(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 699 22 015 T2** 2005.12.01

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 104 564 B1**
(21) Deutsches Aktenzeichen: **699 22 015.7**
(86) PCT-Aktenzeichen: **PCT/US99/18158**
(96) Europäisches Aktenzeichen: **99 939 131.1**
(87) PCT-Veröffentlichungs-Nr.: **WO 00/10081**
(86) PCT-Anmeldetag: **10.08.1999**
(87) Veröffentlichungstag der PCT-Anmeldung: **24.02.2000**
(97) Erstveröffentlichung durch das EPA: **06.06.2001**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **17.11.2004**
(47) Veröffentlichungstag im Patentblatt: **01.12.2005**

(51) Int Cl.[7]: **G06F 9/45**

(30) Unionspriorität:
134073 **13.08.1998** **US**

(84) Benannte Vertragsstaaten:
AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LI, LU, MC, NL, PT, SE

(73) Patentinhaber:
Sun Microsystems, Inc., Palo Alto, Calif., US

(72) Erfinder:
UNGAR, David, Palo Alto, US

(74) Vertreter:
BOEHMERT & BOEHMERT, 28209 Bremen

(54) Bezeichnung: **VERFAHREN UND VORRICHTUNG ZUM ÜBERSETZEN UND AUSFÜHREN VON ARTEIGENEM CODE IN EINER UMGEBUNG MIT VIRTUELLEN MASCHINEN**

Beschreibung

ALLGEMEINER STAND DER TECHNIK

1. ERFINDUNGSGEBIET

[0001] Die Erfindung betrifft den Bereich der Computersysteme und insbesondere Laufzeitumgebungen virtueller Maschinen (siehe „Java Native Interface Specification", 16. Mai 1997, Sun Microsystems, Inc., Mountain View, CA).

[0002] Solaris, Sun, Sun Microsystems, das Sun-Logo, Java und alle Markenzeichen und Logos auf der Basis von Java sind Markenzeichen oder eingetragene Markenzeichen von Sun Microsystems, Inc., in den Vereinigten Staaten und anderen Staaten.

2. STAND DER TECHNIK

[0003] Die von Sun Microsystems® entwickelte Programmiersprache Java™ weist den Vorzug gegenüber anderen Programmiersprachen auf, dass sie ein „write once, run anywhere"™-Sprache ist. Die Programmiersprache Java stellt einen weitgehend plattformunabhängigen Apparat für Anwendungen oder „Applets" bereit, die in der Form von Bytecode-Klassendateien zu entwerfen, zu verteilen und auszuführen sind. Die Java-Virtuelle-Maschine führt die Auflösung der Bytecodes in dem von der erforderlichen Plattform abhängigen Befehlssatz aus, so dass alle Computerplattformen, welche eine Java-Virtuelle-Maschine enthalten, in der Lage sind, dieselben Bytecode-Klassendateien auszuführen. Werden Funktionen benötigt, die nicht durch die Programmiersprache Java unterstützt werden, dann kann eine Java-Anwendung, die in der virtuellen Maschine ausgeführt wird, native Codefunktionen aufrufen, die in angeschlossenen Bibliotheken implementiert sind. Ein nativer Code ist nicht Gegenstand der Java Programmierung und der Ausführungsbeschränkungen, so dass mehr an plattformspezifischer Programmierbarkeit auf Kosten eines weniger gut kontrollierten Ausführungsverhaltens erreicht wird. Eine Verarbeitungsumgebung für Java-Anwendungen und Applets sowie die Verwendung von nativen Codes werden nachfolgend ausführlicher beschrieben.

Die Verarbeitungsumgebung

[0004] Die Programmiersprache Java ist eine objektorientierte Programmiersprache, bei der jedes Programm eine oder mehrere Objektklassen und Schnittstellen umfasst. Anders als viele Programmiersprachen, in denen ein Programm in einen maschinenabhängigen, ausführbaren Programmcode kompiliert wird, werden die in der Programmiersprache Java geschriebenen Klassen in maschinenunabhängige Bytecode-Klassendateien kompiliert. Jede Klasse enthält Code und Daten in einem plattformunabhängigen Format, das als Klassendateiformat bezeichnet wird. Das Computersystem, welches als das Ausführungsmittel dient, enthält ein Programm, das als virtuelle Maschine bezeichnet wird, welche für das Ausführen des Codes in jeder Klasse verantwortlich ist.

[0005] Anwendungen können als autonome Java-Anwendungen oder als Java-„Applets" entworfen werden, die durch ein Applet-Tag in einem HTML-(hypertext markup language)-Dokument identifiziert und durch eine Browser-Anwendung geladen werden. Die mit einer Anwendung oder einem Applet verknüpften Klassendateien können auf dem lokalen Computersystem oder auf einem Server, auf den über ein Netz zugegriffen werden kann, gespeichert werden. Jede Klasse wird durch den „Klassen-Loader" nach Bedarf in die Java-Virtuelle-Maschine geladen.

[0006] Um einem Client den Zugriff auf Klassendateien von einem Server auf einem Netz zu gewähren, wird eine Webserver-Anwendung auf dem Server ausgeführt, um auf HTTP-(hypertext transport protocol)-Anforderungen anzusprechen, die URLs (universal resource locators) zu den HTML-Dokumenten enthalten, die auch als „Webseiten" bezeichnet werden. Wenn eine Browser-Anwendung, die auf einer Client-Plattform ausgeführt wird, ein HTML-Dokument empfängt (z.B. als ein Ergebnis einer Anforderung eines HTML-Dokuments durch Absenden eines URL an den Webserver), dann parst die Browser-Anwendung die HTML und initiiert automatisch das Herunterladen der spezifizierten Bytecode-Klassendateien, wenn er in dem HTML-Dokument auf ein Applet-Tag trifft.

[0007] Die Klassen eines Java-Applet werden auf Anforderung aus dem Netz (auf einem Server gespeichert) oder aus einem lokalen Dateisystem geladen, wenn während des Ausführens der Java-Applets zum ersten Mal Bezug darauf genommen wird. Die virtuelle Maschine lokalisiert und lädt jede Klassendatei, parst das Klassendateiformat, teilt Speicher für die verschiedenen Komponenten der Klasse zu und verbindet die Klasse mit anderen bereits geladenen Klassen. Dieser Prozess macht den Code in der Klasse durch die virtuelle Maschine leicht ausführbar.

[0008] Java-Anwendungen und Applets machen oft Gebrauch von Klassenbibliotheken. Klassen in den Klassenbibliotheken können das enthalten, was als „native Methoden" bezeichnet wird. Anwendungen und Applets können zuweilen auch Klassen enthalten, die native Methoden aufweisen. Eine native Methode spezifiziert das Schlüsselwort „nativ", den Namen der Methode, den Rückgabetyp der Methode und beliebige Parameter, die der Methode angepasst sind. Im Gegensatz zu einer „Standardmethode" (d.h.

einer nicht nativen Methode), die in der Programmiersprache Java geschrieben ist, gibt es keinen Rumpf zu einer nativen Methode in der jeweiligen Klasse. Die Programme einer nativen Methode werden vielmehr durch einen kompilierten nativen Code (d.h. den Code, der in der Programmiersprache C oder C++ geschrieben und in die Binärform kompiliert ist) durchgeführt, der in der Laufzeit dynamisch mit einer gegebenen Klasse in der virtuellen Maschine verbunden ist, wobei eine für die gegebene Plattform spezifische Verbindungseinrichtung verwendet wird, welche die verbundenen Bibliotheken unterstützt.

[0009] In der Solaris™- oder UNIX-Umgebung kann zum Beispiel die verbundene Bibliothek, welche die Binärform des nativen Codes enthält, als „Shared-Objekt"-Bibliothek implementiert werden, die als eine „.so"-Datei geschrieben ist. In einer Windows-Umgebung kann die verbundene Bibliothek die Form einer dynamisch verbundenen (oder dynamisch ladefähigen) Bibliothek annehmen, die als eine „.dll"-Datei geschrieben wird. Der native Code kann verwendet werden, um Funktionen auszuführen, die ansonsten durch die Programmiersprache Java nicht unterstützt würden, wie z.B. das Ankoppeln an eine spezialisierte Hardware (z.B. Display-Hardware) oder Software (z.B. Datenbanktreiber) einer gegebenen Plattform. Der native Code kann auch zum Beschleunigen von rechenaufwändigen Funktionen, wie z.B. des Rendering, verwendet werden.

[0010] Eine Klasse, die eine native Methode enthält, enthält auch einen Aufruf, die jeweilige verbundene Bibliothek zu laden:
System.loadLibrary(„Sample");
wobei „Sample" der Name der verbundenen Bibliothek ist, die gewöhnlich in einer Datei gespeichert ist, die in Abhängigkeit vom Host-Betriebssystem (z.B. UNIX, Windows usw.) als „libSample.so" oder „Sample.dll" bezeichnet wird. Die verbundene Bibliothek wird gewöhnlich zu dem Zeitpunkt geladen, zu dem die zugehörige Klasse in der virtuellen Maschine instantiiert wird.

[0011] Die verbundene Bibliothek des nativen Codes wird mit Stub- und Header-Informationen der zugehörigen Klasse kompiliert, um die verbundene Bibliothek zu befähigen, die Methodensignatur der nativen Methode in der Klasse zu erkennen. Die Implementierung der nativen Methode wird dann als eine native Codefunktion (wie z.B. eine C-Funktion) in der verbundenen Bibliothek bereitgestellt. In der Laufzeit, wenn die native Methode aufgerufen wird, wird die Steuerung auf die Funktion in der verbundenen Bibliothek übertragen, die der aufgerufenen Methode entspricht (z.B. durch Hinzufügen eines Nativmethoden-Rahmens auf den Nativmethoden-Stapel). Der native Code in der verbundenen Bibliothek führt die Funktion aus und gibt die Steuerung der Java-Anwendung oder dem Applet zurück.

[0012] Fig. 1 veranschaulicht die Kompilierungs- und Laufzeitumgebungen für ein Verarbeitungssystem. In der Kompilierungsumgebung erzeugt ein Software-Entwickler Quelldateien **100** (z.B. in der Programmiersprache Java), welche die Programmierer-lesbaren Klassendefinitionen einschließlich der Datenstrukturen, Methodenimplementierungen und Bezugnahmen auf andere Klassen enthalten. Die Quelldateien **100** werden einem Java-Compiler **101** bereitgestellt, der die Quelldateien **100** in kompilierte „.class"-Dateien **102** kompiliert, welche Bytecodes enthalten, die durch eine Java-Virtuelle-Maschine ausführbar sind. Die Bytecode-Klassendateien **102** werden auf einem Server gespeichert (z.B. in einem temporären oder permanenten Speicher) und sind für das Herunterladen über ein Netz verfügbar. Alternativ können die Bytecode-Klassendateien **102** in einem Verzeichnis auf der Client-Plattform lokal gespeichert werden.

[0013] Die Java-Laufzeitumgebung enthält eine Java-Virtuelle-Maschine (JVM) **105**, die in der Lage ist, die Bytecode-Klassendateien auszuführen und native Betriebssystem(„O/S")-Aufrufe zum Betriebssystem **109** auszuführen, wenn das bei der Ausführung notwendig ist. Die Java-Virtuelle-Maschine **105** stellt sowohl eine Abstraktionsebene zwischen der Maschinenunabhängigkeit der Bytecode-Klassen und dem maschinenabhängigen Befehlssatz der zugrundeliegenden Computer-Hardware **110** als auch die plattformabhängigen Aufrufe des Betriebssystems **109** bereit.

[0014] Der Klassen-Loader und Bytecode-Prüfer („Klassen-Loader") **103** ist verantwortlich für das Laden der Bytecode-Klassendateien **102** und erforderlichenfalls für das Unterstützen von Klassenbibliotheken **104** in die Java-Virtuelle-Maschine **105**. Der Klassen-Loader **103** überprüft auch die Bytecodes einer jeden Klassendatei, um für das richtige Ausführen und das Durchsetzen von Sicherheitsregeln zu sorgen. Im Kontext des Laufzeitsystems **108** führt entweder ein Interpreter **106** die Bytecodes direkt aus oder ein „Just-in-time"- (JIT)-Compiler **107** übersetzt die Bytecodes in den Maschinencode, so dass sie durch den Prozessor (oder Prozessoren) in der Hardware **110** ausgeführt werden können. Der native Code, z.B. in der Form einer verbundenen Bibliothek **111**, wird geladen, wenn eine Klasse (z.B. aus den Klassenbibliotheken **104**), welche die zugehörige native Methode enthält, in der virtuellen Maschine instantiiert ist.

[0015] Das Laufzeitsystem **108** der virtuellen Maschine **105** unterstützt eine generelle Stapelarchitektur. Die Art und Weise, wie diese generelle Stapelarchitektur durch die zugrundeliegende Hardware **110** unterstützt wird, wird durch die spezifische Implementierung der virtuellen Maschine bestimmt und spiegelt den Weg wider, auf dem die Bytecodes inter-

pretiert oder JIT-kompiliert werden. Andere Elemente des Laufzeitsystems schließen das Thread-Management (d.h. das Scheduling) und die Datenmüllsammelverfahren ein.

[0016] Fig. 2 veranschaulicht die Laufzeitdatenbereiche, welche die Stapelarchitektur im Laufzeitsystem **108** unterstützen. In Fig. 2 umfassen die Laufzeit-Datenbereiche **200** einen oder mehrere Threadbezogene Datenbereiche **207**. Jeder Thread-bezogene Datenbereich **207** umfasst ein Programmzählerregister (PC REG) **208**, ein Zeigerregister lokaler Variablen (VARS REG) **209**, ein Rahmen-Register (FRAME REG) **210**, ein Operandenstapel-Zeigerregister (OPTOP REG) **211**, einen Stapel **212** (z.B. für Standardmethoden) und optional einen Nativmethodenstapel **216**. Der Stapel **212** umfasst einen oder mehrere Rahmen **213**, die einen Operandenstapel **214** und lokale Variablen **215** enthalten. Der Nativmethoden-Stapel **216** umfasst einen oder mehrere Nativmethoden-Rahmen **217**.

