelle zum Hostsystem realisiert wird (siehe Abbildung zu virtuellen Maschinen für Anwendungen).

Durch die Plattformunabhängigkeit wird die Entwicklung von Anwendungen für mehrere Systeme nun wesentlich kostengünstiger, da man nur noch eine Softwareversion pflegen muss. Man kann also mit wenig Aufwand sehr schnell Programme für viele Plattformen schreiben und somit seinen Nutzerkreis drastisch vergrößern.

Die bekannteste plattformunabhängige Programmiersprache ist sicherlich Java. Ihr Erfolg ist eng verknüpft mit der Tatsache, dass Java von Anfang an für eine ganze Reihe von Plattformen verfügbar war. Im Bereich der Mobiltelefone ist Java sogar zum quasi Standard für portable Anwendungen geworden. Fast jedes Handy hat heutzutage eine JVM mit an Bord. Doch wieso ist die JVM bei mobilen Geräten so beliebt? Diese Frage ist ganz einfach zu beantworten. Mobile Geräte haben aus Kostengründen schon immer nicht besonders viel Speicher gehabt. Doch genau den braucht man reichlich, um Maschinencode zu speichern.

Auch die Common Intermediate Language von Microsoft war von Anfang an so ausgelegt, dass es im Prinzip mit Leichtigkeit möglich ist die virtuelle Maschine auf eine andere Plattform zu portieren. Daher hat Microsoft auch eine Open Source Implementation der Common Language Runtime veröffentlicht, die das Ausführen von .NET Programmen auf Linux und Mac möglich macht. Obwohl dies bis dato die größte Code-Veröffentlichung von Microsoft war, kann die Runtime nicht wirklich genutzt werden. Denn auf Grund der genutzten Lizenz ist die Verwendung für kommerzielle Zwecke untersagt.

Aus diesem Grund wurde eine Reihe von Projekten zur Portierung der virtuellen Maschine mitsamt der Klassenbibliothek ins Leben gerufen. Das bekannteste Projekt ist Mono, mit dem es mittlerweile auch möglich ist .NET Programme auf Linux, Mac und natürlich Windows auszuführen. Zusätzlich gibt es ähnlich wie bei Java eine abgespeckte Version der Laufzeitumgebung für mobile Geräte. Jedoch ist die Umgebung weniger für Handys geeignet, sondern eher für Smartphones und PDAs.

Desweiteren ist es seit Ende 2006 sogar erstmals möglich Managed Code auf einer Spielekonsole auszuführen. Denn im Dezember wurde das XNA Game Studio vorgestellt, eine Entwicklungsumgebung auf Basis der .NET Plattform. Das Besondere dabei ist, dass der Code ohne Änderungen auf Windows und der Xbox 360 läuft. Diese Entwicklung zeigt auch, dass die Geschwindigkeit von Managed Code mittlerweile mit nativem Code vergleichbar ist.

Andersrum ist es mit Hilfe von virtuellen Hardware Maschinen aber auch möglich Konsolenspiele auf dem PC zu starten. Zum Beispiel gibt es für die Playstation One etliche Emulatoren im Internet.

Geschwindigkeit

Die Geschwindigkeit ist auch eine wichtige Eigenschaft, die für den Erfolg oder das Versagen einer virtuellen Maschine entscheidend ist.

Die ersten Laufzeitumgebungen, Java eingeschlossen, waren nicht gerade bekannt für ihre Schnelligkeit. Auch heute noch haben verwaltete Sprachen wie Java und C# den Ruf langsam zu sein. Tatsächlich waren frühe VMs nicht besonders schnell, einfach weil sie noch keinen Just in Time Compiler oder keinen effizienten Garbage Collector besaßen. Daher musste der komplette Code interpretiert werden, was zu einer wesentlich schlechteren Leistung führte.

Doch dauerte es nicht lange und man kam auf die Idee den Zwischencode im Hintergrund schnell in Maschinencode zu verwandeln, wenn er das erste Mal gebraucht wurde. Der Just in Time Compiler war geboren. Prinzipiell handelt es sich dabei nur um eine Weiterentwicklung eines Back End Compilers, wie er schon zu P-Code Zeiten verwendet wurde. Da aber immer nur der benötigte Code

kompiliert wird, fällt die Kompilierzeit relativ kurz aus. Trotzdem war bei den ersten JIT Compilern oftmals eine spürbare Pause zu vernehmen, wenn man beispielsweise das erste Mal auf einen Button klickte. Die Geschwindigkeit des Compilers ist also sehr kritisch. Mittlerweile sind sie jedoch so schnell, dass Verzögerungen in den seltensten Fällen auftreten. Dies geht jedoch auf Kosten des erzeugten Codes, denn anders als beim Kompilieren im Front End bleibt keine Zeit mehr zum Optimieren.

Alternativ kann man deswegen einen Ahead of Time Compiler verwenden, der den Zwischencode vor der ersten Ausführung oder schon bei der Installation komplett übersetzt. In dem Fall bleibt auch mehr Zeit zur Optimierung. Der Vorteil dieser Variante besteht darin, dass die Eigenschaften der Zielplattform genau bekannt sind und somit auch hardwarenahe Optimierungen genutzt werden können.

Kompiliert man beispielsweise C++ muss man stets von der Plattform mit den wenigsten Funktionen ausgehen. Bei einem Ahead of Time Compiler können nun alle verfügbaren Prozessorerweiterungen genutzt werden. Für schnelle Fließkommaberechnungen kann die SSE[7] Erweiterung genutzt werden und für effektives Multithreading könnten Maschinenbefehle verwendet werden, die erst seit einiger Zeit in den Prozessoren vorhanden sind. Werden all diese Optimierungen auch wirklich genutzt, kann man sogar eine höhere Performance als bei C/C++ Programmen erreichen.

