Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

Virtuelle Machinen

- Abstraktion von Rechnern, Betriebssystemen und Software
- Virtuelle Maschinen und Virtualisierung sind zentraler Bestandteil im Betrieb moderner Rechneranlagen und Bestandteil von Betriebssystemen

Die Virtualisierung von Rechnerressourcen bringt in verschiedener Hinsicht neue Nutzen. Sie kann auf verschiedenen Stufen erfolgen und ist bereits teilweise in den Konzepten des Prozessmodells und des virtuellen Speichers realisiert.

Anwendungsbereiche

Eine Reihe von Virtualisierungsansätzen im Umfeld des Betriebssystems:

- Virtuelle Prozessoren: Simulation einer CPU-Hardware durch Software
- Virtuelle Prozessumgebungen: Nutzung virtueller Adressräume für Prozesse (Raummultiplex) zusammen mit CPU-Scheduling (Zeitmultiplex)
- Virtuelles Betriebssystem: Ausführung unveränderter (Binär-)Programme eines Betriebssystems A unter einem Betriebssystem B
- Virtueller Desktop: entfernter Zugriff auf einen grafischen Desktop über das Netzwerk
- Virtuelle Ressourcen: Sekundärspeicher- und Netzwerkvirtualisierung
- Sandboxing: virtuelle Prozesslaufzeitumgebungen inklusive Betriebssystem (Abstrakter Prozessor)
- Virtuelle Computer: Virtualisierung der gesamten Computerhardware mithilfe eines Virtual Machine Monitor (Hypervisor)

Taxonomie der Virtuellen Maschinen

Es können zwei wesentliche Klassen unterschieden werden:

Prozess Virtuelle Maschinen

Virtuelle Prozessoren oder Virtuelle Anwendungsmaschinen die eine Anwendung innerhalb oder außerhalb eines Betriebssystems ausführen und einen einzelnen Prozess unterstützen. Java Virtual Machines, auf denen in Java kompilierte Programme ausgeführt werden, sind Beispiele für Prozess-VMs. Es wird ein Prozessor simuliert.

System Virtuelle Maschinen

System-VMs ermöglichen die gemeinsame Nutzung der zugrunde liegenden physischen Maschinenressourcen durch verschiedene Virtual Machines, die jeweils ihr eigenes Betriebssystem ausführen. Es wird kein Prozessor simuliert, aber je nach Typ emuliert.

Simulation

Abtrakte Maschine ohne Kompatibilität zu bestehender Hardware

Emulation

Konkerete Maschinen mit bestmöglicherKompatibilität zu bestehender Hardware

Taxonomie der Virtuellen Maschinen

[VM,Smith,2005]

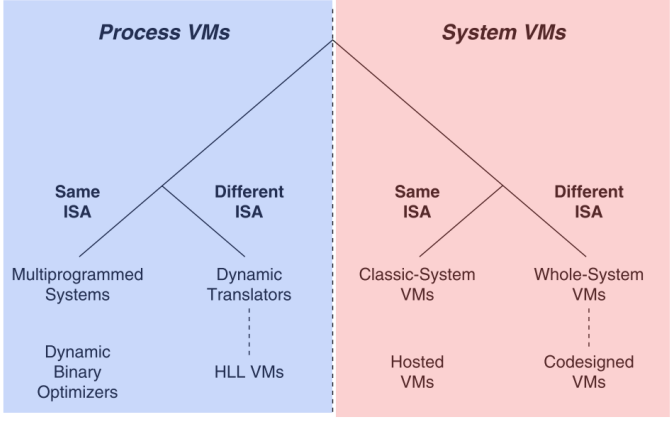


Abb. 1. Übersicht und Klassendiagramm der VM Architekturen und Methodem

Virtualisierung

[VM,Smith,2005]

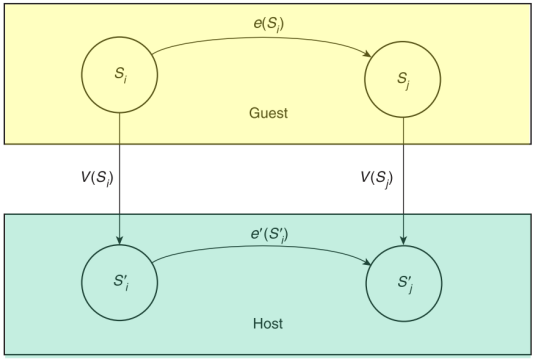


Abb. 2. Virtualisierung. Formal ist Virtualisierung die Konstruktion eines Isomorphismus zwischen einem Gastsystem und einem Host; e' ◦ V(S_i) = V ◦ e(S_i). Dabei ist e eine Funktion im Gastsystem, die den Zustand S_i in S_j überführt. Die Virtualisierung V "übersetzt" den Zustandsübergang in das Hostsystem. Im Host gibt es eine entsprechende Abfolge von Operationen e', die eine äquivalente Änderung des Hostzustands vornimmt (Änderungen S'_i zu S'_j).