[0017] Die Laufzeit-Datenbereiche **200** weisen außerdem einen gemeinsam genutzten Heap **201** auf. Der Heap **201** ist der Laufzeitdatenbereich, aus dem für alle Instanzen und Arrays Speicher zugewiesen wird. Der gemeinsam genutzte Heap **201** umfasst einen Methodenbereich, der von allen Threads gemeinsam genutzt wird. Der Methodenbereich **202** umfasst einen oder mehrere klassenbezogene Datenbereiche **203** zum Speichern von Informationen, die aus jeder geladenen Klassendatei extrahiert werden. Zum Beispiel können die klassenbezogenen Datenbereiche **203** Klassenstrukturen, wie z.B. einen Konstantenpool **204**, Feld- und Methodendaten **205** und einen Code für Methoden sowie Konstruktoren **206** umfassen.

[0018] Eine virtuelle Maschine kann viele Threads zur gleichzeitigen Ausführung unterstützen. Jeder Thread hat seinen eigenen Thread-bezogenen Datenbereich **207**. An einem beliebigen Punkt führt jeder Thread den Code einer einzelnen Methode, der „aktuellen Methode" für diesen Thread, aus. Ist die „aktuelle Methode" keine native Methode, dann enthält das Programmzählregister **208** die Adresse des Befehls der virtuellen Maschine, der aktuell ausgeführt wird. Ist die „aktuelle Methode" eine native Methode, dann ist der Wert des Programmzählregisters **208** unbestimmt. Das Rahmen-Register **210** zeigt auf die Lage der aktuellen Methode im Methodenbereich **202**.

[0019] Jeder Thread weist einen eigenen Stapel **212** auf, der zur gleichen Zeit wie der Thread erzeugt wird. Der Stapel **212** speichert einen oder mehrere Rahmen **213**, die mit den Standardmethoden verknüpft sind, welche durch den Thread aufgerufen werden. Die Rahmen **213** werden sowohl für das Speichern von Daten und Teilergebnissen verwendet als auch für das Ausführen des dynamischen Verbindens, die Rückgabe von Werten für Methoden und das Abfertigen von Ausnahmen. Jedes Mal, wenn eine Standardmethode aufgerufen wird, wird ein neuer Rahmen erzeugt und auf den Stapel geschoben, und ein existierender Rahmen wird vom Stapel entnommen und zerstört, wenn seine Methode abgeschlossen ist. Ein Rahmen, der durch einen Thread erzeugt wird, ist für diesen Thread lokal und im Normalfall kann auf ihn direkt kein Bezug durch irgendeinen anderen Thread genommen werden.

[0020] Nur ein Rahmen, nämlich der Rahmen für die Methode, die aktuell ausgeführt wird, ist an einem beliebigen Punkt in einem gegebenen Steuerungsthread aktiv. Dieser Rahmen wird als der „aktuelle Rahmen" bezeichnet, und seine Methode ist als „aktuelle Methode" bekannt. Ein Rahmen hört auf, aktuell zu sein, wenn seine Methode eine andere Methode aufruft oder wenn seine Methode abgeschlossen wird. Wird eine Methode aufgerufen, dann wird ein neuer Rahmen erzeugt, und er wird aktuell, wenn die Steuerung auf die neue Methode übergeht. Beim Methodenrücksprung gibt der aktuelle Rahmen die Ergebnisse seines Methodenaufrufs – sofern vorhanden – an den vorhergehenden Rahmen zurück. Der aktuelle Rahmen wird dann abgelegt, während der vorhergehende Rahmen zum aktuellen wird.

[0021] Jeder Rahmen **213** weist seinen eigenen Satz lokaler Variablen **215** und seinen eigenen Operandenstapel **214** auf. Das Zeigenegister der lokalen Variablen **209** enthält einen Zeiger auf die Basis eines Arrays von Wörtern, welche die lokalen Variablen **215** des aktuellen Rahmens enthalten. Das Operandenstapel-Zeigerregister **211** zeigt auf den Kopf des Operandenstapels **214** des aktuellen Rahmens. Die meisten virtuellen Maschinenbefehle entnehmen aus dem Operandenstapel des aktuellen 'Rahmens Werte, wirken auf sie ein und geben die Ergebnisse demselben Operandenstapel zurück. Der Operandenstapel **214** wird auch verwendet, um Argumente auf Methoden zu übertragen und Methodenergebnisse zu empfangen.

[0022] Der native Methodenstapel **216** speichert Nativmethoden-Rahmen **217** zur Unterstützung nativer Methoden. Jeder Nativmethoden-Rahmen stellt einen Mechanismus für die Thread-Ausführungssteuerung, Methodenargumente und Methodenergebnisse bereit, die zwischen den Standardmethoden und nativen Methoden zu übergeben sind, welche als native Codefunktionen in einer verbundenen Bibliothek implementiert sind.

[0023] Da die nativen Methoden durch den nativen Code eher in einer verbundenen Bibliothek und nicht wie eine Standardmethode in einer Klasse implementiert sind, sind die nativen Methoden nicht den Einschränkungen unterworfen, die durch die Pro-

grammiersprache Java und den Bytecode-Prüfer auferlegt werden. Das bedeutet, dass der native Code in einer verbundenen Bibliothek, anders als Bytecodes für kompilierte Java-Anwendungen und Applets, anfällig gegenüber einem unerwünschten und nicht zulässigen Verhalten sein kann, das sich über die Laufzeit hinweg ungeprüft fortsetzt. Zum Beispiel können im nativen Code wegen des Auftretens von „wilden" Zeigern (d.h. ein Zeiger, dessen Wert einen vorgeschriebenen Bereich überschreitet, wie z.B. ein Zeiger auf das neunte Element eines Achtelement-Arrays) und durch das Verwenden von Speicherzugriffsmechanismen, die ungeeignete (d.h. eingeschränkte oder bereichsüberschreitende) Speicherplätze adressieren können, Speicherzugriffsfehler vorkommen.

[0024] Das Verwenden nativer Methoden macht deshalb einen Bereich von Programmierfehlern möglich, die hauptsächlich auf dem Einsatz von Zeigern beruhen, welche das Debugging einer speziellen Virtuell-Maschine-Implementierung schwieriger machen.

[0025] Außerdem kann der native Code Spensystemaufrufe (d.h. Aufrufe, die eine nicht spezifizierte Zeitdauer auf ein äußeres Ereignis, das stattfinden wird, warten können) enthalten. Wenn eine virtuelle Maschine ihr eigenes Thread-Management und Scheduling implementiert, dann kann ein Sperrsystemaufruf, der auftritt, wenn die Steuerung auf eine native Codefunktion in einer verbundenen Bibliothek übertragen wurde, die Ausführung der gesamten virtuellen Maschine sperren.

[0026] Die meisten Virtuell-Maschine-Implementierungen vermeiden die mit dem nativen Code verknüpften Sperrprobleme durch Verwenden des „nativen Threading". Das bedeutet, dass multiple Threads der virtuellen Maschine und das Programm oder die Programme (d.h. Anwendungen und/oder Applets), welche die Maschine ausführt, als Threads der zugrundeliegenden Plattform, z.B. als UNIX-Threads, implementiert werden. In diesem Schema können die Threads der virtuellen Maschine gleichzeitig ausgeführt werden. Wenn jedoch das native Threading verwendet wird, dann muss die virtuelle Maschine die Kontrolle über das Thread-Scheduling an das zugrundeliegende Betriebssystem abtreten. Das native Threading hat somit zur Folge, dass das Verhalten des Threads vom Betriebssystem und der Hardware abhängig wird. Ein effektives Debugging von Fehlern, die durch die Gleichzeitigkeit bedingt sind, wird in einer Virtuell-Maschine-Implementierung problematisch, weil mit dem nativen Threading die relative Zeitsteuerung der Thread-Ausführung über die verschiedenen Betriebssysteme und Hardware-Plattformen hinweg variieren kann.

[0027] Die Fig. 3A und Fig. 3B sind Blockdiagramme, welche die Verwendung eines Thread in Laufzeitumgebungen veranschaulichen. Fig. 3A enthält eine virtuelle Maschine, die kein natives Threading verwendet. Fig. 3B enthält eine virtuelle Maschine, die das native Threading verwendet.

[0028] In Fig. 3A läuft das Betriebssystem **109** auf der Hardware **110**, und die virtuelle Maschine **105** läuft auf dem Betriebssystem **109**. In der virtuellen Maschine **105** werden mehrere Anwendungen und/oder Applets, wie z.B. Applet1 (**300**) und Applet2 (**301**), ausgeführt. Applet1 und Applet2 können für sich je eine oder mehrere Bytecode-Klassendateien umfassen. Mit Applet2 ist eine verbundene Bibliothek (LIB) verknüpft, um die nativen Methoden zu unterstützen. Die Bibliothek **302** wird zu der Zeit geladen und verbunden, in der die Klasse von Applet2, welche die verknüpften nativen Methoden enthält, in der virtuellen Maschine **105** instantiiert wird. Der native Code der Bibliothek **302** läuft direkt auf dem Betriebssystem **109**, welches die Bibliothekverbindungseinrichtung und die Hardware **110** unterstützt.

[0029] In der virtuellen Maschine **105** werden multiple Threads zur Ausführung abgearbeitet. Zum Beispiel kann Applet1 zwei Threads, T1 und T2, aufweisen, Applet2 kann zwei Threads, T5 und T6, aufweisen und die virtuelle Maschine selbst kann zwei Threads, T3 und T4, aufweisen, welche Prozesse der virtuellen Maschine, wie z.B. das Sammeln von Datenmüll, durchführen. Die Threads T1 – T6 werden durch den VM-Thread-Scheduler **303** in der virtuellen Maschine **105** verwaltet und terminiert. Der VM-Thread-Scheduler **303** wählt zum Beispiel auf Basis von Prioritäten und Zeitschlitzverfahren aus, welcher Thread aus der Gruppe T1 bis T6 der aktuell auszuführende Thread der virtuellen Maschine TVM auf der Ebene des Betriebssystems sein soll.

[0030] Die Java-Virtuelle-Maschine unterstützt das „kooperative Scheduling", wobei die Threads, die ausgeführt werden, anderen Threads Prozessonessourcen in bestimmten Intervallen oder dann übergeben, wenn eine mit der Ausführung des aktuellen Thread verknüpfte Verzögerung wahrscheinlich ist. Zum Beispiel kann ein Thread höherer Priorität eine Yield-Operation nutzen, um den aktuellen Thread zu verdrängen. Das Ausnutzen von Prozessonessourcen muss in den Standardmethoden nicht explizit programmiert werden. Die virtuelle Maschine kann Yields in den Übersetzungsprozess oder in den kompilierten Code an geeigneten Punkten bei der Ausführung, so z.B. bei Methodenaufrufen und innerhalb von Schleifen (z.B. bei Rückverzweigungen), einsetzen, um das kooperative Scheduling zu implementieren.

[0031] Das Betriebssystem **109** kann zu jeder beliebigen Zeit viele Threads, einschließlich des ausgewählten Virtuell-Maschine-Threads TVM, bedienen.

Zum Beispiel kann das Betriebssystem **109** Threads TA – TZ enthalten, die andere Anwendungen oder andere Prozesse des Betriebssystems unterstützen. Der OS-Thread-Scheduler **304** bestimmt, welcher Thread aus der Gruppe TA – TZ und TVM durch die zugrundeliegende Hardware **110** zu irgendeinem Zeitpunkt auszuführen ist. Wenn die Hardware **110** multiple Prozessoren unterstützt, dann können durch den OS-Thread-Scheduler **304** multiple Threads terminiert werden, um gleichzeitig auf verschiedenen Prozessoren ausgeführt zu werden.

[0032] In der Implementierung von Fig. 3A kann ein Virtuell-Maschine-Thread (z.B. T1 – T6) die Ausführungssteuerung auf eine verbundene Bibliothek (z.B. LIB **302**) übertragen, um eine Funktion für eine native Methode auszuführen, z.B. kann, wie in der Darstellung gezeigt, der Thread T6 eine native Methode von Applet2 aufrufen, die durch den nativen Code in Bibliothek **302** unterstützt wird. Der Thread T6 ist in der Lage, die Steuerung der Bibliothek **302** zu übergeben, weil der Thread T6 als Virtuell-Maschine-Thread TVM aktuell auf das Betriebssystem **109** übertragen wurde. Andere Threads der virtuellen Maschine müssen entsprechend dem kooperativen Scheduling auf den Thread T6 warten, um zugelassen zu werden.

[0033] Die Übergabe der Steuerung auf die Bibliothek **302** kann jedoch Ausführungsprobleme der virtuellen Maschine verursachen. Klassen, die in der virtuellen Maschine ausgeführt werden, rufen gewöhnlich nur Methoden von anderen Klassen auf und lösen in der Regel keine Aufrufe direkt auf das System aus. Der native Code jedoch kann in Abhängigkeit von seiner Funktion häufig sperrende Systemaufrufe auslösen: Da der native Code unabhängig als ein kompilierter Code in einer verbundenen Bibliothek ausgeführt wird, werden Interpreter und Compiler umgangen und können kein kooperatives Scheduling erzwingen, bis die Steuerung einer Standardmethode zurückgegeben wird. Die virtuelle Maschine muss sich deshalb auf den Programmierer des nativen Codes stützen, um explizite Yield()-Aufrufe im nativen Code bereitzustellen.