Ein anderer Ansatz ist die Hot Spot Technologie(Meloan, 1999) von Java, eine Weiterentwicklung des Just in Time Compilers. Dabei geht man von der Annahme aus, dass ein Programm die meiste Zeit in einem sehr kleinen Codebereich verbringt. Anders ausgedrückt sind also nur sehr wenig Codestellen optimierungswürdig. Alle Methoden, die nur ein paar Mal ausgeführt werden, können weiterhin ganz normal interpretiert werden. Für alle hoch frequentierten Methoden nimmt man sich allerdings besonders viel Zeit und kompiliert sowie optimiert diese Stellen. Im günstigsten Fall kann dabei sogar eine ähnliche Geschwindigkeit wie beim Ahead of Time Compiler erreicht werden. In jedem Fall aber verbraucht man wesentlich weniger Speicher, denn Maschinencode ist ein registerbasierter Code, der wesentlich länger ist. Um zu erkennen welche Methoden oft aufgerufen werden, muss eine Anwendung jedoch erst eine Weile laufen, daher eignet sich diese Technologie auch besonders für Server-Anwendungen.

Ein weiterer wichtiger Faktor für die Geschwindigkeit einer VM ist die Effizienz des Garbage Collectors. Denn der schnellste Compiler nützt einem nichts, wenn der Garbage Collector ständig den Speicher säubern muss und damit die Ausführung unterbricht. Auch die Leistungsfähigkeit der GCs

---

[7] Streaming SIMD (Single Instruction Multiple Data) Extensions

wurde immer weiter verbessert, sodass die heutigen Ausführungen so ausgereift sind, dass man sie fast schon für Echtzeit-Anwendungen nutzen kann.

Letztendlich ist natürlich auch zusagen, dass der Code selber auch ein wichtiger Faktor ist. Es ist immer möglich durch ungeschicktes Programmieren die Geschwindigkeit zu verlangsamen. Wenn man beispielsweise sehr oft Objekte erzeugen muss, bietet es sich an einen Cache auf Softwareebene zu realisieren, der nicht mehr benötigte Objekte später wiederverwendet.

Der Beweis für den Erfolg moderner virtueller Maschinen ist zweifelsfrei der bereits erwähnte Einsatz der .NET Plattform auf der Spielekonsole Xbox 360. Denn Spiele gehören zu der Art von Anwendungen für die jegliche Art von Verzögerungen ein großes Problem darstellt. Deswegen wurden Spiele bislang immer in C oder C++ programmiert, zumindest die zeitkritischen Bereiche.

Speicherverbrauch

Der benötigte Speicher einer Anwendung ist ebenso wie die Geschwindigkeit ein Faktor, der sich auf die Gesamtperformance auswirkt. Als erstes soll erklärt werden, weshalb man überhaupt den Speicherverbrauch minimieren will.

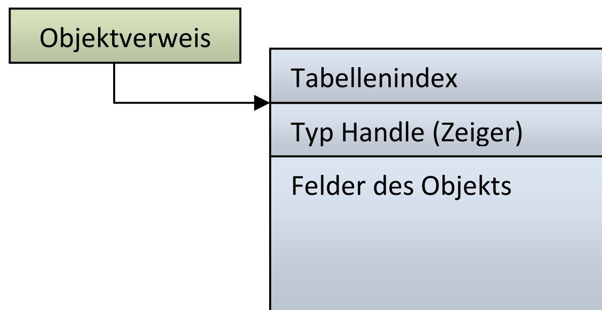
Gerade kleinere mobile Geräte oder auch andere Hardwaresysteme im Embedded Bereich (Autos, Kühlschränke, Fernseher, etc.) besitzen nicht sehr viel Arbeitsspeicher, daher sollte Software für diese Geräte möglichst speicherschonend sein. Außerdem trägt ein geringer Speicherverbrauch wesentlich zu einer höheren Geschwindigkeit bei. Denn umso kleiner der Code ist, umso seltener muss der Cache im Prozessor mit den neuen Daten aufgefüllt werden, d.h. es entstehen weniger Cache Misses.

Wie viel Speicher wirklich verwendet wird, hängt von sehr vielen Komponenten ab, wie der Codegröße, dem Objektlayout, der Effizienz des Garbage Collectors und natürlich auch von den Daten im Programm.

Die geringe Codegröße von Zwischencodes ist wie bereits im ersten Abschnitt der Arbeit erwähnt, ein klarer Vorteil gegenüber Maschinencode. Da viele Zwischencodes für verwaltete Sprachen ausgerichtet sind, kompensiert sich das Verhältnis zwischen Maschinen – und Zwischencode jedoch wieder etwas. Denn verwaltete Sprachen benötigen zusätzlich zum Code noch weitere Typinformationen, die den Aufbau aller Methoden, Klassen und Strukturen definieren. Trotz alledem ist verwalteter Code immer noch etwas kleiner als nativer Code.