Virtualisierung

[VM,Smith,2005]

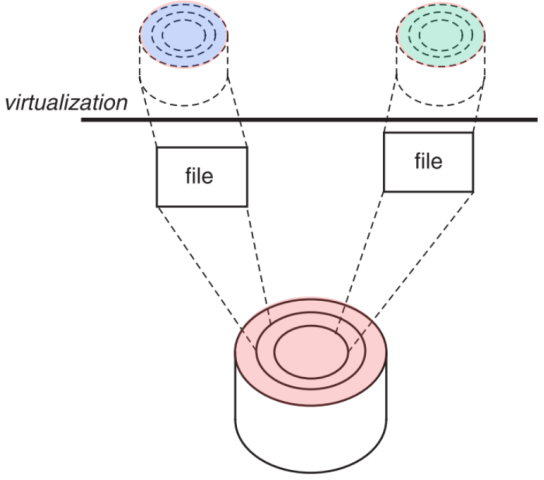


Abb. 3. Implementierung virtueller Festplatten. Virtualisierung bietet eine andere Schnittstelle und / oder Ressourcen auf derselben Abstraktionsebene. Das Dateisystem ist Virtualisierung!

Sandboxing (virtuelles Laufzeitsystem)

- Die Laufzeitumgebung einer Applikation wird vom restlichen System abgeschirmt, d.h. virtualisiert (Container-based Operating Systems, COS).
  - Diese Möglichkeit wird von einigen Betriebssystemen unterstützt, zum Beispiel von Oracle Solaris mit den Containers, vom FreeBSD-Unix mit den Jails und von Linux mit dem Vserver.
  - Dasselbe Konzept wird aber beispielsweise auch in Webbrowsern im Kleinen genutzt, um einzelne Webseiten oder Plug-ins zu isolieren.
  - Beim Sandboxing sieht jeder Prozess nur sich selbst und das Betriebssystem, wodurch eine wechselseitige Beeinflussung von Applikationen untereinander verhindert wird.
  - Dazu gehört, dass jedem Anwenderprozess nur ein begrenzter Anteil an der Nutzung gemeinsamer Ressourcen gewährt wird.

Jede Sandbox ist ein separater User Space, der sich mit anderen Sandboxen lediglich den geschützten Kernel Space teilt.

Die im Unix-Bereich klassische Lösung der chroot-Umgebung kann nicht dieselbe Isolation gewähren wie das Sandboxing-Konzept, da die damit realisierbare Isolierung unvollständig ist.

Virtual Hosting

Anwendung findet das Sandboxing bei Anbietern von Virtual Hosting, zur erhöhten Isolierung kritischer Applikationen bei sich wechselseitig nicht trauenden Systembenutzern, zur feineren Kontrolle der Ressourcenzuteilung und zur Ermöglichung einfacher Migrierungen von Applikationen.

Im Vergleich zu Lösungen mit einem Virtual Machine Monitor ist der Zusatzaufwand gering, da stets dieselbe Systemprogrammierschnittstelle genutzt wird und spezielle Hardware unnötig ist. Allerdings wird nicht dieselbe Flexibilität erreicht, da nur Applikationen für dieselbe Betriebssystemplattform lauffähig sind.

Virtual Private Server

Ein Schritt weiter: Ein ganzer Rechner wird aufgeteilt und wird mehreren Betriebssystemen und Betreibern (Kunden) zur Verfügung gestellt.

Beispiel: https://edu-9.de ist Virtual Private Server Hosting!

- Auf einem virtuellen privaten Server wird eine eigene Kopie eines Betriebssystems (OS) ausgeführt, mit möglicherweise Zugriff auf die Superuser-Ebene dieses Betriebssysteminstanz, sodass sie nahezu jede Software installieren können, die auf diesem Betriebssystem ausgeführt wird.

- Für viele Zwecke entspricht es funktional einem **dedizierten physischen Server** und kann softwaredefiniert einfacher erstellt und konfiguriert werden (z.B. webbasierte Installationd eines Betriebssystems).

Container-Systeme

- Für den betrieblichen Einsatz ist eine möglichst einfache Bereitstellung von Anwendungen wichtig, die voneinander isoliert ablaufen und wenig Systemabhängigkeiten zeigen (Bibliotheken!).

  - Dies ermöglichen Container-Systeme, wie z.B. Docker oder rkt, die beide auf Linux aufsetzen.
  - Dazu werden Anwendungen, wie z.B. ein Webserver oder ein Datenbanksystem, mithilfe entsprechender Werkzeuge als Standardpakete bereitgestellt.
  - Werden diese Pakete, als Abbilder (Images) bezeichnet, auf einer webbasierten Plattform allgemein nutzbar bereitgestellt, so benötigt eine Installation nur noch wenige Arbeitsschritte.
  - Jedes Abbild, als einzelne Datei verfügbar, enthält nicht nur die Anwendung selbst, sondern auch alle Drittsoftware, die bei traditioneller Bereitstellung zusätzlich zu installieren wäre.
  - Für Docker-Images ist dies mit dem Docker Hub realisiert, der als Onlinedienst sowohl eine öffentliche wie auch eine private Bereitstellung erlaubt.