[0034] Löst der native Code der Bibliothek **302** einen Sperrsystemaufruf, wie z.B. einen I/O-Aufruf zum Herunterladen einer Datei, aus, dann sperrt der Thread T6 in der virtuellen Maschine und somit der Thread TVM auf der Betriebssystemebene so lange, bis der Systemaufruf abgeschlossen ist, d.h., bis das Herunterladen beendet ist. Die gesamte Ausführung der virtuellen Maschine ist für die Dauer des Systemaufrufs auch gesperrt, da die Ausführungssteuerung durch den nativen Code der Bibliothek **302** aufrechterhalten wird. Da die Sperrsystemaufrufe bis zu ihrem Abschluss eine relativ lange Zeit benötigen können, ist es unerwünscht, dass alle Threads der virtuellen Maschine **109** gleichermaßen gesperrt sind. Die Leistungsfähigkeit von Applet1, Applet2 und der virtuellen Maschine **105** kann durch Sperrsystemaufrufe der Bibliothek **302** verringert werden. Aus diesem Grund verwenden viele Virtuelle-Maschine-Implementierungen das native Threading wie in Fig. 3B dargestellt.

[0035] In Fig. 3B implementiert der VM-Thread-Scheduler **303** multiple Threads der virtuellen Maschine als Threads auf der Betriebssystemebene. Diese Threads werden als Threads TVM1 –TVMn gekennzeichnet. Der VM-Thread-Scheduler **303** bestimmt, welche Virtuell-Maschine-Threads (T1 – T6) dem Betriebssystem **109** zu einem gegebenen Zeitpunkt als OS-Threads TVM1 – TVMn übergeben werden. In dem extremen Fall, in dem jeder Thread der virtuellen Maschine **105** als ein individueller Thread des zugrundeliegenden Betriebssystems **109** implementiert wird, kann die virtuelle Maschine **105** auf ein Implementieren des VM-Thread-Schedulers **303** verzichten und kann sich vollständig auf den OS-Thread-Scheduler **304** für das Thread-Scheduling verlassen.

[0036] Die Implementierung von Fig. 3B erlaubt es, dass multiple Threads gleichzeitig in der virtuellen Maschine **105** aktiv sind. Das bedeutet, dass ein Sperrsystemaufruf durch den nativen Code der Bibliothek **302** nicht ein vollständiges Sperren der virtuellen Maschine **105** zur Folge hat. Vielmehr wird ein Thread aus der Gruppe TVM1 – TVMn, nämlich der Thread, welcher die Steuerung der Bibliothek **302** übergeben hat (d.h. der Betriebssystem-Thread, welcher dem Virtuell-Maschine-Thread T6 entspricht), gesperrt, aber der Rest der Threads TVM1 – TVMn ist zur Ausführung frei.

[0037] Durch das Implementieren multipler Threads der virtuellen Maschine als OS- oder native Threads überträgt die virtuelle Maschine **105** jedoch die Steuerung über das Scheduling der Threads in der virtuellen Maschine vom VM-Thread-Scheduler **303** gewissermaßen auf den OS-Thread-Scheduler **304**. Zwischen den Threads der virtuellen Maschine können wegen des relativen Mangels an Steuerung, die durch den VM-Thread-Scheduler **303** ausgeübt wird, Synchronisationsfehler auftreten. Die Angelegenheit wird dadurch erschwert, dass dann, wenn die virtuelle Maschine **105** und Applet1 und Applet2 auf einem unterschiedlichen Betriebssystem **109** und/oder einer unterschiedlichen Hardware **110** mit unterschiedlichen Zeitsteuerungsparametern und Schedulingprozessen ausgeführt werden, Synchronisationsfehler wegen des Zugreifens des nativen Threadings auf den OS-Thrad-Scheduler **304** nicht auftreten werden oder auf eine unterschiedliche Weise auftreten können. Somit lassen sich Fehler nicht einfach wiederholen, und das Debugging des Systems wird schwieriger.

Objektorientiertes Programmieren

[0038] Im Folgenden wird zu Zwecken der Bezugnahme eine allgemeine Beschreibung der Prinzipien des objektorientierten Programmierens gegeben. Das objektorientierte Programmieren ist ein Verfahren zum Erzeugen von Computerprogrammen durch Zusammenfassen bestimmter fundamentaler Baublöcke und durch Erzeugen von Beziehungen unter und zwischen den Baublöcken. Die Baublöcke in objektorientierten Programmsystemen werden als „Objekte" bezeichnet. Ein Objekt ist eine Programmeinheit, die eine Datenstruktur (eine oder mehrere Instanzvariable) und die Operationen (Methoden), welche die Daten nutzen oder beeinflussen können, zusammenfasst. Somit besteht ein Objekt aus Daten und einer oder mehreren Operationen oder Prozeduren, die an diesen Daten ausgeführt werden können. Das Zusammenfügen von Daten und Operationen in einen einheitlichen Block wird als „Kapselung" bezeichnet.

[0039] Ein Objekt kann angewiesen werden, eine seiner Methoden auszuführen, wenn es eine „Nachricht" empfängt. Eine Nachricht ist ein Befehl oder eine Anweisung, die an das Objekt gesendet wurde, eine bestimmte Methode auszuführen. Eine Nachricht besteht aus einer Methodenauswahl (z.B. Methodenname) und null oder mehreren Argumenten. Eine Nachricht teilt dem empfangenden Objekt mit, welche Operationen auszuführen sind.

[0040] Ein Vorteil des objektorientierten Programmierens ist die Art und Weise, in der die Methoden aufgerufen werden. Wird eine Nachricht an ein Objekt gesendet, dann ist es nicht nötig, dass die Nachricht dem Objekt vorschreibt, wie ein bestimmtes Verfahren auszuführen ist. Es ist lediglich nötig zu fordern, dass das Objekt die Methode ausführt. Das vereinfacht die Programmentwicklung beträchtlich.

[0041] Die objektorientierten Programmiersprachen beruhen vorwiegend auf einem „Klassen"-Schema. Ein Beispiel für ein auf Klassen beruhendes objektorientiertes Programmierschema wird allgemein beschrieben in „Smalltalk-80: The Language" von Adele Goldberg und David Robson, 1989 veröffentlicht von Addison-Wesley Publishing Company.

[0042] Eine Klasse legt einen Typ eines Objekts fest, das üblicherweise sowohl Felder (d.h. Variable) als auch Methoden für die Klasse enthält. Eine Objektklasse wird verwendet, um eine bestimmte Instanz eines Objekts zu erzeugen. Eine Instanz eines Objekts enthält die für die Klasse festgelegten Variablen und Methoden. Aus einer Objektklasse können multiple Instanzen derselben Klasse erzeugt werden. Jede Instanz, die aus der Objektklasse erzeugt wird, gilt als Instanz desselben Typs oder derselben Klasse.

[0043] Zur Veranschaulichung kann eine Objektklasse die Instanzvariablen „Name" und „Gehalt" und eine Methode „Gehalt bestimmen" einschließen. Für jeden Angestellten in einer Organisation können Instanzen der Angestellten-Objektklasse erzeugt oder instantiiert werden. Jede Objektinstanz gilt als eine vom Typ „Angestellter". Jede Angestellten-Objektinstanz enthält die Instanzvariablen „Name" und „Gehalt" sowie die Methode „Gehalt bestimmen". Die mit den Variablen „Name" und „Gehalt" verknüpften Werte in jeder Angestellten-Objektinstanz enthalten den Namen und das Gehalt eines Angestellten in der Organisation. An die Angestellten-Objektinstanz des Angestellten kann eine Nachricht gesendet werden, um die Methode „Gehalt bestimmen" aufzurufen, um das Gehalt des Angestellten (d.h. den Wert, der mit der Variablen „Gehalt" im Angestellten-Objekt des Angestellten verknüpft ist) zu verändern.

[0044] Es kann eine Hierarchie von Klassen derart definiert werden, dass eine Objektklassen-Definition eine oder mehrere Unterklassen aufweist. Eine Unterklasse erbt die Definition ihrer Eltern (und Großeltern usw.). Jede Unterklasse in der Hierarchie kann zum Verhalten, das durch ihre Elternklasse bestimmt ist, beitragen oder es verändern. Einige objektorientierte Programmiersprachen unterstützen die multiple Vererbung, wo eine Unterklasse eine Klassendefinition von mehr als einer Elternklasse erben kann. Andere Programmiersprachen, wie z.B. die Programmiersprache Java, unterstützen nur eine einfache Vererbung, in der eine Unterklasse darauf beschränkt ist, die Klassendefinition von nur einer Elternklasse zu erben. Die Programmiersprache Java stellt auch einen Mechanismus bereit, der als ein „Interface" bekannt ist, welcher einen Satz von Konstanten und abstrakten Methodenvereinbarungen umfasst. Eine Objektklasse kann die in einem Interface definierten abstrakten Methoden implementieren. Sowohl einfache als auch mehrfache Vererbung sind für ein Interface verfügbar. Das heißt, ein Interface kann eine Interface-Definition von mehr als einem Eltern-Interface erben.

[0045] Ein Objekt ist ein generischer Begriff, der in der objektorientierten Programmierumgebung verwendet wird, um auf einen Modul zu verweisen, der einen verwandten Code und verwandte Variablen enthält. Eine Softwareanwendung kann unter Verwendung einer objektorientierten Programmiersprache geschrieben werden, wobei die Funktionalität des Programms unter Verwendung von Objekten implementiert wird.

KURZDARSTELLUNG DER ERFINDUNG

[0046] Es ist eine Aufgabe der vorliegenden Erfindung, ein verbessertes Verfahren und eine verbesserte Vorrichtung für das Übersetzen und Ausführen eines nativen Codes in einem Computersystem be-

reitzustellen, welche Synchronisationsfehler zwischen multiplen Threads der virtuellen Maschine vermeiden.

[0047] Diese Aufgabe wird gemäß den unabhängigen Ansprüche 1, 11 und 19 gelöst.

[0048] Es werden ein Verfahren und eine Vorrichtung zum Übersetzen und Ausführen eines nativen Codes in einer Virtuell-Maschine-Umgebung bereitgestellt, um Zeigerüberprüfung, Thread-Steuerung und andere vorteilhafte Fähigkeiten zuzulassen. Das Debugging einer Virtuell-Maschine-Umgebung wird durch die Binärübersetzung des nativen Codes erleichtert, welcher eine größere Plattformunabhängigkeit und eine größere Kontrolle über das Thread-Management und -Scheduling ermöglicht sowie ein Identifizieren von Speicherzugriffsfehlern im nativen Code erlaubt. Wenn der native Code in einer Virtuell-Maschine-Umgebung auszuführen ist, dann wird der native Code in eine Zwischenform übersetzt. Diese Zwischenform wird bearbeitet, um zu bestimmen, wo Speicherzugriffs- und Sperrsystemaufrufe auftreten. Zulässigkeitsprüfungen werden in die Speicherzugriffsaufrufe eingefügt, um festzustellen, ob der Speicheranteil, auf den durch jeden Aufruf zugegriffen werden soll, im erlaubten Bereich liegt. Wilde Zeiger und andere Quellen von Speicherzugriffsfehlern, die mit dem nativen Code verknüpft sind, können so identifiziert werden. Sperrsystemaufrufe werden durch nichtsperrende Varianten ersetzt und „Yield"-Operationen können in die Systemaufrufe und Schleifen eingesetzt werden.

[0049] Der korrigierte native Code, der Speicherzugriffs-Zulässigkeitsprüfungen und nichtsperrende Systemaufrufe einschließt, wird durch die virtuelle Maschine kompiliert oder interpretiert, um die durch den nativen Code bestimmten Routinen auszuführen. Da der überarbeitete native Code andere Threads nicht sperrt, kann das Thread-Scheduling durch die virtuelle Maschine statt durch das zugrundeliegende Betriebssystem verwaltet werden, und ein kooperatives Scheduling kann ausgeführt werden.

KURZBESCHREIBUNG DER ZEICHNUNGEN

[0050] Fig. 1 ist ein Blockdiagramm von Kompilierungs- und Laufzeitumgebungen.

[0051] Fig. 2 ist ein Blockdiagramm der Laufzeit-Datenbereiche einer Ausführungsform einer virtuellen Maschine.

[0052] Fig. 3A ist ein Blockdiagramm einer Laufzeitumgebung, die eine virtuelle Maschine aufweist, welche multiple Applets und einen nativen Code unterstützt, die über eine verbundene Bibliothek implementiert sind.

[0053] Fig. 3B ist ein Blockdiagramm einer Laufzeitumgebung, die eine virtuelle Maschine aufweist, welche native Thread-Operationen verwendet.

[0054] Fig. 4 ist ein Blockdiagramm einer Ausführungsform eines Computersystems, das in der Lage ist, eine geeignete Ausführungsumgebung für eine Ausführungsform der Erfindung bereitzustellen.

[0055] Fig. 5 ist ein Flussdiagramm eines Binärübersetzungsprozesses entsprechend einer Ausführungsform der Erfindung.

[0056] Fig. 6A ist ein verallgemeinertes Flussdiagramm eines beispielhaften Ausführungsblocks, der den Binärübersetzungsprozess eines Blocks eines nativen Codes in eine Zwischenform entsprechend einer Ausführungsform der Erfindung veranschaulicht.

[0057] Fig. 6B veranschaulicht das verallgemeinerte Steuerungs-Flussdiagramm von Fig. 6A mit Modifikationen, die entsprechend einer Ausführungsform der Erfindung ausgeführt sind.

[0058] Fig. 7 ist ein Blockdiagramm eines Computersystems, das eine virtuelle Maschine aufweist, die eine Binärübersetzung des nativen Codes entsprechend einer Ausführungsform der Erfindung verwirklicht.

AUSFÜHRLICHE BESCHREIBUNG DER ERFINDUNG

[0059] Die Erfindung ist ein Verfahren und eine Vorrichtung zum Übersetzen und Ausführen eines nativen Codes in einer Virtuell-Maschine-Umgebung. In der folgenden Beschreibung werden zahlreiche spezifische Details dargelegt, um eine gründlichere Beschreibung der Ausführungsformen der Erfindung zu erreichen. Es ist jedoch für einen Fachmann offensichtlich, dass die Erfindung ohne diese spezifischen Details betrieben werden kann. In anderen Fällen wurden gut bekannte Merkmale nicht ausführlich beschrieben, um die Erfindung nicht unkenntlich zu machen.