Eine weitere Eigenschaft verwalteter Sprachen liegt darin, dass Objekte im Freispeicher ein fest definiertes Objektlayout haben. In der imperativen Programmiersprache C werden Objekte im

Speicher exakt so abgelegt, wie man sie im Quelltext definiert hat. Dies ist bei verwalteten Sprachen nicht mehr der Fall. Außerdem haben Objekte in C# und Java immer eine gewisse Mindestgröße. Dies kommt daher, weil man in den verwalteten Sprachen zu jeder Zeit den Typ eines Objekts erfahren kann. Zur Realisierung diese Funktion gibt es nur eine Möglichkeit: Jedes Objekt muss eine Information beinhalten, welche den Objekttyp definiert.



In dieser Abbildung ist beispielhaft das Objektlayout der .NET Laufzeit(Kommalapati & Christian, 2005) zu sehen. Dargestellt ist ein Verweis zu einem Objekt auf dem Heap.

Dieser Objektverweis zeigt auf das Typ Handle, welches ein Zeiger auf Typinformationen und eine Methodentabelle ist. Noch vor dem Typ Handle ist ein vier Byte großer Index, der in eine Tabelle mit näheren Informationen zu dem Objekt verweisen kann. Im Normalfall ist dieser Index null, sodass kein weiterer Speicher reserviert wird.

Wenn man nun aber den Hashcode des Objekts ausrechnet oder das Objekt sperrt (für den Zugriff durch andere Threads), so werden diese Informationen in der zusätzlichen Tabelle abgelegt. Im Anschluss an den Zeiger befinden sich die Instanzfelder aus der dazugehörigen Klasse. In der Summe ergibt sich, dass Objekte in .NET auf einem 32 Bit System mindestens eine Größe von acht Byte haben.

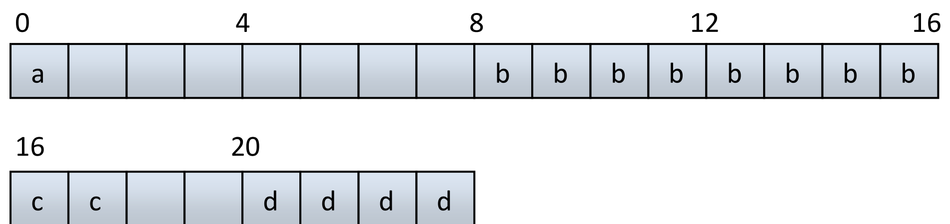
Weiterhin werden die Felder in verwalteten Sprachen, sortiert nach der Größe, im Arbeitsspeicher abgelegt. Dies hat den Vorteil, dass weniger Füllbytes eingefügt werden müssen. Füllbytes werden im Normalfall vom Compiler (z.B. in C++) automatisch eingefügt, damit Variablen auf einer Speicheradresse sitzen, die durch ihre Größe teilbar ist. Ganze Zahlen mit einer Größe von vier Bytes müssen daher auf einer durch vier teilbaren Adresse sitzen. Dieses Vorgehen verringert die Zeit, die ein Prozessor benötigt, um die Variablen in ein Register zu laden. Man spricht auch davon, dass die Variablen auf einer natürlichen Speicheradresse ausgerichtet sind.

Nachfolgend soll ein Beispiel für ein schlechtes Objektlayout gegeben werden, was aber von der .NET sowie von der Java Laufzeitumgebung verhindert wird.

Code:

```
class MyClass
{
    byte a;
    long b;
    short c;
    int d;
};
```

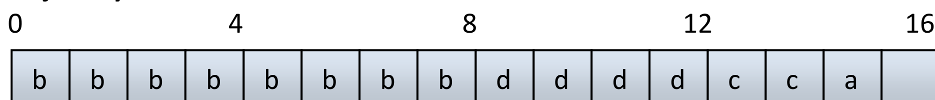
Objektlayout (ein Kästchen entspricht einem Byte):



In diesem Fall werden insgesamt neun Füllbytes eingefügt. Die ersten sieben Bytes an Stelle eins bis sieben entstehen, weil die `long` Variable auf einer Adresse liegen muss, die ein Vielfaches von acht ist. Die restlichen zwei Bytes werden hinzugefügt, da die `int` Variable auf einer durch vier teilbaren Adresse liegen muss.

In der Laufzeitumgebung werden alle Instanzfelder nun einfach von groß nach klein sortiert. Damit muss nur noch ein einziges Füllbyte eingefügt werden und die Instanzgröße reduziert sich um acht Byte auf nur noch sechszehn. Das nächste Bild zeigt die optimierte Anordnung der Felder:

Objektlayout:



Schlussfolgernd lässt sich sagen, dass Objekte im Allgemeinen in den verwalteten Sprachen schon von Anfang an relativ groß sind. Dies wird jedoch wieder etwas gemildert, indem das Layout im Speicher auf der anderen Seite wieder optimiert wird.

Desweiteren ist der Garbage Collector auch ausschlaggebend für den benötigten Platz einer Anwendung. Es gibt eine Menge verschiedener Algorithmen zur Implementation eines GCs. Doch an dieser Stelle sei nur gesagt, dass bei einer Säuberung alle lebenden Objekte zusammen geschoben werden. Damit wird der Fragmentierung des Arbeitsspeichers entgegen gewirkt, welche sich ansonsten nach einiger Zeit bemerkbar machen würde. Die Fragmentierung kommt dadurch zustande, dass Objekte unterschiedlicher Größen nicht in der Reihenfolge zerstört werden wie sie angefordert wurden. Daraus resultieren freie Bereiche im Speicher, die oftmals für neue Objekte zu klein sind. Im Endeffekt wird immer mehr Speicher benötigt, obwohl theoretisch noch genug verfügbar ist. Das Problem ist nur, dass er in zu kleine Portionen aufgeteilt ist.