System-VM

1. Emulation von Prozessoren unter Verwendung physischer Prozessoren;
2. Emulation von Hardwaregeräten unter Verwendung von Betriebssystemmitteln;
3. Virtualisierung von Speicher unter Verwendung von Betriebssystemmitteln.

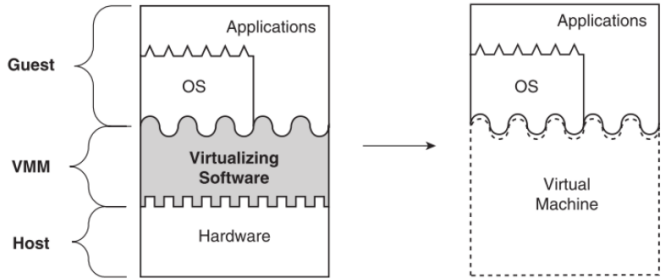


Abb. 4. Eine virtuelle Systemmaschine. Virtualisierungssoftware übersetzt die von einer Hardwareplattform verwendete ISA in eine andere und bildet eine virtuelle Systemmaschine, die in der Lage ist, eine Systemsoftwareumgebung auszuführen, die u.U. auch für eine andere Hardware entwickelt wurde.

System-VM

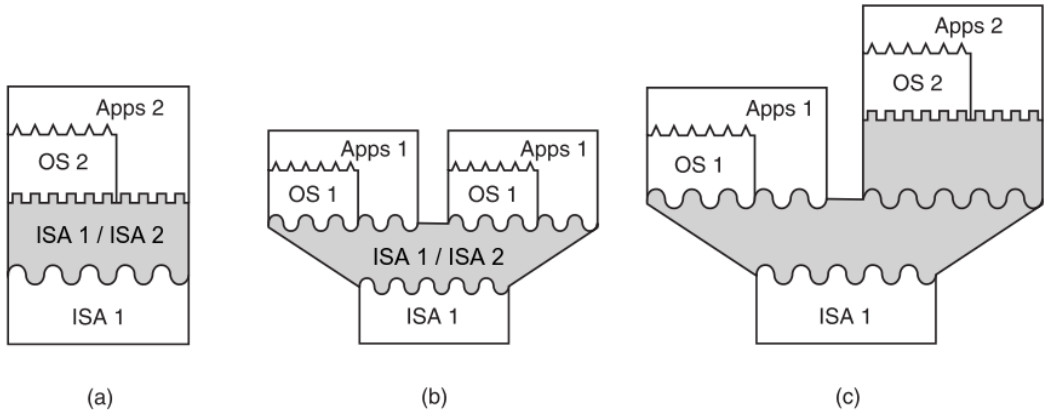


Abb. 5. Beispiele für Anwendungen virtueller Maschinen. (a) Emulieren eines Befehlssatzes mit einem anderen; (b) Replizieren einer virtuellen Maschine, so dass mehrere Betriebssysteme gleichzeitig unterstützt werden können; (c) Komponieren von Software für virtuelle Maschinen zu einem komplexeren, flexibleren System.

System-VM

Virtual Machine Monitor bzw. Hypervisor

Neben der Speichervirtualisierung ist heute die Bereitstellung virtueller Rechnerumgebungen durch einen **Virtual Machine Monitor** (VMM) bzw. Hypervisor die wichtigste Virtualisierungsform.

- Der VMM erlaubt die Erstellung mehrerer virtueller Umgebungen (hier dann VM genannt) in denen sich unterschiedliche Betriebssysteme installieren lassen.

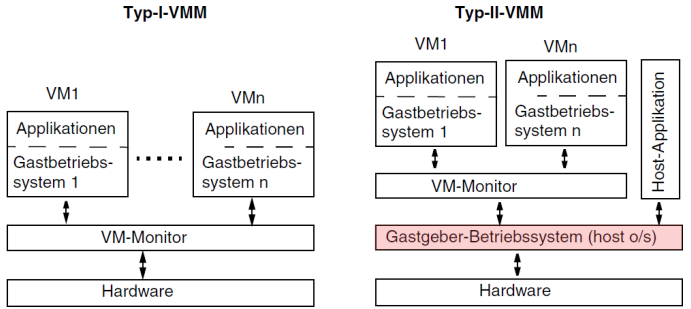


Abb. 6. VMM-Typen: Ohne (bare metal) und mit zwischengeschalteten Betriebssystem

Ein VMM kann auf zwei Arten realisiert werden, die als Typ-I- und Typ-II-VMM bezeichnet werden.

