

# QEMU

## Konzepte und Techniken virtueller Maschinen und Emulatoren

Bernd Schöbel

Lehrstuhl für Informatik 3  
Friedrich-Alexander-Universität Erlangen-Nürnberg

11.Juli 2007

# Was ist QEMU?



QEMU is a generic and open source machine emulator and virtualizer. [5]

QEMU ist ein universeller und quelloffener Maschinenemulator und Virtualisierer.

# Was ist QEMU?

QEMU is a generic and open source machine **emulator** and virtualizer. [5]

## Emulator

QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using **dynamic translation**, it achieves very good performances. [5]

# Was ist QEMU?

QEMU is a generic and open source machine emulator and **virtualizer**. [ 1 ]

## Virtualizer

QEMU achieves near native performances by executing the guest code directly on the host CPU. A host driver called the QEMU accelerator (also known as KQEMU) is needed in this case. The virtualizer mode requires that both the host and guest machine use x86 compatible processors. [ 5 ]

- erlaubt die Ausführung von Instruktionen **direkt** auf der Ziel-CPU
- nur bei gleicher Ziel- und Quell-CPU
- nur für x86 und x86\_64 CPUs
- benötigt Kernelmodul (Linux, FreeBSD) unter Win32 'Service'
- sehr sicher, da sämtliche Gastinstruktionen nur in Ring 3 ablaufen
- zwei Modi:
  - ▶ normal mode:
    - ★ Usermodecode wird direkt ausgeführt, Kernelcode wird dynamisch emuliert
  - ▶ full virtualization mode: ('-kernel-kqemu')
    - ★ Kernel- und Usermodecode werden direkt ausgeführt
    - ★ geht nicht mit jedem Gastsystem



# Betriebsarten

## Full system emulation

In this mode, QEMU emulates a full system (for example a PC), including one or several processors and various peripherals. It can be used to launch different Operating Systems without rebooting the PC or to debug system code.

## User mode emulation

In this mode, QEMU can launch processes compiled for one CPU on another CPU. It can be used to launch the Wine Windows API emulator (<http://www.winehq.org>) or to ease cross-compilation and cross-debugging.

# User mode emulation

- nur 2 Betriebssysteme unterstützt:
  - ▶ Linux (qemu-linux-user)
  - ▶ Mac OS X/Darwin (qemu-darwin-user)
- Linux Systemcall Konverter (byte sex, 32/64Bit)
- dynamic libraries für emulierte CPU müssen installiert sein
- clone() Systemcall erzeugt auch im Hostsystem einen neuen Thread
- Signal Handling
  - ▶ die meisten Signale werden direkt vom Hostsystem an das Gastsytem durchgereicht
  - ▶ nur `sigaction()` und `sigreturn()` müssen komplett emuliert werden



# Geschichte

- 2003 von Fabrice Bellard gestartet
- anfangs nur user-mode und keine Systememulation
- weitere Architekturen hinzugefügt
- ab Version 0.5.0 Systememulation mit Hardwareemulation
- ab 0.5.3 direkte Ausführung von x86 Instruktion auf x86 Hostsystemen mit KQEMU
- ab 0.7.0 64Bit und MMX/SSE/SSE2/PNI Unterstützung
- SMP Unterstützung bis zu 255 CPUs (seit Version 0.8.0)
- integrierter VNC Server (seit Version 0.8.1)
- neuste Version 0.9.0 vom 05.Feb.2007
- am 06.Feb.2007 wird KQEMU Open Source

# Features

## QEMU: open source processor emulator

- geschrieben von Fabrice Bellard
- Lizenz: GNU General Public License (GPL) und LGPL
- integrierter VNC Server
- USB Unterstützung
- SMP System Emulation
- Sound Unterstützung
- Snapshots
- QEMU Monitor
- Unterstützung für externes Debugging mit gdb
- Unterstützung für selbstmodifizierenden Code
- läuft auch ohne X11

# unterstützte Architekturen

## system emulation

- PC (x86 or x86\_64 processor)
- ISA PC (old style PC without PCI bus)
- PREP (PowerPC processor)
- G3 BW PowerMac (PowerPC processor)
- Mac99 PowerMac (PowerPC processor, in progress)
- Sun4m (32-bit Sparc processor)
- Sun4u (64-bit Sparc processor, in progress)
- Malta board (32-bit MIPS processor)
- ARM Integrator/CP (ARM926E or 1026E processor)
- ARM Versatile baseboard (ARM926E)

# unterstützte Architekturen

## user emulation

- x86
- PowerPC
- ARM
- MIPS
- Sparc32/64
- ColdFire(m68k)

CPUs are supported.