Diese automatisch durchgeführte Defragmentierung bewirkt einen Vorteil gegenüber Sprachen ohne einen Garbage Collector mit dieser Eigenschaft. Auch nach langer Laufzeit eines Programms bleibt so der Speicherverbrauch annähernd konstant, vorausgesetzt das Verhalten der Anwendung in Bezug auf die Allokationen ändert sich nicht drastisch.

Abschließend kann man sagen, dass das Speicherverhalten von verwalteten Sprachen einige Vorteile bietet. Doch trotzdem kann auch die virtuelle Maschine dazu beitragen den Speicherbedarf hochzutreiben. Denn wenn ein großer Teil der Anwendung vom Just in Time Compiler oder sogar vom Ahead of Time Compiler kompiliert wurde, verliert man den Vorteil den der Zwischencode bietet.

Sicherheit

Die nächste Eigenschaft ist die angebotene Sicherheit einer virtuellen Maschine. Das Wort Sicherheit steht hier sowohl für den Schutz des PCs vor bösen Zugriffen als auch die sinkende Gefahr von Programmfehlern. Zuerst soll auf den zweiten Aspekt, die sinkende Fehlergefahr eingegangen werden. Dafür muss zunächst geklärt werden, was unabhängig von Fehlern in der Programmlogik die am häufigsten auftretenden Bugs sind.

Die wohl bekanntesten Fehler sind die gefürchteten Buffer Overflows, von denen man ständig lesen muss. Im Prinzip ist dies ein ganz simpler Fehler: Man schreibt in einen Speicherbereich, den der aktuelle Prozess überhaupt nicht angefordert hat. Die größte Gefahr droht dann, wenn ein Buffer Overflow auf dem Stack geschieht, da dann die Rücksprungadresse eventuell mutwillig überschrieben werden kann. Doch es gibt auch Methoden, um Buffer Overflows auf dem Heap auszunutzen.

```
int _tmain(int argc, _TCHAR* argv[])
{
    int* myArray = new int[10];

    for (size_t i = 0; i <= 10; i++)
        myArray[i] = i * i;

    delete[] myArray;

    return 0;
}
```

Diese kleine unmanged C++ Anwendung würde beispielsweise einen Buffer Overflow auf Grund des falsch gesetzten <= Operators verursachen. Das Programm würde versuchen in das nicht existente zehnte Arrayelement zu schreiben. Hat man Glück, so kommt es zu einem Absturz.

Für dieses Problem gibt es eine sehr einfache Lösung: Standardmäßig werden Zeiger in verwalteten Sprachen einfach verboten. Somit muss man bereits existierende Klassen zur Speicherung von Listen und Arrays verwenden, die jedoch bei jedem Zugriff dafür sorgen, dass es zu keinem ungültigen Speicherzugriff kommt. Tatsächlich ist es aber in C# und Managed C++ trotzdem weiterhin möglich mit Zeigern zu arbeiten. Doch dies muss in einem speziellen unsafe (unsicheren) Block geschehen, der auch erst über einen Compilerschalter aktiviert werden muss.

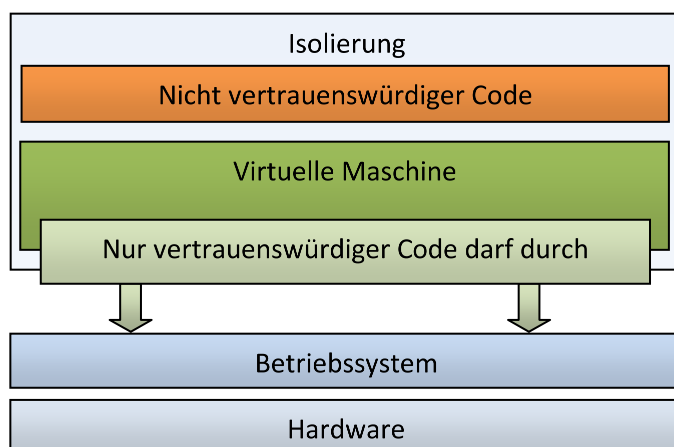
Zu Speicherlöchern und Dangling (baumelnden) Pointers kann es im Regelfall außerdem auch nicht kommen. Da der Garbage Collector das Löschen von Instanzen übernimmt, kann der Programmierer

keine Fehler mehr in diesem Bereich verursachen. Damit sind schon zwei mögliche Fehlerquellen eliminiert.

In C# gibt es außerdem noch eine weitere Möglichkeit, um einer Fehlerquelle aus dem Weg zu gehen. Sogenannte checked Blöcke verhindern das Auftreten von Integer Overflows. Tritt ein solcher Fehler innerhalb eines Blockes auf, wird eine Ausnahme geworfen und man kann entsprechend reagieren. In den meisten anderen Sprachen wird der Fehler – falls es sich denn um einen Fehler handelt – stillschweigend hingenommen.

Zweitens schützt die virtuelle Maschine vor unberechtigten Zugriffen durch fremden Code, indem Anwendungen in einer isolierten Umgebung ausgeführt werden. Sun hat dieses Konzept als eine Sandbox bezeichnet. Die Sandbox soll z.B. Anwendungen aus dem Internet den Zugriff auf die lokalen Ressourcen wie das Dateisystem verweigern. Im Falle von Java ist diese Isolierung besonders wichtig, da man Java Programme in Form von Applets in Webseiten integrieren kann, sodass der Nutzer nicht einmal merken muss, dass ein Java Code ausgeführt wird.