[0060] Obwohl die Erfindung hier mit Bezug auf die Programmiersprache Java und die Java-Virtuelle-Maschine diskutiert wird, kann die Erfindung in einer beliebigen Virtuell-Maschine-Umgebung, welche native Methoden oder Funktionen einschließt, implementiert werden.

Ausführungsform der Computer-Ausführungsumgebung (Hardware)

[0061] Eine Ausführungsform der Erfindung kann als Computer-Software in der Form eines computerlesbaren Codes auf einem Mehrzweck-Computer,

wie z.B. dem in Fig. 4 veranschaulichten Computer **400**, oder in der Form von Bytecode-Klassendateien implementiert werden, die in einer auf einem derartigen Computer laufenden Java-Laufzeitumgebung ausführbar sind. Eine Tastatur **410** und eine Maus **411** sind an einen bidirektionalen Systembus **418** gekoppelt. Die Tastatur und die Maus führen die Nutzer-Eingabewerte in das Computersystem ein und übermitteln diese Nutzer-Eingabewerte dem Prozessor **413**. Andere geeignete Eingabegeräte können zusätzlich zu oder anstatt der Maus **411** und der Tastatur **410** verwendet werden. Die an den bidirektionalen Bus **418** gekoppelte I/O(Eingabe/Ausgabe)-Einheit **419** verkörpert solche I/O-Elemente wie einen Drucker, A/V(AudioNideo)-I/O usw.

[0062] Der Computer **400** enthält neben Tastatur **410**, Maus **411** und Prozessor **413** einen Videospeicher **414**, Hauptspeicher **415** und Massenspeicher **412**, die alle an den bidirektionalen Systembus **418** gekoppelt sind. Der Massenspeicher **412** kann sowohl feste als auch auswechselbare Medien, wie z.B. magnetische, optische oder magneto-optische Speichersysteme, oder irgendeine andere verfügbare Massenspeichertechnologie enthalten. Der Bus **418** kann zum Beispiel Adressleitungen zum Adressieren des Videospeichers **414** oder des Massenspeichers **415** enthalten. Der Systembus **418** enthält zum Beispiel auch einen Datenbus zur Datenübertragung zwischen und unter den Komponenten, wie z.B. Prozessor **413**, Hauptspeicher **415**, Videospeicher **414** und Massenspeicher **412**. Alternativ können Multiplex-Daten/Adress-Leitungen anstelle der separaten Daten- und Adressleitungen verwendet werden.

[0063] In einer Ausführungsform der Erfindung ist der Prozessor **413** ein von Motorola produzierter Mikroprozessor, wie z.B. der 680x0-Prozessor, oder ein von Intel produzierter Prozessor, wie z.B. der 80x86- oder Pentium-Prozessor, oder ein SPARC-Mikroprozessor von Sun Microsystems, Inc. Es kann jedoch ein beliebiger anderer geeigneter Mikroprozessor oder Mikrorechner verwendet werden. Der Hauptspeicher **415** enthält einen dynamischen Schreib-Lese-Speicher (DRAM). Der Video-Speicher **414** ist ein Video-Schreib-Lese-Speicher mit zwei Anschlüssen. Ein Anschluss des Video-Speichers **414** ist an den Video-Verstärker **416** gekoppelt. Der Video-Verstärker **416** wird als Treiber für den Kathodenstrahlröhren(CRT)-Rastermonitor **417** verwendet. Der Video-Verstärker **416** ist vom Stand der Technik her gut bekannt und kann durch jede geeignete Anlage realisiert werden. Diese Schaltung wandelt die im Videospeicher **414** gespeicherten Pixeldaten in ein Rastersignal um, das zur Verwendung durch den Monitor **417** geeignet ist. Der Monitor **417** ist ein Monitortyp, der für die Wiedergabe grafischer Bilder geeignet ist. Alternativ könnte der Videospeicher als Treiber für einen Flachbildschirm oder eine Flüssigkristallanzeige (LCD) oder irgendein anderes geeignetes Datenwiedergabegerät verwendet werden.

[0064] Der Computer **400** kann auch eine an den Bus **418** gekoppelte Kommunikationsschnittstelle **420** enthalten. Die Kommunikationsschnittstelle **420** stellt eine Zweiweg-Datenkommunikationskopplung über eine Netzverbindung **421** an ein lokales Netz **422** bereit. Ist die Kommunikationsschnittstelle **420** zum Beispiel eine dienstintegrierende digitale Netz(ISDN)-Karte oder ein Modem, dann stellt die Kommunikationsschnittstelle **420** eine Datenkommunikationsverbindung zum entsprechenden Telefonleitungstyp bereit, welche einen Teil der Netzverbindung **421** ausmacht. Ist die Kommunikationsschnittstelle **420** eine Lokalnetz(LAN)-Karte, dann stellt die Kommunikationsschnittstelle **420** eine Datenkommunikationsverbindung über die Netzverbindung **421** an ein kompatibles LAN bereit.

[0065] Die Kommunikationsschnittstelle **420** könnte auch ein Kabelmodem oder eine Funk-Schnittstelle sein. In einer solchen Ausführung sendet und empfängt die Kommunikationsschnittstelle **420** elektrische, elektromagnetische oder optische Signale, die digitale Datenströme mitführen, welche verschiedene Informationstypen verkörpern.

[0066] Die Netzverbindung **421** stellt eine Datenkommunikation durch ein oder mehrere Netze zu anderen Datengeräten bereit. Zum Beispiel kann die Netzverbindung **421** eine Verbindung durch das lokale Netz **422** zum lokalen Server-Computer **423** oder zu Dateneinrichtungen herstellen, die durch einen Internet-Dienstanbieter (ISP) **424** betrieben werden. ISP **424** wiederum stellt Datenkommunikationsdienste durch das weltweite Paketdatenkommunikationsnetz bereit, das jetzt gewöhnlich als „Internet" **425** bezeichnet wird. Lokales Netz **422** und Internet **425** nutzen beide elektrische, elektromagnetische oder optische Signale, welche digitale Datenströme tragen. Die Signale durch die verschiedenen Netze und die Signale auf der Netzverbindung **421** sowie durch die Kommunikationsschnittstelle **420**, welche die digitalen Daten vom und zum Computer **400** tragen, sind beispielhafte Formen von Trägerwellen, welche die Informationen transportieren.

[0067] Der Computer **400** kann durch das (die) Netz(e), die Netzverbindung **421** und die Kommunikationsschnittstelle **420** Nachrichten senden und Daten einschließlich des Programmcodes empfangen. In dem Beispiel Internet könnte ein entfernt liegender Server-Computer **426** einen angeforderten Code für ein Anwendungsprogramm über Internet **425**, ISP **424**, lokales Netz **422** und Kommunikationsschnittstelle **420** übertragen.

[0068] Der empfangene Code kann, sobald er empfangen wurde, durch den Prozessor **413** ausgeführt und/oder im Massenspeicher **412** oder einem ande-

ren Permanentspeicher zur späteren Ausführung gespeichert werden. Auf diese Weise kann der Computer **400** einen Anwendungscode in der Form einer Trägerwelle erhalten. Entsprechend einer Ausführungsform der Erfindung enthalten solche heruntergeladenen Anwendungen ein oder mehrere Elemente einer Laufzeitumgebung, wie z.B. die virtuelle Maschine, Klassen-Loader, Bytecode-Klassendateien, Klassenbibliotheken und die Vorrichtung zum Übersetzen und Ausführen des hier beschriebenen nativen Codes.

[0069] Der Anwendungscode kann durch irgendeine Form eines Computerprogrammprodukts verkörpert werden. Ein Computerprogrammprodukt umfasst ein Medium, das eingerichtet ist, einen computerlesbaren Code oder Daten zu speichern oder zu transportieren, oder in welches computerlesbarer Code oder Daten eingebettet sein können. Einige Beispiele für Computerprogrammprodukten sind CD-ROM-Platten, ROM-Karten, Disketten, Magnetbänder, Computer-Festplattenlaufwerke, Server in einem Netz und Trägerwellen.

[0070] Die oben beschriebenen Computersysteme dienen nur als Beispiel. Eine Ausführungsform der Erfindung kann in irgendeinem Typ von Computersystem oder Programmier- oder Verarbeitungsumgebung implementiert werden, wobei die eingebetteten Geräte (z.B. Web-Telefone usw.) und „Dünn"-Client-Verarbeitungsumgebungen (z.B. Netzcomputer (NC) usw.), die eine virtuelle Maschine unterstützen, eingeschlossen sind.

Binärübersetzung des nativen Codes

[0071] Wie zuvor beschrieben wurde, können die in einer virtuellen Maschine ausgeführten Klassen native Methoden enthalten, welche durch native Codefunktionen in einer verbundenen Bibliothek implementiert werden. Entsprechend einer Ausführungsform der Erfindung wird der native Code der verbundenen Bibliothek durch Komponenten der virtuellen Maschine verarbeitet und ausgeführt, um ein kooperatives Scheduling zu erlauben und Debugging-Fähigkeiten bereitzustellen, die gegenüber den Ausführungsprozessen nativer Methoden vom Stand der Technik erweitert sind. Die Verarbeitung des nativen Codes schließt als ein Teil eines Binärübersetzungsprozesses das Einfügen von Überprüfungen von Speicherzugriffsfehlern, die z.B. durch „wilde" Zeiger erzeugt sein könnten, und das Ersetzen von Sperrsystemaufrufen durch nicht-sperrende Varianten ein, um ein kooperatives Scheduling in der virtuellen Maschine zu erlauben, ohne das native Threading zu benötigen.

[0072] Die Binärübersetzung wird in der virtuellen Maschine gewöhnlich während der Debugging-Operationen aktiviert und während des normalen Betriebs deaktiviert. Zum Beispiel hat ein Aufruf „System.loadLibrary()", wenn er aktiviert wird, die Binärübersetzung der spezifizierten Bibliothek für eine interpretierte oder kompilierte Ausführung innerhalb der virtuellen Maschine zur Folge. Bei Deaktivierung wird die spezifizierte Bibliothek geladen und standardgemäß verbunden. In einigen Ausführungsformen ist es auch möglich, dass die Binärübersetzung die gesamte Zeit über statt nur während der Debugging-Prozesse ausgeführt wird.

[0073] Fig. 5 ist ein Flussdiagramm eines Verfahrens zum Ausführen einer Binärübersetzung entsprechend einer Ausführungsform der Erfindung. In Schritt **500** wird der jeweilige native Code aus der verbundenen Bibliothek erhalten. Dieser Schritt kann zum Beispiel die Bestimmung der Quelldatei aus der verbundenen Bibliothek umfassen, wenn die jeweilige Klasse in der virtuellen Maschine instantiiert wird, und er kann das Lesen der Binärform des nativen Codes (d.h. des Maschinencodes) aus der Quelldatei umfassen. Die Binärübersetzung kann auch vor der Ausführung in der virtuellen Maschine vorgenommen werden.

[0074] In Schritt **501** wird die Binärform des nativen Code durch eine Binärübersetzungskomponente der virtuellen Maschine in eine Zwischenform, wie z.B. Bytecodes, einen abstrakten Syntaxbaum oder einen Ablaufsteuerungsgraphen, übersetzt. Die Bytecodes können ähnlich wie die Standard-Bytecodes, die zum Beispiel durch einen Java-Compiler (Element **101** von Fig. 1) erzeugt wurden, implementiert werden. Abstrakte Syntaxbäume und Ablaufsteuerungsgraphen sind Darstellungen der Programmausführung, welche die Ausführungsoperationen als Knoten eines Baums oder Graphen spezifizieren. Gewöhnlich ist die Zwischenform (die hier auch als die „übersetzte Form" bezeichnet wird) eine, welche das Identifizieren der Speicherzugriffspunkte und/oder -Aufrufe und Verzweigungsoperationen vereinfacht.

[0075] In Schritt **502** werden die Orte der Speicherzugriffsaufrufe bestimmt, und es werden in das Signal Überprüfungen während des Ausführens eingefügt, wenn der Speicherzugriffsaufruf versucht, auf einen Teil eines Speichers zuzugreifen, der eingeschränkt oder anderweitig bereichsüberschreitend ist. Das Signal kann zum Beispiel das Anzeigen einer Fehlernachricht (z.B. in einer Dialogbox), das Protokollieren eines Fehlers in einer Log-Datei, das Auslösen einer Ausnahme oder irgendeine Kombination der vorangehenden Maßnahmen umfassen. Das Debugging von wilden Zeigern und anderen Speicherzugriffsfehlern, die mit dem nativen Code verknüpft sind, kann deshalb während der Ausführung ermittelt werden, indem jedes unzulässige Speicherzugriffsereignis dann angezeigt wird, wenn es auftritt. Als Teil der Speicherzugriffsüberprüfungen werden die Anteile des Speichers, auf die ein Zugriff durch

Elemente des nativen Codes zulässig ist, für den Vergleich mit Zeigerwerten aufgespürt.

[0076] Im Schritt **503** werden Sperrsystemaufrufe in der Zwischenform identifiziert und es werden, wo es möglich ist, nicht-sperrende Varianten der Systemaufrufe eingesetzt. In Schritt **504** werden „Yield"-Funktionen in Aufrufe und Schleifen eingesetzt. Die Yield-Punkte (d.h. die Punkte für das Einsetzen von Yield()-Funktionen) für Schleifen können zum Beispiel auf Basis von Rückverzweigungs-Operationen bestimmt werden. Die Schritte **503** und **504** bewirken, dass das Ausführen der virtuellen Maschine und beliebiger laufender Anwendungen und/oder Applets so weit wie möglich frei wird von der Abhängigkeit von den Aktivitäten des nativen Codes. Die anderen Threads der virtuellen Maschine werden nicht durch Systemaufrufe des nativen Codes gesperrt, und bei den Aufrufen sowie in den Schleifen werden Yield-Punkte eingerichtet, um anderen wartenden Threads Verarbeitungsressourcen zu liefern. Die virtuelle Maschine ist somit in der Lage, ein kooperatives Scheduling von allen zugeordneten Threads auszuführen. Dieses kooperative Scheduling ermöglicht eine Synchronisation oder ein zuverlässiges Identifizieren und Korrigieren von durch Gleichzeitigkeit bedingten Fehlern unabhängig vom zugrundeliegenden Betriebssystem und der Hardware.