# unterstützte CPUs

## Host CPU

vollständig	x86, x86_64, PowerPC
testing	Alpha, Sparc32, ARM, S390, MIPS
dev	Sparc64, ia64, m68k

# unterstützte CPUs

## Gast CPU

Target CPU	User emulation	System emulation
x86	OK	OK
x86_64	Not supported	OK
ARM	OK	OK
SPARC	OK	OK
SPARC64	Dev only	Dev only
PowerPC	OK	OK
PowerPC64	Not supported	Dev only
MIPS	OK	OK
m68k (Coldfire)	OK	OK
SH-4	Dev only	Dev only
Alpha	Dev only	Dev only

# IBM kompatibler PC

- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card or dummy VGA card with Bochs VESA extensions
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- NE2000 PCI network adapters
- Serial ports
- Creative SoundBlaster 16 sound card
- ENSONIQ AudioPCI ES1370 sound card
- Adlib(OPL2) - Yamaha YM3812 compatible chip
- PCI UHCI USB controller and a virtual USB hub.

QEMU verwendet das Plex86/Bochs LGPL VGA BIOS. [9]

# IBM kompatibler PC

```
QEMU
debian:~# lspci -v
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
      Flags: fast devsel

00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton III]
      Flags: bus master, medium devsel, latency 0

00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton III] (prog-if 80 [Master])
      Flags: bus master, medium devsel, latency 64
      I/O ports at c000 [size=16]

00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI
      Flags: fast devsel, IRQ 9

00:02.0 VGA compatible controller: Cirrus Logic GD 5446 (prog-if 00 [VGA])
      Flags: fast devsel
      Memory at f0000000 (32-bit, prefetchable) [size=32M]
      Memory at f2000000 (32-bit, non-prefetchable) [size=4K]

00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8029(AS)
      Flags: fast devsel, IRQ 10
      I/O ports at c100 [size=256]

debian:~# _
```



# QEMU Monitor

Über den QEMU Monitor kann man dem QEMU Emulator Befehle geben, mit denen man z.B folgendes tun kann:

- Wechseldatenträger in das emulierte System einhängen oder entfernen ( CD-Roms, Disketten, USB-Laufwerke)
- Die Ausführung der Virtuellen Maschine anhalten oder fortsetzen
- Einen Zustand der Virtuellen Maschine auf Festplatte speichern oder laden (snapshoting)
- Den Zustand der Virtuellen Maschine mit einen externen Debugger untersuchen

Der QEMU Monitor ist auf Console 2.

Wechseln zu der Console  $n$  mit `Ctrl-Alt-n`.

# QEMU Monitor

```
QEMU
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=c0101a5a EDX=c0314000
ESI=c034a5c4 EDI=c1405de4 EBP=00000008 ESP=c0315fd8
EIP=c0101a8b EFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=1
ES =007b 00000000 ffffffff 00cff300
CS =0060 00000000 ffffffff 00cf9a00
SS =0068 00000000 ffffffff 00c09300
DS =007b 00000000 ffffffff 00cff300
FS =0000 00000000 00000000 00000000
GS =0000 00000000 00000000 00000000
LDT=0088 c034e020 00000027 00008200
TR =0080 c14018c0 00002073 00008900
GDT= c140a000 000000ff
IDT= c030b000 000007ff
CR0=8005003b CR2=b7f38600 CR3=1f033000 CR4=00000690
FCW=037f FSW=0000 IST=01 FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=f7b5a7d45d867e00 4003
FPR6=f7b5a7d45d867e00 4003 FPR7=f7b5a7d45d868000 4003
XMM0=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu) █
```