Dieses Konzept ist möglich, weil alle Zugriffe auf die Hardware und das Betriebssystem über die vordefinierte Schnittstelle laufen müssen. Baut man in diese Schnittstelle eine Kontrolle ein, kann man unbefugte Aufrufe zurückweisen. Auf der anderen Seite kann man aber auch gewisse Programme explizit durchlassen. Signierte Anwendungen, d.h. geprüfte Anwendungen kann der Zugriff auf lokale Ressourcen z.B. erlaubt werden.



Diese Abbildung soll das Schema der Isolierung bildlich darstellen. Gut zu sehen ist, dass es nur eine einzige Stelle gibt, an der Code durchdringen kann.

Auf der .NET Plattform gibt es außerdem das Code Access Security (CAS) Modell. Damit kann der Zugriff auf lokale Ressourcen noch detaillierter gesteuert werden. Folgende Bereiche können mit Hilfe von CAS für Anwendungen aktiviert oder deaktiviert werden: Informationen über den angemeldeten Nutzer, Zugriff auf das Dateisystem, Zugriff auf die Registrierung, Erlaubnis zur Anzeige einer Oberfläche, Ansprechen eines Druckers, Schreiben in das Windows Logbuch und das Verbinden mit einer Datenbank.

Die Berechtigungen, die eine Anwendung beim Start erhält, sind abhängig von der Quelle aus der sie stammt. Internetanwendungen haben selbstverständlich am wenigsten Rechte, danach folgen

Programme aus dem lokalen Intranet. Die meisten Erlaubnisse haben lokale Anwendungen, sie dürfen praktisch auf alle Ressourcen zugreifen. Es sei denn der Administrator verbietet dies explizit.

Desweiteren gibt es noch einen Sicherheitsmechanismus, der ein Bestandteil des Klassenladers ist: Die Verifizierung des Zwischencodes. Damit wird sichergestellt, dass der auszuführende Code gültig und typsicher ist. Diese Funktion findet man in Java wie in .NET.

Virtuelle Maschinen bieten also einen erheblichen Sicherheitsvorteil, falls sie fehlerfrei implementiert sind. Einen einzigen negativen Punkt gibt es jedoch. So schön ein Zwischencode auch ist, er hat trotzdem einen Nachteil. Man kann aus ihm sehr leicht den ursprünglichen Code wieder herstellen (Reverse Engineering). Mittlerweile gibt es für Java als auch für C# Tools mit denen man auf Knopfdruck einen kompletten Quelltext generieren kann. Das einzige mögliche Gegenmittel ist die Verschleierung des Codes durch einen Obfuskator.

Flexibilität

Ein weiterer wichtiger Aspekt bei der modernen Programmierung ist die Flexibilität eines Programmiersystems. Heutzutage ist es oftmals sehr nützlich Informationen über den eigenen Code (auch Metadaten genannt) rauszufinden. Dies wird als Reflektion bezeichnet. Dabei kann man alle Beschreibungen zum Aufbau und zur Hierarchie von Typen, Methoden und Feldern abrufen.

Eine erweiterte Form der Reflektion ist das dynamische Hinzufügen von eigenen Klassen und eigenem Code. Damit erreicht man eine neue Dimension der Flexibilität.

Das beste Anwendungsbeispiel für die Reflektion ist die Serialisierung von Objekten. Denn mit Hilfe der Reflektion ist die Serialisierung wesentlich leichter umzusetzen. Bisher musste man zur Serialisierung in Sprachen ohne Reflektion für jede Klasse mindestens zwei Methoden schreiben, die das Speichern und Laden steuern. Diese Methoden sahen meistens sehr ähnlich aus: Man schreibt alle wichtigen Instanzfelder in den Datenstream, damit das Objekt in einer Datei gespeichert oder über eine Verbindung versendet werden kann.

Dank der Reflektion kann dies im günstigsten Fall komplett entfallen. Möchte man ein Objekt serialisieren, liest man als erstes den Objekttyp aus. Auf Basis dieser Information kann man herausfinden wie das Objekt aufgebaut ist. Schließlich kann man alle Felder einzeln auslesen und abspeichern. All dies geschieht nur auf Grund der Typinformationen, die zur Laufzeit bereitgestellt werden. Die virtuelle Maschine gibt einem also den Zugriff auf alle Daten, die der Klassenlader am Anfang in den Speicher aufgenommen hat.

Weit aus mächtiger ist aber die Möglichkeit zur Erzeugung von eigenem Code zur Laufzeit. Eine simple Anwendung dafür wäre zum Beispiel ein Taschenrechner, der den eingegebenen Ausdruck kompiliert und dann ausführt.

Eine weitere Neuerung sind Attribute bzw. Annotations (bei Java). Wie der Name bereits andeutet, sind dies einfache Informationen, die man an Typen, Feldern und Methoden notieren kann. Was erstmal einfach klingt, kann für viele Bereiche sehr nützlich sein. Tools können spezielle Informationen im Code mitspeichern, das Verhalten des Codes kann gesteuern werden und Rechte können verteilt werden. In Java wurde dieses Konzept auch vor kurzer Zeit eingeführt, da es dort aber noch nicht sehr lange existiert, gibt es bisher kaum Anwendungen.

Als Beispiel sei hier wieder die Serialisierung genannt. Um lediglich eine Untermenge aller Instanzfelder in einer Klasse zu serialisieren, genügt es in .NET ein Attribut namens `NonSerialized` an die entsprechenden Variablen zu pinnen:

```
[Serializable]  // legt fest, dass die Klasse serialisierbar ist
class MyClass
{
    string name;
    int age;
    [NonSerialized] // die folgende Variable bei der Serialisierung ignorieren
    float income;
}
```

Durch die Flexibilität lässt es sich nun wesentlich produktiver arbeiten, da man viel Arbeit sparen kann.