1. Beim Typ I setzt der VMM direkt auf der Hardware auf und virtualisiert diese (bare-metal). Der VMM stellt ein eigenes Betriebssystem dar, dessen Aufgabe in der Ressourcenverwaltung und der Bereitstellung von VMs liegt. Die Ressourcenverwaltung umfasst die Aufteilung der Rechenzeit (scheduling) und die Zuteilung (allocation) von Speicher und Peripherie auf die einzelnen VMs. Beispiel eines solchen Produkts ist VMware ESX, das aber nur auf Hardware ablaufen kann, für die es eigene Treiber besitzt.
2. Ein Typ-II-VMM läuft als eine von mehreren Applikationen (hosted) auf einem Gastgeber-Betriebssystem (host operating system). Damit er jedoch die nötigen Rechte auf der Hardware erhält, benutzt er einen speziellen VMM-Treiber, der unter dem Gastgeber-Betriebssystem installiert ist und dadurch dem Hypervisor Kernmodus-Privilegien eröffnet. Jedes weitere Betriebssystem, das unter dem VMM läuft, wird als Gastbetriebssystem (guest operating system) bezeichnet. Der VMM kann auf Mechanismen des unterliegenden Gastgeber-Betriebssystems für die Ressourcenverwaltung zurückgreifen. Typ-II-Produkte sind VMware Workstation, Virtual-Box und Xen.
3. Eine zwischen Typ I und Typ II anzusiedelnde Variante sind Hypervisors, die in das Betriebssystem von Haus aus bereits integriert sind. Realisiert wird dies in Linux durch die KVM (Kernel-based Virtual Machine) und unter Windows mit Hyper-V. Diese Zwischenform lässt sich so einsetzen, dass keine Applikationen auf dem Betriebssystem selbst laufen, sondern nur der Hypervisor genutzt wird. In diesem Szenario sind keine Leistungsnachteile im Vergleich zu reinen Typ-I-VMM zu erwarten.

System-VM

Anforderungen

Ein VMM muss nach Popek/Goldberg (1974) drei Kerneigenschaften unterstützen:

1. Ausführungsumgebung (fidelity, equivalence): Programme laufen auf einem virtualisierten Rechner identisch ab wie auf einem realen Rechner, mit Ausnahme der Geschwindigkeit.
2. Effizienz (performance, efficiency): Die große Mehrheit aller Maschinenbefehle, die innerhalb einer VM abgearbeitet werden, müssen direkt durch den realen Prozessor ausgeführt werden ohne die Intervention durch den VMM.
3. Ressourcenverwaltung (safety, resource control): Die Ressourcen werden vollständig durch den VMM verwaltet und den einzelnen VMs zugeteilt bzw. wieder entzogen.

System-VM

Funktionsweise eines VMM

In der Anfangszeit der VMM in der PC-Welt fehlte eine passende Hardwareunterstützung, sodass sich der VMM und das in der VM ablaufende Betriebssystem den Kernmodus teilten.

Ohne Unterstützung durch Hardware

Ein sicherer VM-Monitor setzt eine voll virtualisierbare CPU voraus. Das heißt, dass alle Instruktionen, die auf die Hardware zugreifen oder indirekt die Hardware beeinflussen können, nur privilegiert ausgeführt werden. Dazu muss die CPU zwischen einem privilegierten und nicht privilegierten Betriebsmodus unterscheiden. Typischerweise werden diese als Kernmodus (kernel/supervisor mode) und Benutzermodus (user mode) bezeichnet. Prozessorintern wird der privilegierte Modus durch ein gesetztes KM-Bit repräsentiert (KM für Kernel Mode). Dieses Bit kann im Benutzermodus natürlich nicht direkt gesetzt werden. Erfolgt jedoch ein Hardware- oder Software-Interrupt, so wird es automatisch durch die CPU selbst gesetzt.

Funktionsweise eines VMM

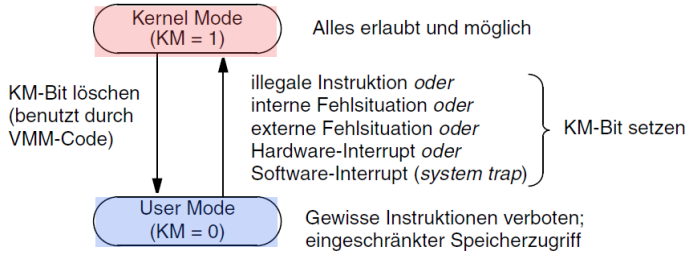


Abb. 7. Zustandsmodell für Privilegiensystem der CPU

Funktionsweise eines VMM

Was passiert nun, wenn ein Programm im Benutzermodus eine privilegierte Instruktion ausführen will? Die CPU wird in diesem Fall den Befehl nicht ausführen, sondern einen Software-Interrupt (trap) auslösen. Wird ein VMM eingesetzt, so läuft nur der VMM im privilegierten Modus. Aller restlicher Code, einschließlich irgendwelcher Betriebssysteme, läuft im Benutzermodus. Die Unterscheidung eines Betriebssystems zwischen Benutzer- und Kernmodus wird nämlich durch den VMM ebenfalls virtualisiert.