# QEMU Monitor

## einige Funktionen:

- Informationen über emuliertes Netzwerk, PCI Devices, Festplatten, USB Geräten und Register anzeigen
- Wechseldatenträger aus dem emuliertem System entfernen
- Emulation anhalten und wieder fortsetzen
- Logging starten
- Screendump machen
- Sound der Virtuellen Maschine aufzeichnen
- einen Snapshot erstellen oder laden
- gdbserver starten
- virtuellen oder physikalischen Memorydump ausgeben
- virtuellen PC neu starten
- Tastenkombination an das Gastsystem schicken

## Blockdatenträger unter QEMU

Es gibt drei Möglichkeiten auf Blockdatenträger unter QEMU zuzugreifen:

- Imagedateien
  - ▶ werden mit `qemu-img` erstellt und verwaltet
- Hosttreiber: `-hdb /dev/cdrom`
  - ▶ CD, Diskette: müssen beim Start von QEMU nicht eingelegt sein
  - ▶ Festplatte: `-hdb /dev/hdb` (ganze Platte angeben nicht einzelne Partition)
- virtuelle FAT Diskimages: `-hdb fat:/my_directory`
  - ▶ default read-only, read-write experimentel

# Image dateiformate

- raw (default)
- qcow2: QEMU Imageformat
  - ▶ AES Verschlüsselung
  - ▶ zlib Kompression
  - ▶ Unterstützung für mehrere VM snapshots
- qcow: altes QEMU Imageformat
- cow: User Mode Linux Copy On Write Imageformat
- vmdk: VMware 3 und 4 kompatible Formate
- cloop: Linux Compressed Loop Image. (z.B Knoppix CD)
- bochs: Bochs Imageformat

## user mode network stack

- `-net user`
- mit internen DHCP-Server (10.0.2.2) vergibt Adressen ab 10.0.2.15
- und interne DNS-Server (10.0.2.3)
- VM versteckt im Hostsystem
- über Socket kann man mehrere QEMU Instanzen in VLAN stecken

## TUN/TAP

- `-net tap`
- benötigt root-Rechte um TAP in Hostsystem einzurichten
- VM im Netz sichtbar

# Netzwerk

```
QEMU
10.0.2.2      ether  52:54:00:12:35:02  C      eth0
debian:~# ifconfig
eth0  Link encap:Ethernet  HWaddr 52:54:00:12:34:56
      inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:23 errors:0 dropped:0 overruns:0 frame:0
      TX packets:38 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:3109 (3.0 KiB)  TX bytes:3213 (3.1 KiB)
      Interrupt:10 Base address:0xc100

lo      Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
      RX packets:22 errors:0 dropped:0 overruns:0 frame:0
      TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:2128 (2.0 KiB)  TX bytes:2128 (2.0 KiB)

debian:~# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.3                 ether   52:54:00:12:35:03   C                      eth0
10.0.2.4                 (incomplete)                      eth0
10.0.2.2                 ether   52:54:00:12:35:02   C                      eth0
debian:~# _
```

# Dynamic Translation

## Interpreter

Für jede Instruktion der emulierten CPU gibt es eine Funktion, die diese simuliert. Bei der Emulation der Gast-CPU wird jetzt für jede auszuführende Instruktion die entsprechende Funktion aufgerufen, die diese simuliert.

Bochs [6] macht das so.

Interpretation ist einfach aber leider sehr langsam.



# Dynamic Translation

## Die Grundidee von 'Dynamic Translation'

- C-Funktionen schreiben, die die OP-Codes der Gast-CPU emulieren
- den Compiler Objekdateien für diese Funktionen generieren lassen
- den Zielcode generieren durch aneinanderreihen der erzeugten Objekdateien

Diese Methode ist viel schneller als die Interpretation, wenn der Code öfter als einmal ausgeführt wird.