[0077] Im Schritt **505** wird der revidierte native Code in seiner Zwischen- oder übersetzten Form durch die virtuelle Maschine kompiliert oder interpretiert, um die Funktionen darin auszuführen. In einigen Ausführungsformen kann ein weiterer Übersetzungsschritt ausgeführt werden, um die Zwischenform in Bytecodes zu übersetzen, die für das Interpretieren oder Kompilieren durch den Standard-Interpreter oder JIT-Compiler geeignet sind. Das Scheduling von Threads, die mit dem Ausführen von nativen Codefunktionen verknüpft sind, kann durch den VM-Thread-Scheduling-Prozess wie ein beliebiger anderer interpretierter oder kompilierter Prozess der virtuellen Maschine gesteuert werden. Da die Speicherzugriffsüberprüfungen ausgeführt werden, werden Verletzungen protokolliert. Wo es nötig ist, können separate asynchrone Threads erzeugt werden, um unabhängig von den anderen Threads in der virtuellen Maschine zuzulassen, dass eine weitere Verarbeitung stattfindet.

[0078] In einigen Beispielen kann der native Code in Schritt **501** vollständig geparst werden. Bestimmte Aspekte des Codes, wie z.B, der Beginn einer Routine oder einer berechneten Verzweigung, können unbekannt sein, bis der übersetzte Code in Schritt **505** ausgeführt wird (d.h., wenn die Routine tatsächlich aufgerufen wird). Aus diesem Grunde kann der Übersetzungsprozess vom Schritt **505** zum Schritt **501** zurückkehren, wie es durch den Rückkopplungspfeil **506** angezeigt ist, um auf Basis der neuen Informationen, die während des Ausführens bestimmt wurden, den zuvor nicht geparsten nativen Code zu parsen und zu übersetzen (oder Teile des bereits geparsten Codes erneut zu parsen).

[0079] In den Fig. 6A und Fig. 6B ist ein Beispiel einer verallgemeinerten Zwischenform einer nativen Methode dargestellt, die eine Binärübersetzung durchläuft. Fig. 6A ist ein verallgemeinertes Steuerungs-Flussdiagramm eines beispielhaften Ausführungsblocks, das eine Binärübersetzung eines Blocks eines nativen Codes in eine Zwischenform entsprechend einer Ausführungsform der Erfindung veranschaulicht. Fig. 6B veranschaulicht das verallgemeinerte Steuerungs-Flussdiagramm von Fig. 6A mit Modifikationen, die entsprechend einer Ausführungsform der Erfindung ausgeführt sind. In dieser Ausführungsform beruht das Identifizieren von Yield-Punkten in Schleifen auf dem Auftreten von Rückverzweigungsoperationen.

[0080] Zum Kennzeichnen der dargestellten Operationen dient in Fig. 6A und Fig. 6B die folgende Legende:

RD	= Speicherleseoperation
WR	= Speicherschreiboperation
BR	= Verzweigungsoperation (z.B. „if")
MC	= Methoden-(Funktions-)Aufruf
BSC	= Sperrsystemaufruf
OP	= andere allgemeine Operation (sonstiges)
CHK	= Zeigerüberprüfungsoperation
YLD	= Yield-Operation
NBSC	= nicht-sperrender Systemaufruf
FLAG	= Signalzugriffsverletzung

[0081] In Fig. 6A beginnt der Ausführungsblock mit einer allgemeinen Operation **600** gefolgt von einer Leseoperation **603**. Auf die Leseoperation **603** folgen nacheinander die allgemeinen Operationen **605**, **606** und **607**. Nach der allgemeinen Operation **607** wird eine Schreiboperation **610** ausgeführt, gefolgt von den allgemeinen Operationen **612** und **613** sowie der Verzweigungsoperation **614**. Die Verzweigungsoperation springt entweder nach vorn zur allgemeinen Operation **616**, oder sie springt zurück zur allgemeinen Operation **606**. Von der allgemeinen Operation **616** wird der Methodenaufruf **618** ausgeführt, gefolgt von der allgemeinen Operation **619**, dem Sperrsystemaufruf **621A** und der allgemeinen Operation **622**.

[0082] Die Operationen, die für den Binärübersetzungsprozess von Interesse sind, sind die Leseoperation **603**, die Schreiboperation **610**, die Verzweigungsoperation **614**, der Methodenaufruf **618** und der Sperrsystemaufruf **621A**, die alle hervorgehoben sind. Die Leseoperation **603** und die Schreiboperation **610** werden als speicherzugriffsbezogene Operationen für das Einfügen von Zeigerüberprüfungen be-

stimmt. Die Verzweigungsoperation **614** und der Methodenaufruf **618** werden für das Einfügen von Yield()-Aufrufen bestimmt. Der Sperrsystemaufruf **621A** ist für das Ersetzen durch eine nichtsperrende Systemaufrufvariante bestimmt.

[0083] In Fig. 6B werden die Modifikationen für der Zwischenform des Ausführungsblocks von Fig. 6A veranschaulicht. Die Leseoperation **603** wird durch die Zeigerüberprüfungsoperation **601**, die Verzweigungsoperation **602**, die Leseoperation **603** und die Flag-Operation **604** ersetzt. Die Überprüfungsoperation **601** bestimmt, ob der Zeigerwert im zulässigen Bereich liegt, und auf sie folgt die Verzweigungsoperation **602**. Die Verzweigungsoperation **602** führt entweder die Leseoperation **603** aus, wenn der Zeiger zulässig ist, oder sie führt eine Flag-Operation **604** aus, um zu melden, dass die Zeigerüberprüfung einen unzulässigen Zeiger anzeigt. Die Operationen **603** und **604** führen beide zur Operation **605**.

[0084] Ähnlich zu dem oben beschriebenen Einfügen, das für die Leseoperation **603** durchgeführt wird, ist die Leseoperation **610** Gegenstand einer Überprüfung und Flag-Einfügung. Die Leseoperation **610** wird durch die Zeigerüberprüfungsoperation **608**, die Verzweigungsoperation **609**, die Leseoperation **610** und die Flag-Operation **611** ersetzt. Die Überprüfungsoperation **608** bestimmt, ob der Zeigerwert im zulässigen Bereich liegt, und auf sie folgt die Verzweigungsoperation **609**. Die Verzweigungsoperation **609** führt entweder die Schreiboperation **610** aus, wenn der Zeiger zulässig ist, oder sie führt eine Flag-Operation **611** aus, um zu melden, dass die Zeigerüberprüfung einen unzulässigen Zeiger anzeigt. Die Operationen **610** und **611** führen beide zur Operation **612**.

[0085] Die Rückverzweigungsoperation **614** weist eine in die Rücksprungschleife zur Operation **606** eingefügte Yield-Operation **615** auf. Die eingefügte Yield-Operation **615** lässt für andere Threads die Möglichkeit zu, Prozessorressourcen zu erhalten, bevor die Schleife, die durch die Verzweigungsoperation **614** gebildet wird, noch einmal beginnt. Das verhindert, dass ein langer Schleifen-Rekursionsprozess den anderen Threads die Prozessonessourcen wegnimmt und fördert das kooperative Scheduling. Auf die gleiche Weise wird die Yield-Operation **617** vor dem Methodenaufruf **618** eingefügt, um nötigenfalls ein Ausführen anderer Threads zu erlauben, bevor eine neue Methode durch den aktuellen Thread initiiert ist.

[0086] Der Sperrsystemaufruf **621A** wird in Fig. 6B durch einen nicht-sperrenden Systemaufruf **621B** ersetzt. Wahlweise kann eine Yield-Operation **620** vor dem Systemaufruf eingefügt werden. Wenn nötig kann ein nicht-sperrender Systemaufruf **621B** einen neuen asynchronen Thread erzeugen, um Aktivitäten der übersetzten Funktion als ein unabhängig ausgeführter Thread weiterzuführen. Wenn er durch die virtuelle Maschine interpretiert oder kompiliert ist, bietet der revidierte Ausführungsblock von Fig. 6B bedeutende Vorteile beim Debugging und Scheduling gegenüber der nativen Methodenausführung vom Stand der Technik.

[0087] Fig. 7 ist ein Blockdiagramm, das eine Laufzeitumgebung veranschaulicht, die eine Binärübersetzung entsprechend einer Ausführungsform der Erfindung implementiert. In Fig. 7 läuft das Betriebssystem **109** über der Hardware **110**, und die virtuelle Maschine **105** läuft über dem Betriebssystem **109**. Die Ausführung des Betriebssystems **109** wird durch die Hardware **110** unterstützt. Wie in Fig. 3A und Fig. 3B umfassen die virtuelle Maschine **105** und das Betriebssystem **109** den VM-Thread-Scheduler **303** bzw. den OS-Thread-Scheduler **304** zum Verwalten der Thread-Ausführung. Zusätzlich umfasst die virtuelle Maschine **105** den Binärübersetzungsprozess **701**.

[0088] In der virtuellen Maschine **105** werden multiple Anwendungen und/oder Applets, wie z.B. Applet1 (**300**) und Applet2 (**301**), ausgeführt. Applet1 und Applet2 können jedes eine oder mehrere Bytecode-Klassendateien umfassen. Eine verbundene Bibliotheksdatei (LIB) **302** ist mit Applet2 verknüpft, um native Methoden zu unterstützen. Der native Code der Bibliotheksdatei **302** ist geparst und durch den Binärübersetzungsprozess **701** der virtuellen Maschine **105** übersetzt, um die übersetzte Bibliothek **700** zu erzeugen.

[0089] Die übersetzte Bibliothek **700** umfasst die Zwischenform des nativen Codes einschließlich von Speicherzugriffsüberprüfungen, Yields und nicht-sperrenden Aufrufvarianten. Wird eine native Methode von Applet2 durch den Thread T6 aufgerufen, dann wird die übersetzte Bibliothek **700** in der virtuellen Maschine **105** interpretiert oder kompiliert, um die gewünschte Funktion auszuführen. In Abhängigkeit von der Zwischenform der übersetzten Bibliothek **700** kann sich der Interpretations- oder Kompilierungsprozess für die übersetzte Bibliothek von dem Interpretations- und Kompilierungsprozess, der auf die Klassen von Applet1 oder Applet2 angewendet wird, unterscheiden oder nicht unterscheiden. Die allgemeine Handhabung der und die ausgeübte Kontrolle über die Bibliothek **700** durch die virtuelle Maschine **105** entsprechen jedoch denen von Applet1 und Applet2. In einigen Ausführungsformen kann die übersetzte Bibliothek **700** im Thread T6 über Rahmen im Stapel **212** statt über den Nativmethoden-Stapel **216** verarbeitet werden. Tatsächlich kann die übersetzte Bibliothek **700** ausgeführt werden, als ob die übersetzte Bibliothek **700** zusätzliche Standardmethoden ohne die Mängel des unveränderten nativen Codes bereitstellen würde.

[0090] Da die übersetzten Nativcode-Funktionen der verbundenen Bibliothek durch die virtuelle Maschine **105** statt eines separaten verbundenen Bibliotheksprozesses ausgeführt werden, der durch das Betriebssystem **109** durchgeführt wird, und weil Sperraufrufe in der übersetzten Bibliothek **700** nicht vorkommen, kann ein kooperatives Scheduling durch den VM-Thread-Scheduler **303** vollzogen werden. Somit ist das native Threading, wie es in Fig. 3B implementiert ist, nicht notwendig. Die Synchronisation von Thread-Ereignissen in der virtuellen Maschine **105** ist unabhängig vom zugrundeliegenden Betriebssystem und der Hardware, und ein Debugging kann ohne Berücksichtigung der auf dem Betriebssystem beruhenden Gleichzeitigkeitsprobleme durchgeführt werden.

[0091] Somit wurden in Verbindung mit einer oder mehreren spezifischen Ausführungsformen ein Verfahren und eine Vorrichtung zum Übersetzen und Ausführen eines nativen Codes in einer Virtuell-Maschine-Umgebung beschrieben. Die Erfindung ist durch die Ansprüche und durch deren vollständigen Umfang an Äquivalenten festgelegt.

Patentansprüche

1. Verfahren zum Übersetzen und Ausführen eines nativen Codes in einem Computersystem (**400**), wobei das Verfahren ein Erlangen eines nativen Codes aus einer Bibliothek (**302**) umfasst, gekennzeichnet durch ein Parsen des nativen Codes in eine Zwischenform, ein Verarbeiten der Zwischenform in eine übersetzten Form (**700**), die ein kooperatives Scheduling aller Threads in einer virtuellen Maschine (**105**) ermöglicht, und ein Ausführen der übersetzten Form (**700**) des nativen Codes.

2. Verfahren nach Anspruch 1, bei dem das Verarbeiten in einer virtuellen Maschine (**105**) durchgeführt wird.

3. Verfahren nach Anspruch 1, bei dem das Verarbeiten ein Identifizieren eines Sperrsystemaufrufs in der Zwischenform und ein Ersetzen des Sperrsystemaufrufs (**621A**) durch eine nicht-sperrende Variante des Systemaufrufs (**621B**) umfasst.

4. Verfahren nach Anspruch 1, bei dem das Verarbeiten ein Identifizieren eines Speicherzugriffvorgangs in dem Zwischenformat und ein Hinzufügen einer Überprüfung auf eine Speicherzugriffsverletzung zu dem Speicherzugriffvorgang umfasst.