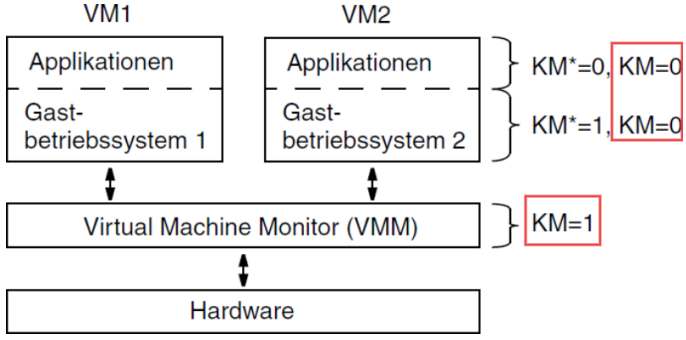


Abb. 8. Sichere Virtualisierung mittels Privilegiensystem: KM*: virtualisiertes Privilegien-Bit (pro VM)

Funktionsweise eines VMM

Mit Unterstützung durch Hardware

Moderne CPUs enthalten eine Zusatzhardware für eine einfachere Realisierung des VMM, der dann als Hardware-assisted VMM bezeichnet wird. Dies trifft für viele Prozessortypen zu, auch für die x86- und x86-64-Prozessoren. Ergänzend zu den Privilegienstufen, die das Betriebssystem nutzt, steht eine nochmals höher privilegierte Hypervisor-Stufe zur Verfügung, in die bei der Ausführung kritischer Instruktionen verzweigt wird. Bei x86-Prozessoren wird diese als Root-Betriebsmodus bezeichnet.

System-VM

Einsatz von VMM für ...

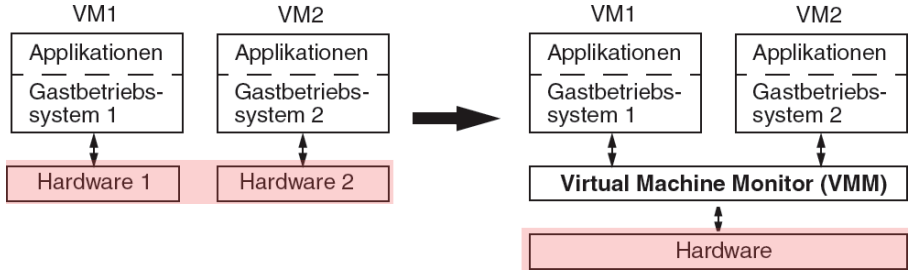


Abb. 9. VMM für die Serverzusammenführung (Konsolidierung) auf einer Hardware

System-VM

Einsatz von VMM für ...

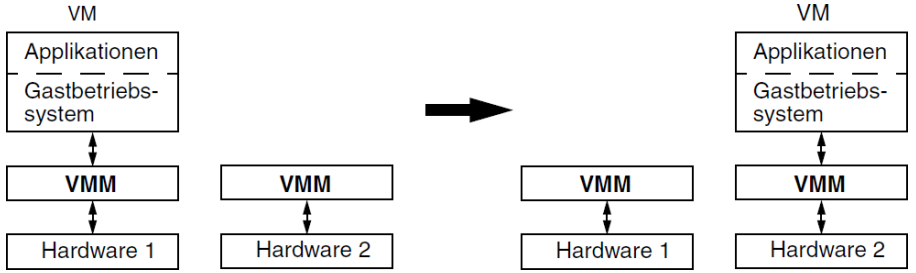


Abb. 10. VMM-Einsatzszenario Applikationsmigration

System-VM

Einsatz von VMM für ...

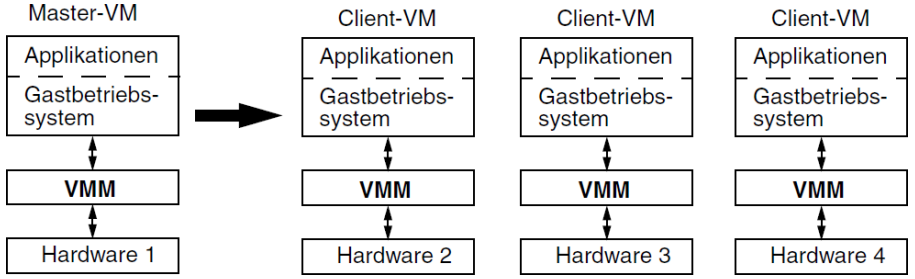


Abb. 11. VMM-Einsatzszenario Softwaredistribution (Beispiel) mit Replikation

Prozess-VM

Beispiele für Hochsprachen Prozess-VMs:

1. Java Virtual Machine (Bytecode, Stack- und Registerprozessor)
2. Lua (Lua, LuaJit, Bytecode)
3. JavaScript (Google V8 - Bytecode und nativer Maschinenkode, Spidermonkey - Bytecode, WebAssembly)
4. Continuum (JavaScript -> JavaScipt)

Kompiler werden eine Programmiersprache in ein i.A. lineares Maschinenprogramm übersetzen. Dabei wird die Komplexität reduziert.

Für die zuvor beschriebenen Prozess-VMs ist die plattformübergreifende Portabilität eindeutig ein sehr wichtiges Ziel. Vollständige plattformübergreifende Portabilität lässt sich leichter erreichen, wenn die Prozess-VM in ein gesamtes Software-Framework integriert wird.