## 'Dynamic Translation' im Detail

- ❶ Opcode in Micro Operations zerlegen
  - ▶ Um die Anzahl der zu implementierenden C-Funktionen klein zu halten werden die Ziel CPU-opcodes in kleinere Teile sogenannte *Micro Operations* zerlegt. Typischerweise einige Hundert anstatt die Kombination aller Instruktionen und Operanden der Ziel CPU.
  - ▶ Dieser Code ist von Hand geschrieben
- ❷ mit GCC die kleinen C Funktionen zu einer Objekdatei übersetzen
- ❸ mit `dynngen` eine Code-Generator erzeugen
  - ▶ Compile Time Tool
  - ▶ nimmt die generierten Objekdateien als input
  - ▶ erzeugt daraus einen dynamischen Code-Generator
  - ▶ dieser Code-Generator wird dann zur Laufzeit aufgerufen um eine Host-Funktion aus mehreren micro operations zu erzeugen

Das `dyngen` Tool ist der Schlüssel zum Übersetzungsprozess von QEMU. Hier die einzelnen Schritte die es ausführt:

- Objektdatei parsen und Symboltabelle, Relokationseinträge und Code Sektion einlesen
- Prolog und Epilog von Funktionen erkennen und ignorieren
- Die Relokationen für jede Micro Operation anschauen um die Anzahl der konstanten Parameter herauszufinden
- Das Offset der einzelnen Parameter merken
- Ein Memory Copy in C generieren um den Code zu kopieren
- Anschließend Instruktionen mit konstanten Parameter patchen

## Anmerkungen:

- beim Übersetzen der Micro Operation Funktionen werden spezielle Compilerflags verwendet, damit Prolog und Epilog einfacher erkannt werden können
- eine Dummy Assembler Instruktion erzwingt, dass jede Funktion nur einen einzigen Return-Punkt hat
- Prozess sehr Compiler abhängig

# Dynamic Translation - Beispiel

## Beispiel

Wir müssen folgende PowerPC Instruktion nach x86 Code übersetzen:

```
addi r1, r1, 42           # r1 = r1 + 42
```

# Dynamic Translation - Beispiel

## ❶ Opcode in Micro Operations zerlegen

### Beispiel

```
addi r1, r1, 42                # r1 = r1 + 42
```

folgende Micro Operations werden generiert:

```
movl_T0_r1                    # T0 = r1
movl_T0_im 42                 # T0 = T0 + 42
movl_r1_T0                    # r1 = T0
```

Die Register T0, T1, T2 werden normalerweise über ein GCC-Makro auf Register der Ziel-CPU gemappt.

- ❷ mit GCC die kleinen C Funktionen zu einer Objekdatei übersetzen
- ❸ mit `dynngen` eine Code-Generator erzeugen

# Dynamic Translation - Beispiel

## Implementierung der Micro Operation `movl_T0_r1`

```
void op_movl_T0_r1(void)
{
    T0 = env->regs[1];
}
```

`env` ist eine Struktur die den Zustand der emulierten CPU enthält.  
`env->regs[32]` ist ein Array der 32 PowerPC Register.

# Dynamic Translation - Beispiel

## Implementierung der Micro Operation `movl_T0_im`

```
extern int __op_param1;

void op_addl_T0_im(void)
{
    T0 = T0 + ((long) (&__op_param1));
}
```

Der konstante Parameter `__op_param1` wird zur Laufzeit aufgelöst.



# Dynamic Translation - Beispiel

- 1 Opcode in Micro Operations zerlegen
- 2 mit GCC die kleinen C Funktionen zu einer Objekdatei übersetzen
- 3 mit `dynngen` eine Code-Generator erzeugen

# Dynamic Translation - Beispiel

## der von `dyngen` generierte Code-Generator

```
[...]  
for(;;) {  
    switch(*opc_ptr++) {  
        case INDEX_op_movl_T0_r1:  
            [...]  
        case INDEX_op_addl_T0_im:  
            [...]  
        case INDEX_op_movl_r1_T0:  
            [...]  
    }  
}  
[...]
```

`opc_ptr` Zeiger auf den zu übersetzenden Micro Operation Stream.

# Dynamic Translation - Beispiel

## der von dyngen generierte Code-Generator

```
case INDEX_op_movl_T0_r1:
{
    extern void op_movl_T0_r1();
    memcpy(gen_code_ptr,
           (char *)&op_movl_T0_r1+0,
           3);
    gen_code_ptr += 3;
    break;
}
```

`opc_ptr` Zeiger auf den zu übersetzenden Micro Operation Stream.  
`gen_code_ptr` Zeiger auf den zu erzeugenden Ziel Code.