5. Verfahren nach Anspruch 2, bei dem das Ausführen der übersetzten Form weiterhin ein Kompilieren der übersetzten Form (**700**) durch die virtuelle Maschine (**105**) umfasst.

6. Verfahren nach Anspruch 2, bei dem das Ausführen der übersetzten Form (**700**) ein Interpretieren der übersetzten Form (**700**) durch die virtuelle Maschine (**105**) umfasst, um die Funktionen darin auszuführen.

7. Verfahren nach Anspruch 1, bei dem die Verarbeitung ein Identifizieren eines Yield-Punktes in der Zwischenform und ein Einfügen einer Yield-Funktion an dem Yield-Punkt umfasst.

8. Verfahren nach Anspruch 7, bei dem der Yield-Punkt ein Methodenaufruf umfasst.

9. Verfahren nach Anspruch 7, bei dem der Yield-Punkt eine Schleife umfasst.

10. Verfahren nach Anspruch 9, bei dem die Schleife ein Identifizieren einer Rückverzweigung umfasst.

11. Computerprogrammprodukt, das ein computernutzbares Medium mit einem computerlesbaren Code, der darin enthalten ist, zum Übersetzen und Ausführen eines nativen Codes umfasst, wobei das Computerprogrammprodukt einen computerlesbaren Code umfasst, der dazu eingerichtet ist, ein Computer zu veranlassen, einen nativen Code aus einer Bibliothek (**302**) zu erlangen, gekennzeichnet durch computerlesbaren Code, der dafür eingerichtet ist, einen Computer zu veranlassen, den nativen Code in eine Zwischenform zu parsen, computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die Zwischenform in eine übersetzte Form (**700**) zu verarbeiten, die ein kooperatives Scheduling aller Threads in einer virtuellen Maschine (**105**) erlaubt, und computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die übersetzte Form (**700**) des nativen Codes auszuführen.

12. Computerprogrammprodukt nach Anspruch 11, bei dem der computerlesbare Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die übersetzte Form (**700**) auszuführen, in einer virtuellen Maschine (**105**) ausgeführt wird.

13. Computerprogrammprodukt nach Anspruch 11, bei dem der computerlesbare Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die Zwischenform zu verarbeiten, einen computerlesbaren Code umfasst, der dazu eingerichtet ist, einen Computer zu veranlassen, einen Sperrsystemaufruf (**621A**) in der Zwischenform zu identifizieren, und einen computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, den Sperrsystemaufruf durch eine nicht-sperrende Variante (**621B**) des Systemaufrufs zu ersetzten.

14. Computerprogrammprodukt nach Anspruch 11, bei dem der computerlesbare Code, der dazu eingerichtet ist einen Computer, zu veranlassen, die

Zwischenform zu verarbeiten, folgendes umfasst:
einen computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, einen Speicherzugriffsvorgang in der Zwischenform zu identifizieren, und einen computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, eine Überprüfung nach einer Speicherzugriffsverletzung zu dem Speicherzugriffsvorgang hinzuzufügen.

15. Computerprogrammprodukt nach Anspruch 11, bei dem der computerlesbare Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die Zwischenform zu verarbeiten, einen computerlesbaren Code umfasst, der dazu eingerichtet ist, einen Computer zu veranlassen, in dem Zwischenformat einen Yield-Punkt zu identifizieren, und einen computerlesbaren Code, der dazu eingerichtet ist, einen Computer zu veranlassen, eine Yield-Funktion an dem Yield-Punkt einzufügen.

16. Computerprogrammprodukt nach Anspruch 15, bei dem der computerlesbare Code, der dazu eingerichtet ist, einen Computer zu veranlassen, den Yield-Punkt zu identifizieren, einen computerlesbaren Code umfasst, der dazu eingerichtet ist, einen Computer zu veranlassen, ein Methodenaufruf zu identifizieren.

17. Computerprogrammprodukt nach Anspruch 15, bei dem der computerlesbare Code, der dazu eingerichtet ist, den Computer zu veranlassen, einen Yield-Punkt zu identifizieren, einen computerlesbaren Code umfasst, der dazu eingerichtet ist, einen Computer zu veranlassen, eine Schleife zu identifizieren.

18. Computerprogrammprodukt nach Anspruch 17, bei dem der computerlesbare Code, der dazu eingerichtet ist, einen Computer zu veranlassen, die Schleife zu identifizieren, einen computerlesbaren Code umfasst, der dazu eingerichtet ist, einen Computer zu veranlassen, eine Rückverzweigung zu identifizieren.

19. Vorrichtung, die eine Klasse umfasst, die eine native Methode umfasst, wobei die native Methode durch einen nativen Code in einer Bibliothek (**320**) unterstützt wird, da
durch gekennzeichnet, daß
eine virtuelle Maschine (**105**) die Klasse verarbeitet, wobei die virtuelle Maschine (**105**) dazu eingerichtet ist, den nativen Code in einer übersetzten Form (**700**) auszuführen, und die virtuelle Maschine (**105**) folgendes umfasst:
einen Thread-Scheduler, der ein kooperatives Scheduling implementiert, und eine Übersetzungseinrichtung, die dazu eingerichtet ist, den nativen Code in eine Zwischenform und die Zwischenform in die übersetzte Form (**700**) zu übertragen, wobei die übersetzte Form (**700**) eine Form ist, die für kooperatives Scheduling aller Threads in der virtuellen Maschine (**105**) geeignet ist.

20. Vorrichtung nach Anspruch 19, bei dem die Übersetzungseinrichtung weiterhin dazu eingerichtet ist, ein Sperrsystemaufruf (**621A**) in der Zwischenform durch eine nichtsperrenden Variante (**621B**) des Systemaufrufs zu ersetzen.

21. Vorrichtung nach Anspruch 19, bei dem die Übersetzungseinrichtung weiterhin dazu eingerichtet ist, eine Speicherzugriffsüberprüfung bei einem Speicherzugriffsvorgang in der Zwischenform einzufügen.

22. Vorrichtung nach Anspruch 19, bei dem die Übersetzungseinrichtung weiterhin dazu eingerichtet ist, einen Yield-Vorgang an einem Yield-Punkt in der Zwischenform einzufügen.

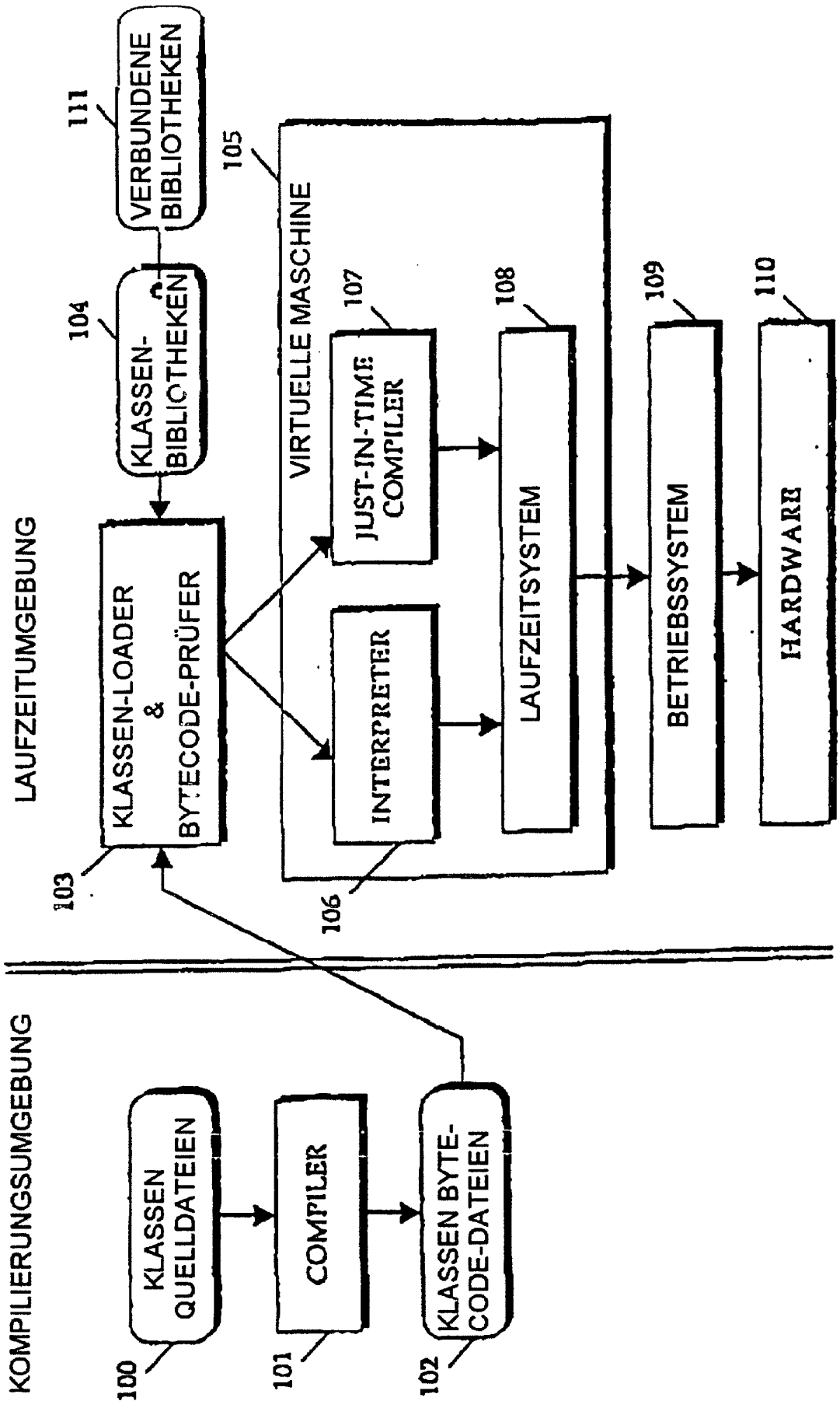
23. Vorrichtung nach Anspruch 22, bei welcher der Yield-Punkt ein Methodenaufruf ist.

24. Vorrichtung nach Anspruch 22, bei welcher der Yield-Punkt eine Schleife ist.

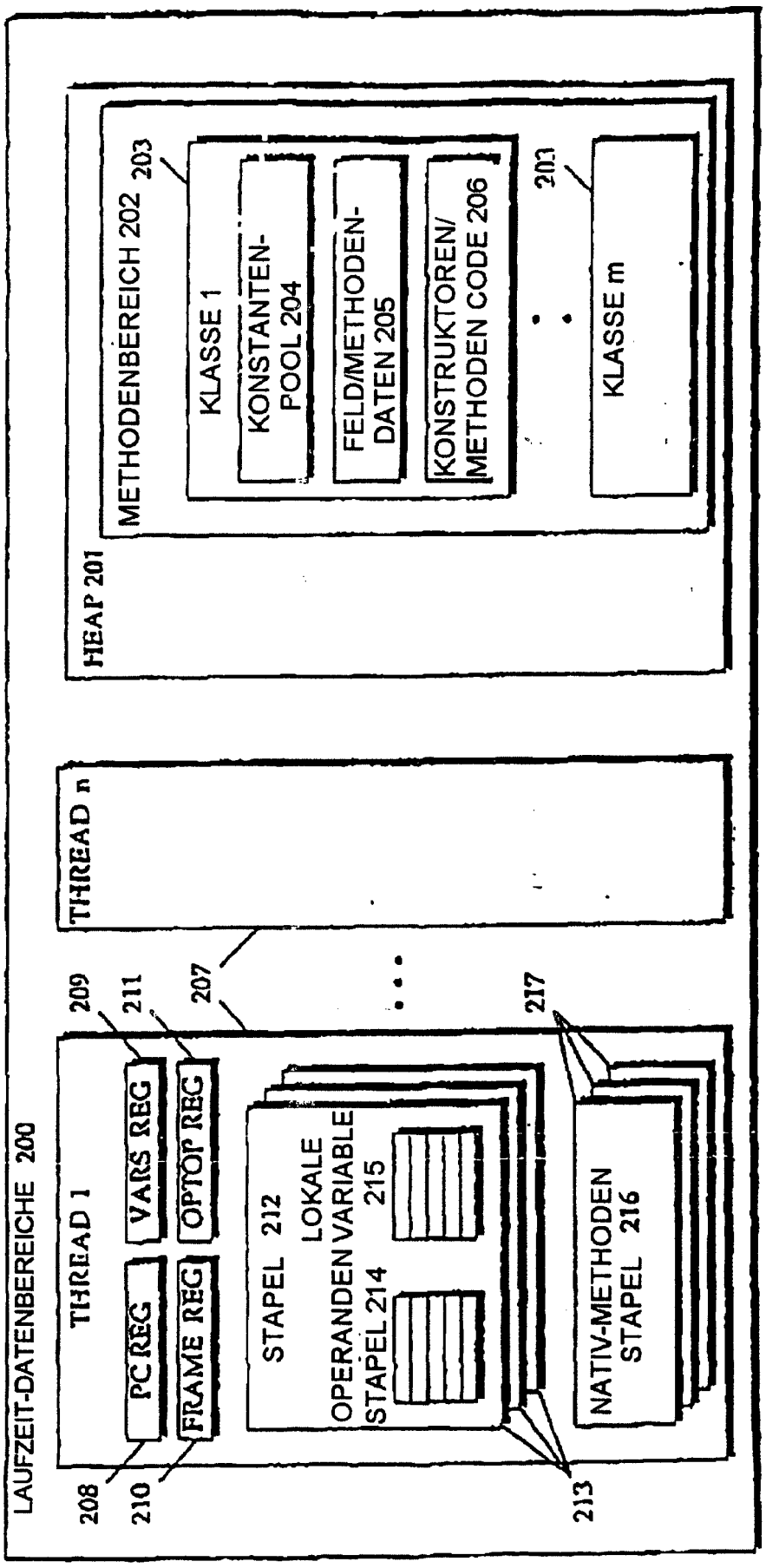
25. Vorrichtung nach Anspruch 24, bei der die Schleife durch eine Rückverzweigung identifiziert wird.