Interoperabilität

Die Akzeptanz einer neuen Programmiersprache hängt stark von seiner Interoperabilität zu bereits bestehenden Sprachen und Technologien ab. Selbst die beste Sprache wird kaum Anwendung finden, wenn bereits bestehende Codebibliotheken dafür komplett neu geschrieben werden müssen.

Daher ist es sowohl in der Java VM als auch in C# möglich nativen Code aufzurufen. Für Java gibt es das JMI (Java Native Interface), mit dem man in der Lage ist Java Klassen und Methoden direkt in C/C++ zu implementieren. Dafür schreibt man zuerst die Methodendefinitionen und lässt sich dann eine Header Datei automatisch generieren. Andersrum ist es außerdem auch möglich Java Methoden aus C++ heraus aufzurufen.

Unter Interoperabilität kann man auch noch folgendes verstehen: Die Möglichkeit mehrere verwaltete Sprachen in eine einzige Zwischensprache zu übersetzen. Der Java Bytecode lässt dies nur

bedingt zu, da die Java Zwischensprache direkt auf Java zugeschnitten ist. Es gibt z.B. keine direkte Unterstützung für generische Programmierung, benutzerdefinierte Wertetypen oder Zeiger.

Die .NET Plattform verfolgte hingegen von Anfang an das Ziel der Interoperabilität. Deswegen ist das Verwenden von bereits bestehenden Codebasen hier wesentlich leichter. Die Common Intermediate Language hat nämlich einen großen Befehlssatz, der sogar die Implementation von funktionalen Programmiersprachen zulässt. Und auch das Ansprechen von unmanaged Code ist kein großes Problem, da mit C++/CLI (Managed C++) eine neue Sprache enstanden ist, die das ideale Bindeglied zwischen der verwalteten und unverwalteten Welt darstellt.

So können .NET Programme in dieser Sprache geschrieben werden und so alle Vorteile einer managed Umgebung genießen. Trotzdem ist es möglich den alten C++ Code fast unverändert zu übernehmen. Möchte man den alten Code aber mit C# oder Visual Basic .NET verwenden, so muss zuerst ein Wrapper in Managed C++ geschrieben werden. Die alten Klassen können dann wie ganz normale Typen im Code verwendet werden.

Weiterhin ist in C# auch das Verwenden von externen Funktionen aus dynamischen Bibliotheken (DLL) möglich. Dies kann dann nützlich sein, wenn man eine nicht vorhandene Funktion in der Klassenbibliothek direkt aus der Windows API ansprechen will. Diese Vorgehensweise sollte jedoch die allterletze Möglichkeit sein, da man somit plattformabhängige Funktionen verwendet.

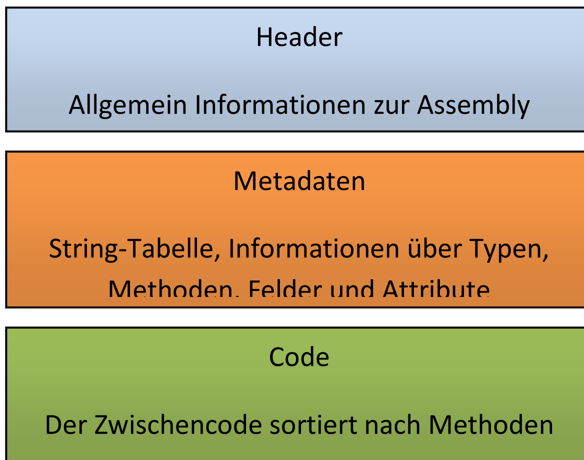
Ayox Intermediate Language

Im Rahmen dieser Arbeit entstand eine eigene virtuelle Maschine zusammen mit einer Zwischensprache, die auf den folgenden Seiten vorgestellt wird. Auf Grund des begrenzten Umfangs der Arbeit kann nicht auf alle Aspekte der Implementierung genau eingegangen werden.

Übersicht

Die Ayox Virtual Machine ist eine einfache stapelbasierte Maschine, welche die Ayox Intermediate Language (AIL) ausführt. Momentan besitzt die VM nur einen Klassenlader, einen abstrakten Prozessor sowie einen Garbage Collector. Ein vorrangiges Ziel bei der Entwicklung war die einfache Zusammenarbeit mit C++ Code, sodass die Ail beispielsweise auch als Skriptsprache für Anwendungen eingesetzt werden könnte.

Der Klassenlader lädt sogenannte Assemblies, die den jar Dateien in Java entsprechen. Eine Verifizierung des Codes findet nicht statt. Die Assemblies haben einen dreiteiligen Aufbau:



Die komplette Laufzeit lässt sich in zwei große Projekte einteilen. Einmal gibt es das Ail Framework, welches für das simple Laden und Speichern von Assemblies verantwortlich ist. Dann existiert die Ail Runtime, welche auf dem Framework aufsetzt. Diese Laufzeitumgebung verknüpft die Assemblies und führt sie aus. Der Garbage Collector gehört ebenso zu der Runtime.