# Dynamic Translation - Beispiel

## der von dyngen generierte Code-Generator

```
case INDEX_op_addl_T0_im:
{
    long param1;
    extern void op_addl_T0_im();
    memcpy(gen_code_ptr,
           (char *)&op_addl_T0_im+0,
           6);
    param1 = *opparam_ptr++;
    *(uint32_t *) (gen_code_ptr + 2) = param1;
    gen_code_ptr += 6;
    break;
}
```

`opc_ptr` Zeiger auf den zu übersetzenden Micro Operation Stream.

`gen_code_ptr` Zeiger auf den zu erzeugenden Ziel Code.

`opparam_ptr` Zeiger auf Parameter der Micro Operations

# Dynamic Translation - Beispiel

## original PowerPC Gast-Code

```
addi r1, r1, 42
```

```
# r1 = r1 + 42
```

## der von dem Code-Generator erzeugte x86 Ziel-Code

```
# movl_T0_r1
```

```
movl 0x4(%ebp), %ebx
```

```
# ebx = env->regs[1]
```

```
# addl_T0_im 42
```

```
addl $0x2a, %ebx
```

```
# ebx = ebx + 42
```

```
# movl_r1_T0
```

```
movl %ebx, 0x4(%ebp)
```

```
# env->regs[1] = ebx
```

T0 ist auf x86 auf `%ebx` gemappt.

Der CPU Zustand wird über `%ebp` erreicht.

# Translation Cache

- Gast-Code wird in Blöcken übersetzt (TB: Translated Block)
- Ein TB endet bei einer Sprunginstruktion
- TBs werden gecached (LRU)
- Nach der Ausführung eines TBs werden Interrupts und Exceptions bearbeitet

# Testsysteme

## Hostsystem

- Linux 2.6.21.1 #1 SMP x86\_64 GNU/Linux 4.0
- Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
- 2GB RAM
- Raid 1 auf 2 SAMSUNG HD501LJ (SATA, 7200rpm)

## Gastsystem

- Linux 2.6.18-4-686 #1 SMP i686 GNU/Linux 4.0
- QEMU 0.9.0
- `qemu -m 768 -hda debian-32bit.img -boot c`
- QEMU Disk Image im Format `raw`

# Test I - Linux Kernel

Linux Kernel Sourcen `linux-2.6.22.tar.bz2` entpacken und packen.

## Ergebnisse der Messungen

	Entpacken	Packen
Hostsystem	0m14s	1m05s
QEMU	3m40s	15m01s
relative Geschwindigkeit	6%	7%



## Test II - Festplattendurchsatz

`hdparm -tT /dev/XXX` Aufruf

### Ergebnisse der Messungen

MB/sec	buffered disk reads	cached reads
Hostsystem	72.89	4559.88
QEMU	37.12	333.19
relative Geschwindigkeit	50%	13%

# Fragen?

# Quellen I



## [1] QEMU: Home

<http://fabrice.bellard.free.fr/qemu/>



## [2] Wikipedia: QEMU

<http://en.wikipedia.org/wiki/QEMU>



## [3] QEMU porting

<http://libvncserver.sourceforge.net/qemu/qemu-porting.html>



## [4] QEMU, a Fast and Portable Dynamic Translator

[http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full\\_papers/bellard/bellard.pdf](http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf)



## [5] QEMU: About

<http://fabrice.bellard.free.fr/qemu/about.html>


# Quellen II

 [6] Bochs  
<http://bochs.sourceforge.net/>

 [7] Wikipedia: Dynamic recompilation  
[http://en.wikipedia.org/wiki/Dynamic\\_recompilation](http://en.wikipedia.org/wiki/Dynamic_recompilation)

 [8] Wikipedia: Dynamic translation  
[http://en.wikipedia.org/wiki/Dynamic\\_translation](http://en.wikipedia.org/wiki/Dynamic_translation)

 [9] Plex86/Bochs LGPL VGABios  
<http://www.nongnu.org/vgabios/>

 [10] QEMU templates ala Fabrice  
<http://libvncserver.sourceforge.net/qemu/qemu-templates-ala-Fabrice.txt>

 [11] Johannes Schindelin : QEMU - a Xen alternative  
<http://www.avc-cvut.cz/avc.php?id=3284>