Prozess-VM

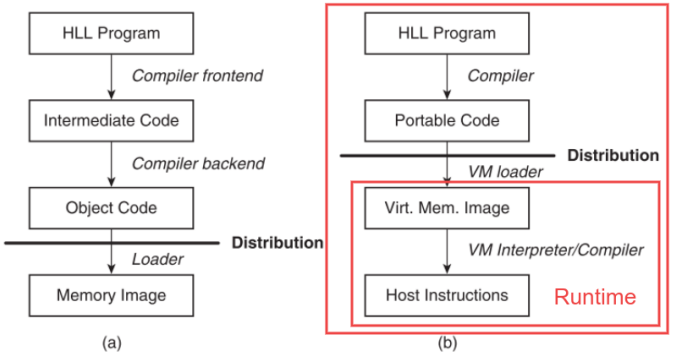


Abb. 12. Hochsprachenumgebungen. (a) Ein herkömmliches System, in dem plattformabhängiger Objektcode erzeugt, verteilt und ausfgeführt wird → starke Betriebssystem- und Rechnerabhängigkeit; (b) eine HLL-VM-Umgebung, in der portabler Zwischencode von einer plattformabhängigen virtuellen Maschine "ausgeführt" wird → schwache Betriebssystem- und Rechnerabhängigkeit.

Prozess-VM

[VM,Smith,2005]

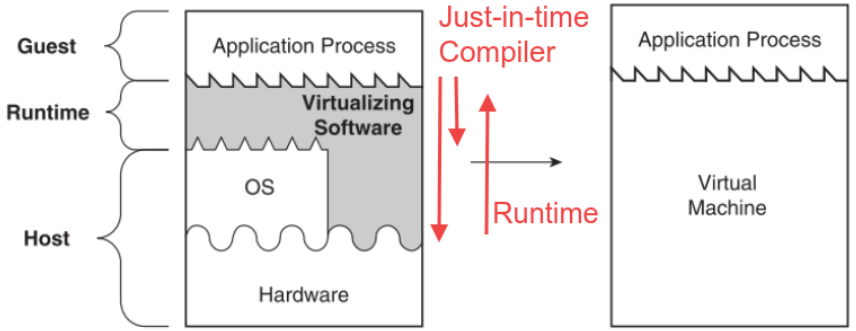


Abb. 13. Eine virtuelle Prozessmaschine ist Virtualisierungssoftware die einen Satz von Anweisungen auf Betriebssystem- und Benutzerebene übersetzt, und eine Plattform auf eine andere abbildet - quasi ein Kompiler mit laufzeitumgebung (runtime).

Prozess-VM

Klassen der Hochsprachen Prozess-VMs:

1. Interpreter ("eat and think", Programmkode wird sofort ausgeführt)
   - Basic
   - Bash Shell

```
interpret(method) {
  while( code remains in sequence ) {
    read the next code from the sequence;
    if (the code needs more data){
      read more data from the sequence;
    }
    perform actions specified by the code;
  }
}
```

Alg. 1. Der Interpreter

Prozess-VM

- Die Komplexität verbirgt sich im Schritt “Aktionen ausführen, die durch den Code definiert sind."
  - Wenn der Code beispielsweise eine neue Instanz einer Klasse erstellen soll, ruft der Interpreter den Garbage Collector (Speichermanager der VM) auf, um einen Speicherabschnitt zuzuweisen, den Speicherinhalt auf Null zu setzen, den Objektheader zu initialisieren (z.B. einen vtable-Zeiger der Klasse zu installieren) und dann gibt er den Objektzeiger zurück.
  - Wenn der Code eine virtuelle Methode aufrufen soll, muss der Interpreter die Methodenadresse herausfinden, einen Stapelrahmen vorbereiten, die Argumente auf den Stapel laden, die Methode aufrufen, was rekursives interpretieren bedeuten kann, und das Ergebnis zurückgeben.
  - Beim Aufruf einer Zielmethode/Funktion muss u.U. der Methodencode geladen und analysiert werden, wenn er sich noch nicht im Speicher befindet oder noch nicht initialisiert wurde.
  - Verzweigungen im Programmfluss (Sprünge) müssen ggfs. im Programmkode gesucht werden.

Prozess-VM

2. Besser: Bytecode oder Super Code Maschinen (abstrakte Software ISA)

```
int x; x=0; x=x+1; printf("%d",x);
if (x<0) { x=-x; } else { x=1; };
-----------------------------------------------
ALLOCATE(4) VARIABLE(x) NUMBER(0) WRITE
VARIABLE(x) VARIABLE(x) READ NUMBER(1) ADD WRITE
STRING("%d") VARIABLE(x) READ CALL(printf)
VARIABLE(x) NUMBER(0) LT BRANCHZ(5)
VARIABLE(x) VARIABLE(x) READ NEG WRITE BRANCH(3)
VARIABLE(x) NUMBER(1) WRITE
```

Bsp. 1. Beispiel Überstzung C Programm in lineare Instruktionssequenz für abstrakte Stackmaschine

Abstrakte Maschine

Register Maschine

- Die meisten Instruktionen werden mit einem Registersatz und direkten Speicherzugriff ausgeführt
- Einen Stack gibt es meist nur für temporäre Varibalen
- Das Instruktionsformat kann keinen, einen, zwei oder mehr Operanden enthalten (Konstante, Register, Speicheradresse)

Stack Maschine

- Die meisten Instruktionen werden über einen oder mehreren Stapelspeichern und einem Akkumulator ausgeführt
- Eventuell kleiner Registersatz für temporäre Ergebnisse
- Das Instruktionsformat enthält i.A. keine Operanden, diese befinden sich auf dem Stack, und auch Ergebnisse von Berechnungen werden wieder auf dem Stack abgelegt,.