Abstrakter Prozessor

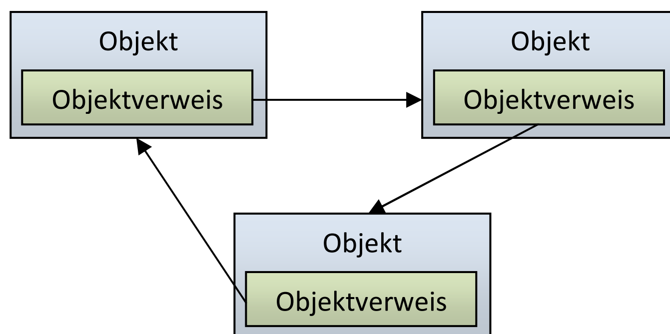
Der abstrakte Prozessor interpretiert den vorhandenen Zwischencode, da ein Just in Time Compiler nicht existiert. Anders als die Java VM oder die CLR besitzt die Ayox VM nur einen einzigen Stack für die Aufnahme der Variablen, Parameter und Berechnungswerte. Die beiden bereits existierenden Plattformen besitzen aber zusätzlich einen Auswertungsstack (Evaluation oder Operand Stack genannt).

Der Vorteil des Evaluation Stack liegt darin, dass die Variablen von den temporären Werten der Berechnung abgekoppelt sind. Außerdem kann am Anfang einer Methode der Typ jedes Slots auf

dem Stack eindeutig definiert werden (in Java ist dies aber nicht der Fall). In der CIL ist es daher nicht möglich am Anfang einer Methode eine ganze Zahl an der Position null und später eine Fließkommazahl zu speichern. Die AIL erlaubt dies, dadurch wird die Verifizierung des Codes jedoch erschwert. Weiterhin verkürzt sich der Code in der AIL im Gegensatz zum Java Bytecode oder zur CIL, da das Kopieren vom Operanden Stack in den anderen Stack entfällt.

Garbage Collector

Zur Realisierung eines Garbage Collectors gibt es prinzipiell zwei gänzlich verschiedene Ansätze. Der erste besteht darin, dass man die Verweise auf jedes Objekt während der Laufzeit verwaltet. Bei der Objekterzeugung steht der Zähler auf eins, weist man eine Objektreferenz einer anderen Referenz zu, so hat man zwei Verweise auf die Instanz und der Zähler muss auf zwei erhöht werden. Dies nennt sich Referenzzählung (Reference Counting). Der Nachteil dieser Methode besteht darin, dass ständig ein kleiner Overhead entsteht und dass zweitens zirkuläre Referenzen ohne großen Aufwand nicht erkannt werden. Auf der anderen Seite muss die Ausführung des Programms niemals gestoppt werden, was ein klarer Vorteil ist.



Zirkuläre Referenzen treten auf, wenn sich Objekte untereinander referenzieren. Da immer mindestens ein Objekt auf das andere zeigt, wird der Zähler des Objektes nie null und der Speicher wird nicht freigegeben.

Die zweite Art und Weise einen Garbage Collector zu implementieren besteht darin, die Programmausführung bei Speicherengpässen zu unterbrechen. Dann werden alle Objekte markiert, die erreichbar sind. Nicht markierte Objekte werden demnach nun nicht mehr benötigt und können verworfen werden.

Der Garbage Collector der Ail Runtime implementiert die zweite Variante, d.h. er ist ein Tracing Garbage Collector. Bei jeder Speicherreinigung muss nun also nach Referenzen auf dem Stack, in statischen Variablen und in Objekten gesucht werden.

Beispiel

Abschließend soll nun ein kleines Codebeispiel gezeigt werden, welches die Fakultät einer natürlichen Zahl berechnet. Die Signatur der Methode lautet `int fac(int number)`, sie ist

rekursiv implementiert. Dabei ist anzumerken, dass die Parameter einer Methode automatisch den ersten lokalen Variablen entsprechen. Der erste Parameter wird also so angesprochen als wäre es die nullte lokale Variable. Der Rückgabewert muss außerdem am Ende auf der Position null sitzen.

```
pushl 0         // nullte lokale Variable (number) auf den Stack pushen
load int32 0    // 0 auf den Stack laden
jmpequal ende   // springe zu finish, falls die oberen Werte gleich sind
pushl 0         // nullte lokale Variable (number) auf den Stack pushen
load int32 1    // 1 auf den Stack laden
jmpequal ende   // springe zu finish, falls die oberen Werte gleich sind

pushl 0         // nullte lokale Variable (number) auf den Stack pushen
dup             // Variable duplizieren
dec             // und dekrementieren, dann an Funktion übergeben..
call int fac(int)
mul             // number mit dem Rückgabewert multiplizieren
storel 0        // in Variable null speichern (Rückgabewert)
ret             // Funktion verlassen

ende:
load int32 1    // 1 auf den Stack laden
storel 0        // und in nullter Variable speichern
ret             // zu dem Zeitpunkt existiert nur eine Variable auf dem
                // Stack, dies ist das Ergebnis – springe nun zurück
```

Der Code prüft, ob der Parameter null oder eins ist, wenn ja wird eins zurückgegeben. Ansonsten wird der Parameter multipliziert mit `fac(number -1)` und dann zurückgegeben.

Abschluss

Die Arbeit hat gezeigt, dass virtuelle Maschinen in Verbindung mit verwalteten Sprachen eine sehr flexible und leistungsfähige Umgebung darstellen. Die selbst entwickelte virtuelle Maschine kann natürlich nicht mit der Geschwindigkeit ausgereifter Produkte mithalten, da ein Just in Time Compiler, eine Klassenbibliothek und ein Sicherheitsmanagement fehlt. Trotz alledem ist das Prinzip einer virtuellen Maschine an der AIL gut erkennbar.