Es folgen 8 Blatt Zeichnungen

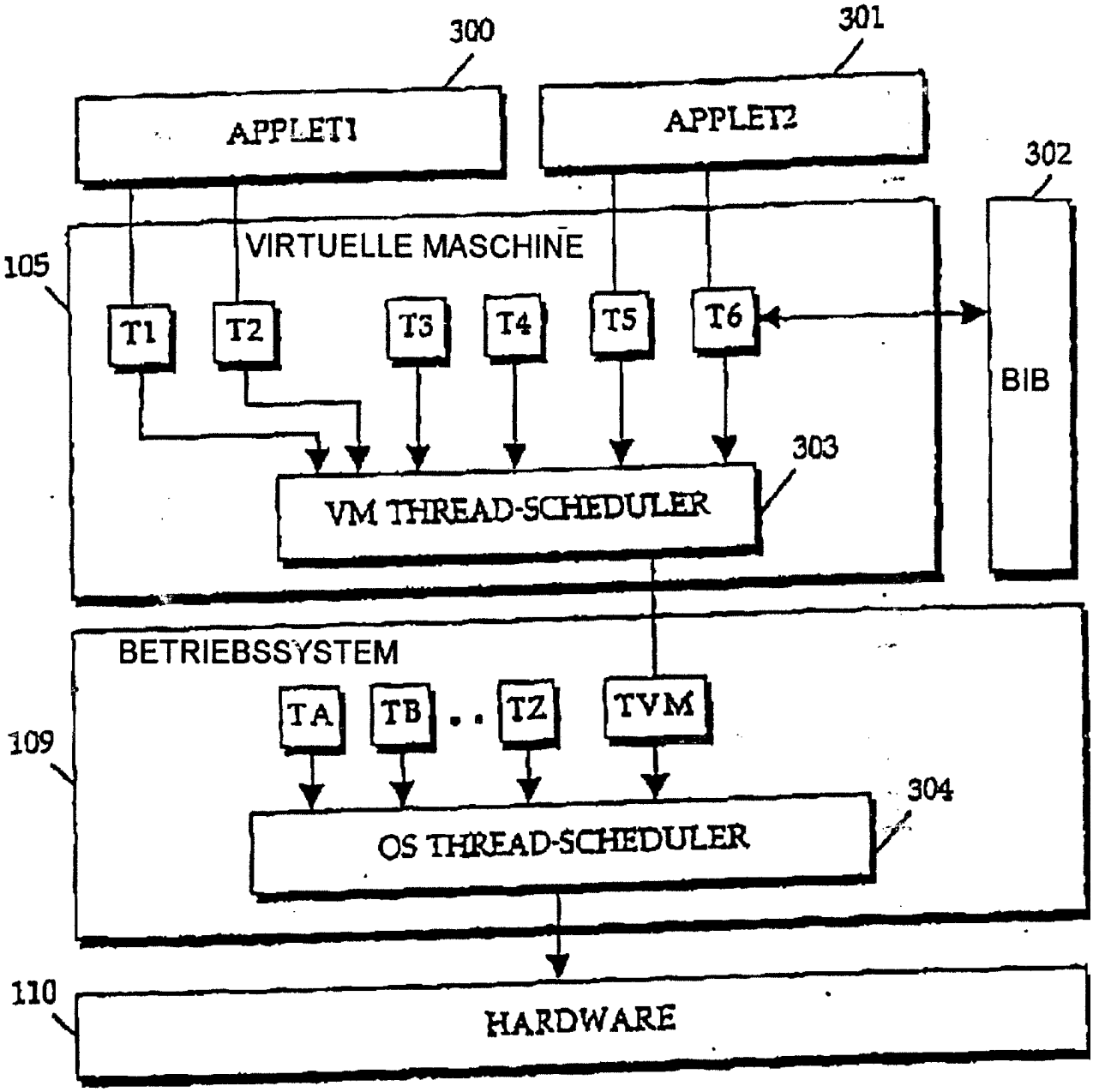
Anhängende Zeichnungen



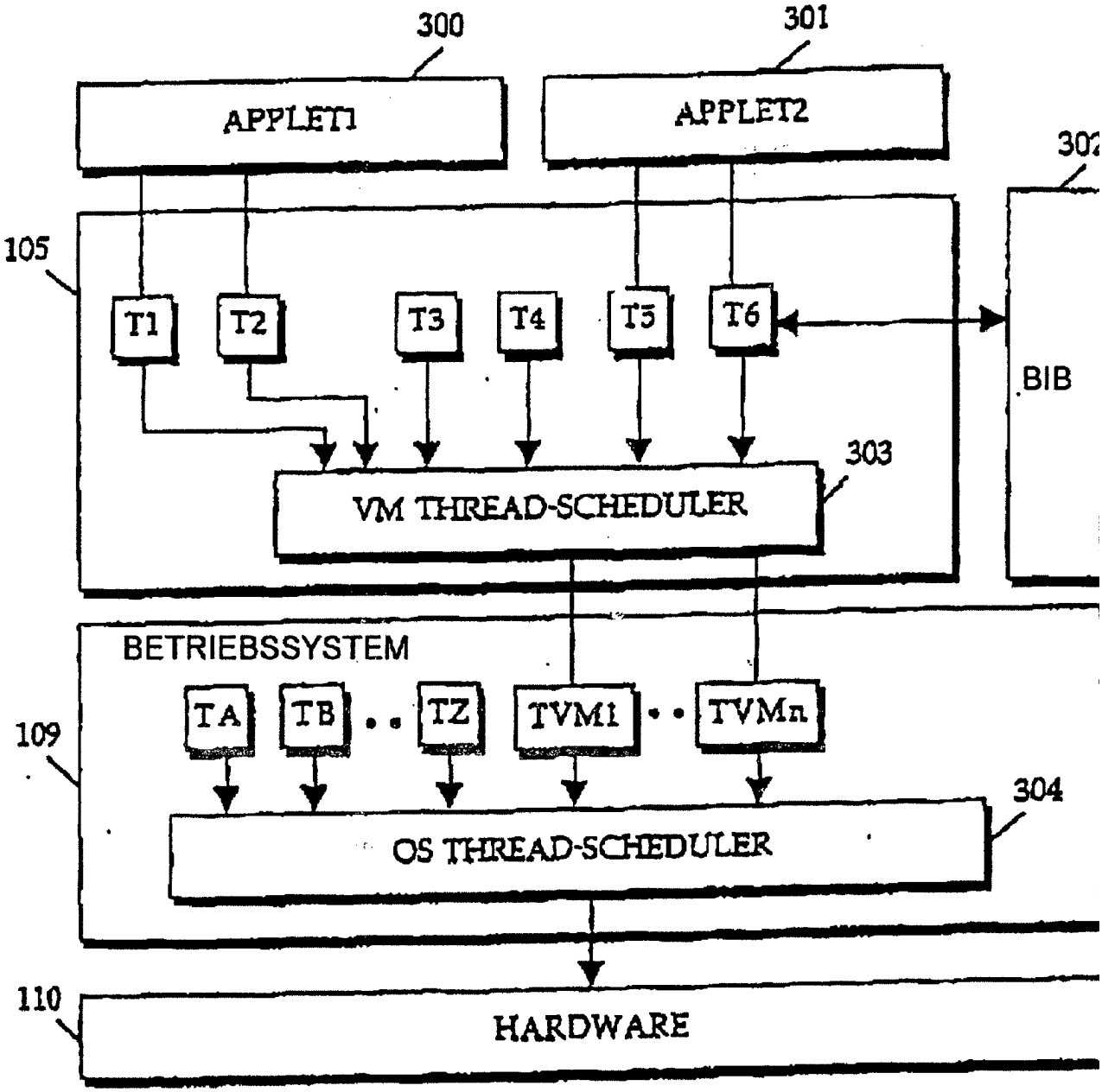
FIGUR 1


LAUFZEIT-DATENBEREICHE 200
THREAD 1
PC REG
FRAME REG
VARS REG
OPTOP REG
STAPEL 212
LOKALE VARIABLE 215
OPERANDEN STAPEL 214
NATIV-METHODEN STAPEL 216
THREAD n
HEAP 201
METHODENBEREICH 202
KLASSE 1
KONSTANTEN-POOL 204
FELD/METHODEN-DATEN 205
KONSTRUKTOREN/ METHODEN CODE 206
KLASSE m
203
208
210
209
211
207
213
217

FIGUR 2

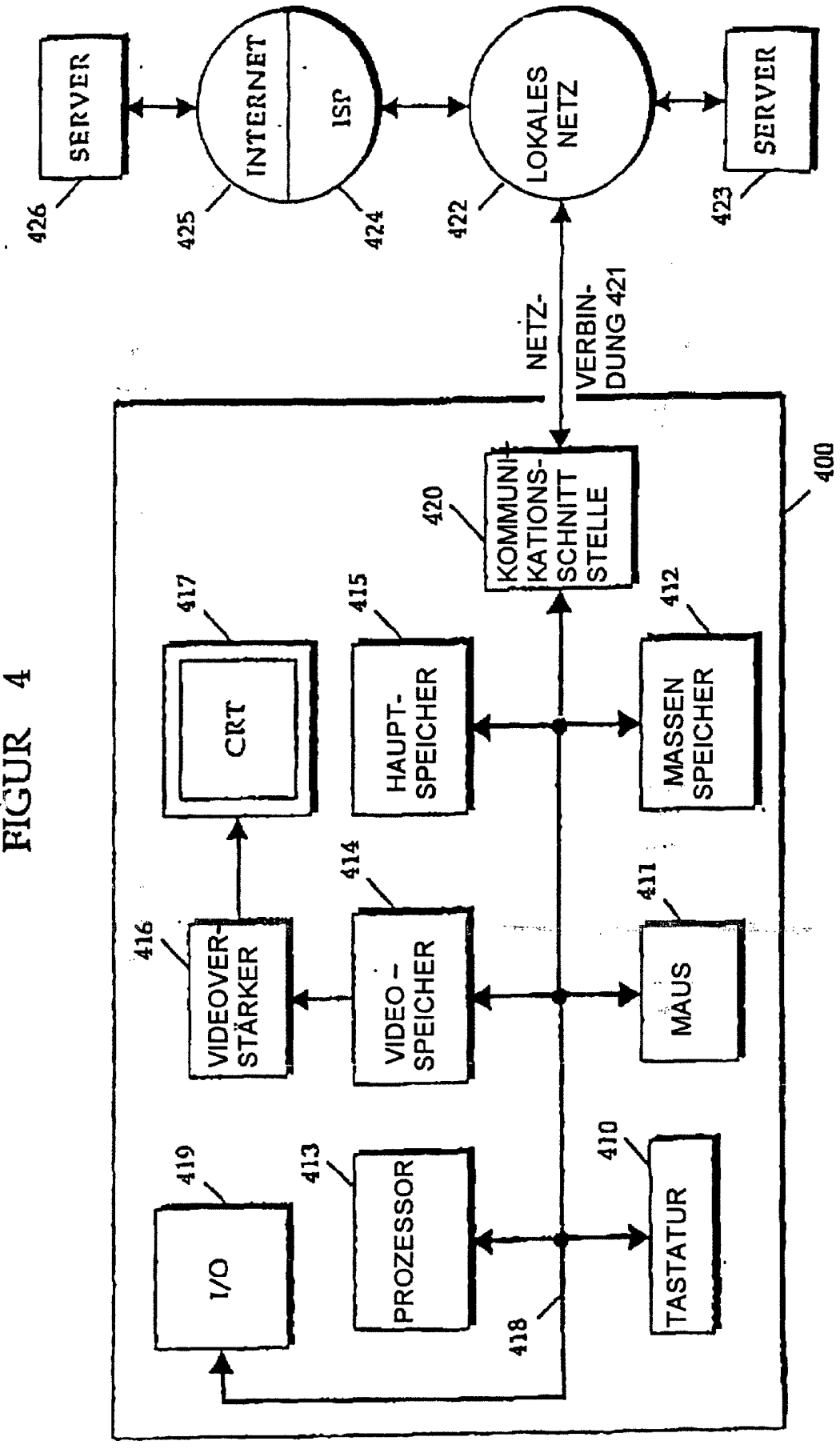

300
301
APPLET1
APPLET2
302
VIRTUELLE MASCHINE
105
T1
T2
T3
T4
T5
T6
BIB
303
VM THREAD-SCHEDULER
BETRIEBSSYSTEM
TA
TB
TZ
TVM
109
304
OS THREAD-SCHEDULER
110
HARDWARE