Prozess-VM Interpreter Loop

- Die Ausführung von abstrakten Instruktionen findet vollständig in Software statt, was langsam ist.
- Es gibt i.A. einen Schleife die aus einem Programmcodespeicher Instruktionen liest, diese kodieren und schließelich ausführen muss.

```
CS[];
ip=0;
while(!terminated) {
  code=CS[ip]
  instr=decode(cmd)
  status=execute(instr)
  ip=next(instr,status)
}
```

Alg. 2. Die Main Loop

Instruktionsformat

- Neben "binären" Bytecode (man spricht von seriellisierten komopakten Code) kann man auch nicht kodierte Instruktionsformate wählen die keine Dekodierung benötigen.

```
struct command {
  opcode   : opcodes[OPNAME],
  operands : operand [],
};
enum opcodes {
  OP1 = 1,
  OP2 = 2,
  ...
}
union operand {
  number_t constant;
  address_t address;
  address_t register;
}
```

Def. 1. Instruktionsformat

Instruktionsdekoder

- Es muss eine spezifische opCode Funktion ausgeführt werden (Mikrooperationen). Wir gehen davon aus dass der opCode bereits dekodiert ist.

Wie bildet man einen (numerischen oder symbolischen) opCode auf eine Funktion **effizient** ab?

1. Eine lange *if-then-else-if* Kaskadenstruktur die den opCode mit Anweisungen verbindet und diese direkt ausführt ⇒ nicht konstante Laufzeit!
2. Eine *switch-case* Mehrfachauswahl. Zunächst kein große Verbesserung zu 1. da auf Maxhinenebene ein *switch-case* in eine Vielzahl von Vergleichsanweisungen mit bedingten und unbedingten Sprüngen umgesetzt wird (wie *if-then-else-if*)
3. Irgeneine Form eine Lookup Tabelle deren Index der opCode ist und der Tabelleneintgrag entweder eine Funktions- und interne Codesprungadresse ausgibt.
   - Hier müssen die opCodes (bestenfalls) aufeinander folgend nummeriert werden.
   - Hier gibt es konstante Dekodierungslaufzeit!

Prozess-VM Interpreter Loop (Option 2)

```
while() {
  op=CS[ip]
  if (op==OP1) { status=doop1(arguments); }
  else if (op==OP1) { status=doop1(arguments); }
  else if (op==OP2) { status=doop2(arguments); }
  ...
  else if (op==OPN) { status=doopN(arguments); }
  ip=next(ip,op,status);
}
```

Alg. 3. Nicht effiziente Implementierung einer Instruktionsschleife in C mit linearer Ausführungszeit ohnen Möglichkeit der Optimierung durch Compiler - Option 1

Prozess-VM Interpreter Loop (Option 2)

```
while() {
  op=CS[ip]
  switch (op.code) {
    case OP1: status=doop1(arguments); break;
    case OP2: status=doop2(arguments); break;
    ...
    case OPN: status=doop3(arguments); brak;
  }
  ip=next(ip,op,status);
}
```

Alg. 4. Effiziente Implementierung einer Instruktionsschleife in C mit linearer Ausführungszeit (Mulitplexer), aber mit Möglichkeit für den Compiler mittels Tabellen in konstanter Laufzeit zu realisieren - Option 2

Prozess-VM Interpreter Loop (Option 3)

```
enum opCode {
  OP1,
  OP2,
  ...
  MAXOP
};
void op1(..); ...
OPTABLE[MAXOP]={op1,op2,..}
...
while() {
  next=OPTABLE[op.code]
  next()
}
```

Alg. 5. Effiziente Implementierung einer Instruktionsschleife in C mit konstanter (?) Ausführungszeit (LUT) - Option 3

JIT Übersetzung

Die Ausführung von Maschineninstruktionen in Software führt zu geringer Verarbeitungsgeschwindigkeit. Typisch werden 100 native Maschineninstruktionen für die Ausführung einer abstrakte Instruktion benötigt.

- Die JIT-Kompilierung kompiliert einen Teil des Anwendungscodes zur Laufzeit in binären (nativen) Maschinencode und ermöglicht es der VM dann, den generierten Code direkt auszuführen, anstatt den ursprünglichen Teil des Anwendungscodes zu interpretieren.
- Es ist, als würde man den gesamten Anwendungscode als eine einzige Superanweisung behandeln.
- Man unterscheidet:

1. Übersetzung von Programmkode
2. Übersetzung von Daten(strukturen)

Die erste Frage an JIT ist, wie der zu kompilierende Anwendungscode ausgewählt wird?

JIT Übersetzung

- Eine Methode oder Funktion wird aufgrund ihrer genau definierten semantischen Grenze als Kompilierungseinheit betrachtet.
- Bei Daten wird es schwieriger die semantische Grenze zu finden!