Anhang

Entstehung der Arbeit

Schon seit Ende der elften Klasse stand für mich fest, dass ich eine schriftliche besondere Lernleistung als fünfte Prüfungskomponente schreiben will. Im Sommer 2005 gab es dann erste Überlegungen zu einem möglichen Thema. Zeitweise wollte ich über das Thema „Virtuelle 3D-Welten“ schreiben, was sich dann aber als zu komplex erwies. Aber nachdem ich mich in den Sommerferien mit Assembler und der Implementation einer kleinen virtuellen Maschine beschäftigt hatte, stand mein Thema fest: Virtuelle Maschinen.

Seit den Sommerferien 2005 habe ich dann versucht jede mir verfügbare Literatur über virtuelle Maschinen und Entwicklung von Laufzeitumgebungen zu lesen. Ende 2005 habe ich dann angefangen meine eigene Zwischensprache namens Ayox Intermediate Language zu entwickeln. Diese Sprache hat sich seit Ende 2005 stetig weiterentwickelt, sodass einige Veränderungen vollzogen wurden.

Anfang 2006 hat sich dann Herr Riefstahl bereit erklärt die Arbeit zu betreuen. In einem Gespräch im Februar habe ich dann meine Absichten genauer erläutert. So stand z.B. von Anfang an fest, dass ich eine eigene virtuelle Maschine mitsamt einer eigenen Zwischensprache entwickeln will.

Im Laufe des zweiten Halbjahres hat Herr Köstler unseren Informatikkurs übernommen und erfreulicherweise auch die Betreuung meiner schriftlichen Arbeit. Irgendwann im Mai habe ich ihm dann in mündlicher Form etwas genauer erklärt, was ich in die Arbeit schreiben möchte.

Nachdem ich mir in den Ferien schon ein paar Gedanken und Stichpunkte gemacht hatte, gab ich Herr Köstler einen schriftlichen Überblick über die virtuelle Maschine und die Zwischensprache, die ich zu dem Zeitpunkt noch „My Intermediate Language“ nannte.

In den Herbstferien im Jahr 2006 begann ich dann einen genaueren Plan aufzustellen mit Informationen, die ich niederschreiben wollte. In diesen zwei Wochen habe ich mich auch entschieden über alle Arten der virtuellen Maschinen zu schreiben. Bisher wollte ich lediglich über VMs für Anwendungen schreiben. Nachdem ein ungefährer Plan fertiggestellt war, begann ich dann mit abschließenden Recherchen zu der Geschichte der virtuellen Maschinen.

Erst Ende November begann ich dann mit der konkreten Arbeit an diesem Dokument. Am ersten Wochenende entstand der erste Teil über die Historie. Danach arbeitete ich vorrangig abends innerhalb der Woche an der Arbeit. Im Dezember wurde ich dann noch von Herr Köstler in Bezug auf die Form der Arbeit beraten, d.h. z.B. wie das Literaturverzeichnis anzufertigen ist.

Literaturverzeichnis

Fischereder, W. (2003). *Inside .NET and Java.* Retrieved August 2006, from The Common Intermediate Language: http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2003/reports/Fischereder/Fischereder.pdf

Gough, K. J. (2001). *Stacking them up: a comparison of virtual machines*. Retrieved November 23, 2006, from http://citeseer.ist.psu.edu/rd/25751164%2C499841%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/24233/http:zSzzSzsky.fit.qut.edu.auzSz%7EgoughzSzVirtualMachines.pdf/gough01stacking.pdf

Kommalapati, H., & Christian, T. (2005, Mai). *Drill Into .NET Framework Internals to See How the CLR Creates Runtime Objects.* Retrieved März 2006, from MSDN Magazine: http://msdn.microsoft.com/msdnmag/issues/05/05/JITCompiler/

License, G. F. (2006, Oktober 20). *http://de.wikipedia.org/wiki/Bytecode*. Retrieved November 23, 2006, from http://de.wikipedia.org/wiki/Bytecode

License, G. F. (2006, November 17). *Java Virtual Machine*. Retrieved November 23, 2006, from http://de.wikipedia.org/wiki/Java_Virtual_Machine

License, G. F. (2006, November 2). *P-Code*. Retrieved November 23, 2006, from http://de.wikipedia.org/wiki/P-Code

License, G. F. (2006, November 21). *Virtual machine*. Retrieved November 23, 2006, from http://en.wikipedia.org/wiki/Virtual_machine

License, G. F. (2006, Oktober 12). *Virtuelle Maschine*. Retrieved Oktober 24, 2006, from http://de.wikipedia.org/wiki/Virtuelle_Maschine

Meloan, S. (1999, Juni). *The Java HotSpot Performance Engine: An In-Depth Look*. Retrieved November 23, 2006, from http://java.sun.com/developer/technicalArticles/Networking/HotSpot/

Venners, B. (2006, Januar 9). *Bytecode basics*. Retrieved November 23, 2006, from http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html

Venners, B. (96, Januar 8). *Java's garbage-collected heap*. Retrieved November 22, 2006, from http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html?page=1

Abschließende Klausel

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe; ferner, dass diejenigen Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen sind, in jedem einzelnen Falle unter Angabe der Quelle als Entlehnung kenntlich gemacht sind. Soweit die Arbeit Zeichnungen, Kartenskizzen und bildliche Darstellung enthält, versichere ich ebenfalls, dass ich sie selbstständig angefertigt bzw. ihre Entnahme aus anderen Werken jeweils angegeben habe.

Berlin, den 15.12.2006 Georg Wächter