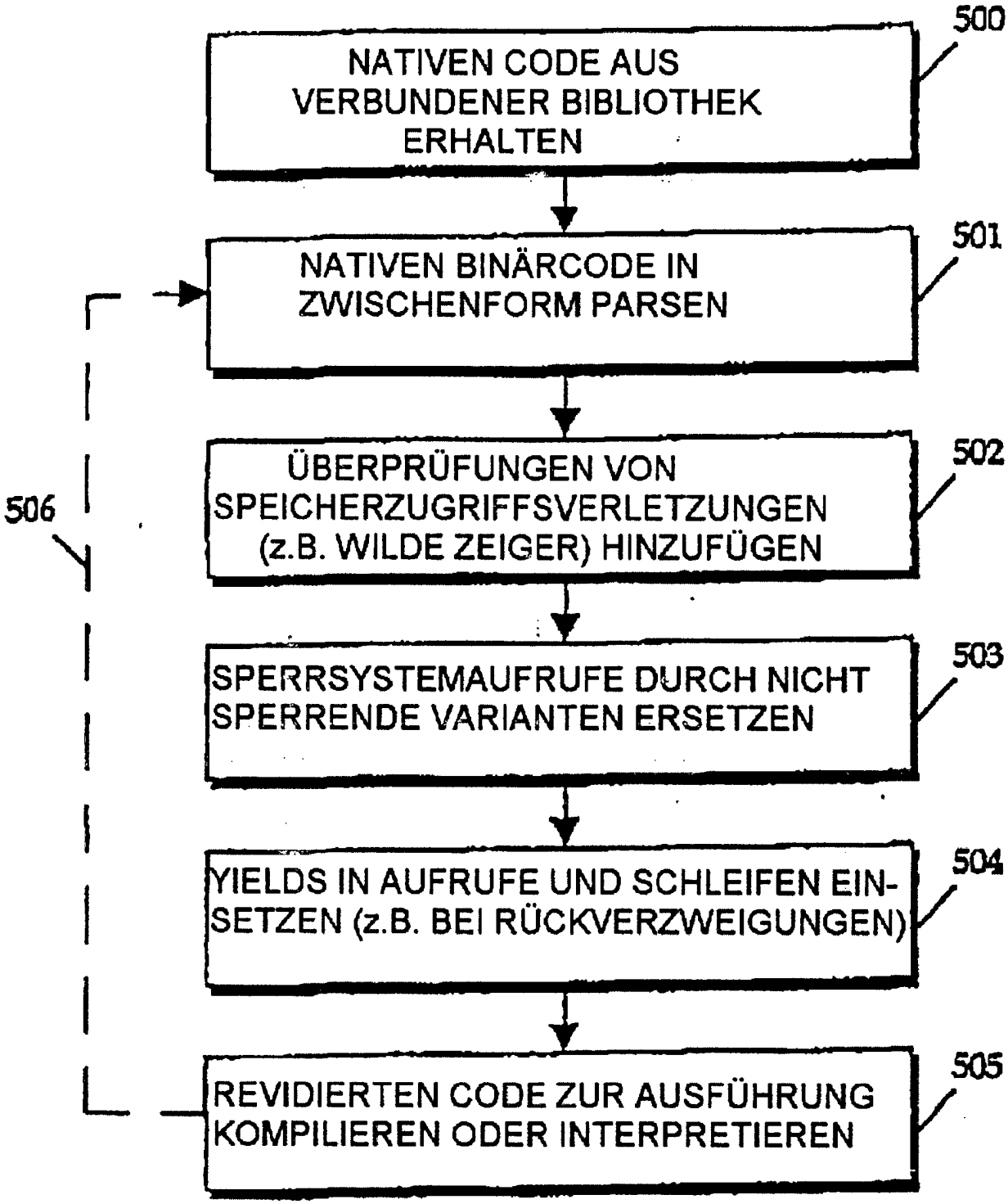
FIGUR 3A


300
301
APPLET1
APPLET2
105
T1
T2
T3
T4
T5
T6
BIB
303
VM THREAD-SCHEDULER
BETRIEBSSYSTEM
TA
TB
TZ
TVM1
TVMn
109
304
OS THREAD-SCHEDULER
110
HARDWARE

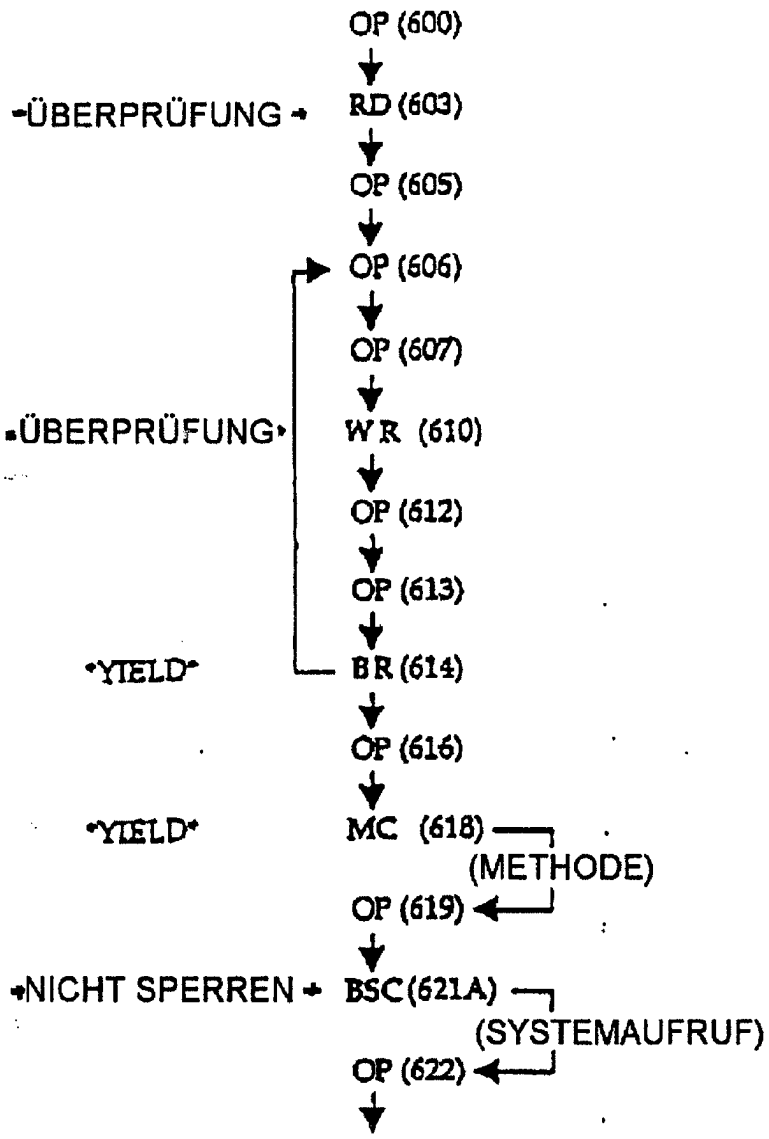
FIGUR 3B

FIGUR 4


SERVER
426
INTERNET
ISP
425
424
LOKALES NETZ
422
SERVER
423
NETZ-VERBIN-DUNG 421
400
KOMMUNI-KATIONS-SCHNITT-STELLE
420
CRT
417
HAUPT-SPEICHER
415
MASSEN SPEICHER
412
VIDEOVER-STÄRKER
416
VIDEO-SPEICHER
414
MAUS
411
I/O
419
PROZESSOR
413
TASTATUR
410
418

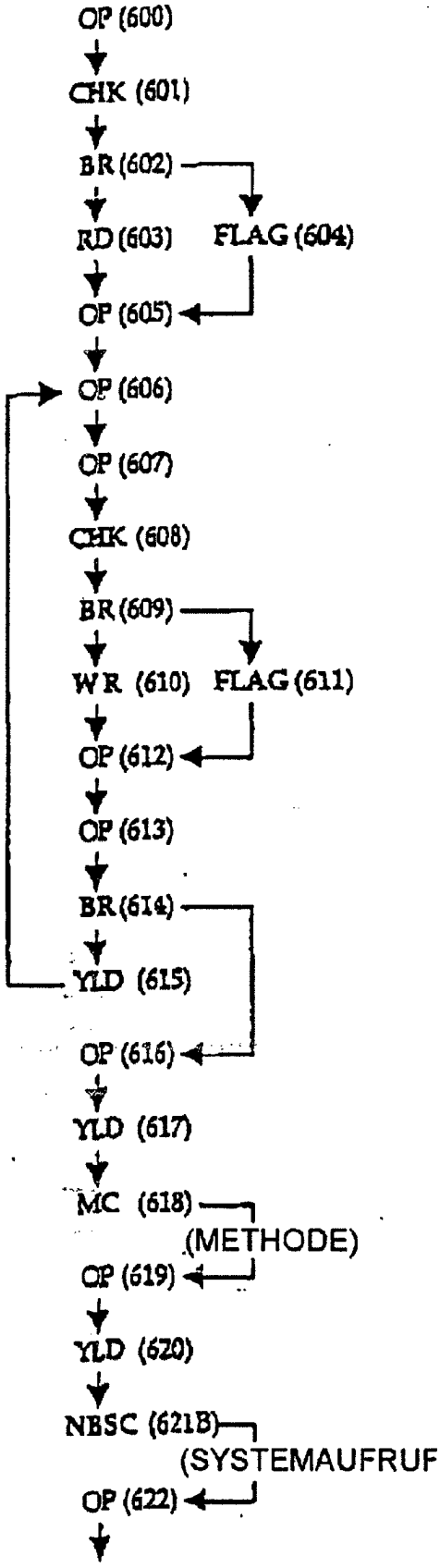

500
NATIVEN CODE AUS VERBUNDENER BIBLIOTHEK ERHALTEN
501
NATIVEN BINÄRCODE IN ZWISCHENFORM PARSEN
502
ÜBERPRÜFUNGEN VON SPEICHERZUGRIFFSVERLETZUNGEN (z.B. WILDE ZEIGER) HINZUFÜGEN
506
503
SPERRSYSTEMAUFRUFE DURCH NICHT SPERRENDE VARIANTEN ERSETZEN
504
YIELDS IN AUFRUFE UND SCHLEIFEN EINSETZEN (z.B. BEI RÜCKVERZWEIGUNGEN)
505
REVIDIERTEN CODE ZUR AUSFÜHRUNG KOMPILIEREN ODER INTERPRETIEREN

FIGUR 5

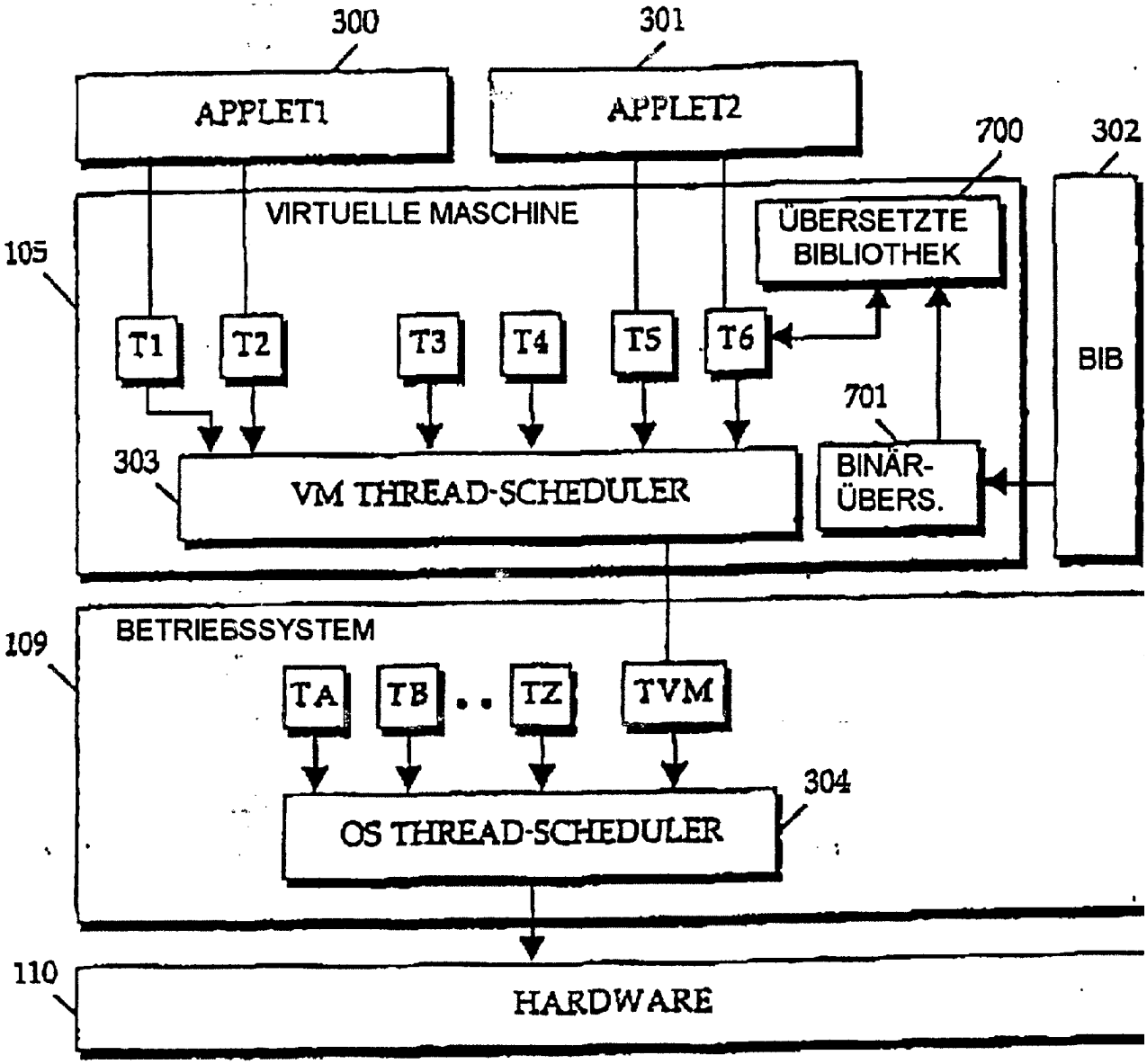


FIGUR 6A

LEGENDE: FIG: 6A-6B
RD = Speicherleseoperation
WR = Speicherschreiboperation
BR = Verzweigungsoperation (z.B. „if")
MC = Methoden-(Funktions-)Aufruf
BSC = Sperrsystemaufruf
OP = andere allgemeine Operation (sonstiges)
CHK = Zeigerüberprüfungsoperation
YLD = Yield-Operation
NBSC = nicht-sperrender Systemaufruf
FLAG = Signalzugriffsverletzung



FIGUR 6B


300
301
APPLET1
APPLET2
700
302
VIRTUELLE MASCHINE
ÜBERSETZTE BIBLIOTHEK
105
T1
T2
T3
T4
T5
T6
BIB
701
303
VM THREAD-SCHEDULER
BINÄR-ÜBERS.
BETRIEBSSYSTEM
109
TA
TB
TZ
TVM
304
OS THREAD-SCHEDULER
110
HARDWARE

FIGUR 7