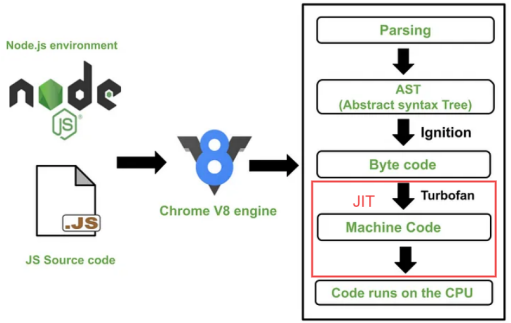
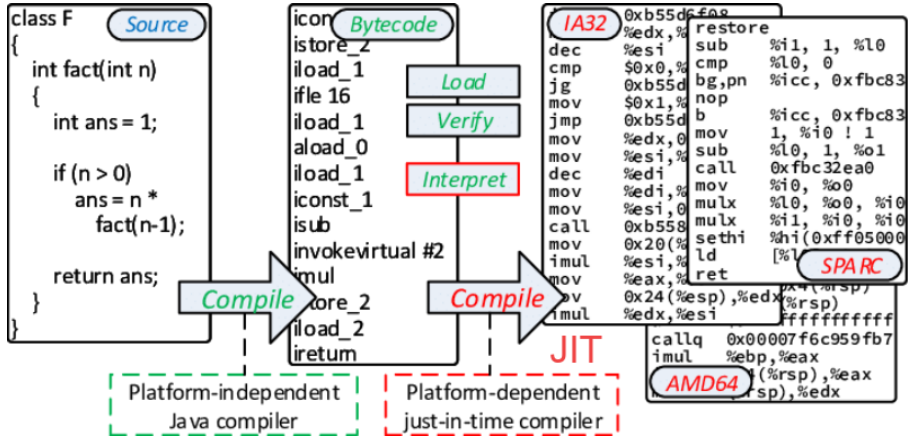


Abb. 14. Googles V8 Engine als dominanter Vertreter einer JIT HLL Prozess-VM. Bei numerischen Berechnungen erreicht V8 nahezu native Code Performanz (z.B. Matrixmultiplikation)

JIT Übersetzung



[DOI:10.1145/3067695.3082521]

Abb. 15. Java-Quellcode wird zuerst zu Bytecode kompiliert und anschließend als nativer Code interpretiert oder ausgeführt. Umfangreiche Optimierungen sind für die JIT-Kompilierungsphase reserviert. JAVA JVM JIT erzeugt aber schlechteren Code (langsamer) wie V8 JIT und die Konzepte konzentrien sich auf Programmcode (und nicht Daten). Dafür werden eine Vielzahl von Prozessoren und Rechneranlagen unterstützt - V8 ist nur für ia32/ia64 optimiert!

JIT Übersetzung

JIT benötigt Datentypen und Speicherinformation von Variablen, Funktionen und Funktionsargumenten!

- Dynamisch typisierte Sprachen (JavaScript, Lua) können diese Informationen nicht liefern, sondern Typsignaturen müssen zur Laufzeit vom JIT Compiler abgeleitet werden. Zeitaufwändige Operation!
  - Und noch viel schlimmer: Datentypen von Variablen, Funktionsargumenten und Rückgabewerten könen sich zur Laufzeit ändern!
  - Es muss für jede Typsigantur einer Funktion (oder Datenstruktur/Objekt) eine eigene native Übersetzung angelegt werden. Teuer im Speicher!
- Statisch typisierte Sptachen wie Java haben dieses Problem nicht, jedoch ist fraglich woher die Typinformationen kommen. Im Bytecode sind nur sehr begrenzte Informationen verfügbar.

JIT Übersetzung

Ein abschließendes Beispiel:

- Der gute alte Basic Interpreter
  - Aufteilung eines Programms in Zeilen (ursprünglich mit vorangestellter Zeilennummer)
  - Es gibt Sprungbefehle wie `goto <line>`
  - Basic wird Zeile für zeile entweder interpretiert oder kompiliert
  - Bei einem Vorwärtssprung ist die Programmcodeadresse einer noch folgenden Zeile noch nicht bekannt, erst bei der Ausführung wenn das Programm vollständig ist.
  - Wenn aber die Zeile des Ziels ermittelt wurde kann diese durch eine Codeadresse ersetzt werden: JIT

```
10: a=1
15: goto 20
20: b=a
-------------------------------------------------------------------------------------------
0             2           4      5         7              9    10         12
LINENUMBER(10) VARIABLE(a) ASSIGN NUMBER(1) LINENUMBER(15) GOTO LINENUMBER(20) LINENUMBER(20) ...
LINENUMBER(10) VARIABLE(a) ASSIGN NUMBER(1) LINENUMBER(15) GOTO ADDRESS(12)    LINENUMBER(20) ...
```

Vertiefende Literatur

[1] J. E. Smith and R. Nair, Virtual Machines Versatile Platforms for Systems and Processes. Elsevier, 2005.

[2] X.-F. Li, Advanced Design and Implementation of Virtual Machines. CRC Press, 2017.

[3] Reinhard Wilhelm , H. Seidl, Übersetzerbau VirtuelleMaschinen. Springer, 2007.

[4] lain D. Craig, Virtual Machines. Springer London, 2006.