Projektarbeit CXL

Julius Armbrüster Moritz Riefer

16.11.2025

Emulation mit QEMU

CXL kann mit `qemu-system-x86_64` emuliert werden[1]. Hierzu muss die Netzwerkstruktur der CXL-Geräte per Kommandozeilenargumente definiert werden. Eine Beispielstruktur mit einem CXL-Type-3-Gerät, dass 256 MB Persistent Memory bereitstellt, sieht wie folgt aus:

```
qemu-system-x86_64 \
  -machine q35,cxl=on \
  -m 2048 \
  -smp 2 \
  -cpu max \
  -drive file=debian-x86.img,format=qcow2 \
  -boot d \
  -net nic -net user,hostfwd=tcp::10022-:22 \
  -object memory-backend-file,id=cxl-mem0,share=on,mem-path=cxl-nvdimm0,size=256M \
  -object memory-backend-file,id=cxl-lsa0,share=on,mem-path=cxl-lsa0,size=256M \
  -device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.0 \
  -device cxl-rp,port=0,bus=cxl.0,id=cxl-rp0,chassis=0,slot=2 \
  -device cxl-type3,bus=cxl-rp0,persistent-memdev=cxl-mem0,lsa=cxl-lsa0,id=cxl-pmem0 \
  -M cxl-fmw.0.targets.0=cxl.0,cxl-fmw.0.size=256M
```

D3OS

In D3OS haben wir auf dem Code von Fabian Gremme aufgebaut und einige weitere Memory-Mapped-Register ausgelesen, die sich auf CXL beziehen. Der Code befindet sich in einem öffentlichen GitHub-Repo[2].

Ab CXL 2.0 ist der Speicher auf zwei Arten verwendbar. Einerseits über das sogenannte Fixed-Memory-Window, über den der gesamte Speicher als ein zusammenhängender Block addressierbar ist. Das Fixed-Memory-Window ist in der CEDT ACPI-Tabelle definiert. Andererseits können bestimmte Blöcke (mind. 256MB) des Speichers dynamisch für verschiedene Hosts erreichbar gemacht werden. Hierfür müssen jedoch sogenannte Decoder konfiguriert werden, die für das Routing verantwortlich sind.

Mit dem Befehl `cargo make --no-workspace qemu-simple-cxl` kann D3OS in QEMU mit der oben beschriebenen CXL-Struktur gestartet werden.

Auf der Konsole werden einige weitere Debugausgaben bzgl. CXL angezeigt. Unter anderem ist der Inhalt der CEDT Tabelle als auch die Register für die Decoder zu sehen.

Leider konnten wir weder, den Speicher über das Fixed-Memory-Window zu nutzen, noch konnten wir die Decoder konfigurieren, um den Speicher hierüber zu verwenden.

Daraufhin wollten wir zunächst versuchen, ob wir in Linux auf den über CXL bereitgestellten Speicher zugreifen können.

[1] https://www.qemu.org/docs/master/system/devices/cxl.html
[2] https://github.com/jarmbrue/D3OS

Probleme unter Linux

Unter Linux wird in der `cxl`-Utility die korrekte Struktur angezeigt.

```
$ cxl list
[
  {
    "memdev":"mem0",
    "pmem_size":268435456,
    "serial":0,
    "host":"0000:0d:00.0",
    "firmware_version":"BWFW VERSION 00"
  }
]
```

Ungünstigerweise lässt sich mit QEMU 10.1.0 keine CXL-Region erstellen. Als Host haben wir sowohl Apple-Silicon mit macOS als auch x86_64 Debian 13 und Windows 11 ausprobiert. Das Guest-OS war Debian 13 mit Kernelversion 6.15.4-1.

```
$ sudo cxl create-region -d decoder0.0 -s 256M -m mem0
cxl region: create_region: region0: failed to commit decode: No such device or address
cxl region: cmd_create_region: created 0 regions
```

So läuft CXL unter QEMU in Linux

Mit dieser Anleitung[3] hat das Emulieren eines CXL-Type-3-Geräts in QEMU erfolgreich funktioniert. Hierfür werden sowohl ein gepatchtes QEMU als auch ein gepatchter Kernel verwendet.

QEMU-Fork von moking klonen und die Version dcd-v6 kompilieren.

```
$ git clone https://github.com/moking/qemu
$ git switch dcd-v6
$ ./configure --target-list=x86\_64-softmmu --enable-debug
$ make -j 16
```

Linux-Kernel mit DCD-Support von weiny2 klonen und für das Kompilieren konfigurieren.

```
$ git clone https://github.com/weiny2/linux-kernel
$ git switch dcd-2024-03-24
$ make menuconfig
```

Folgende Config-Einstellungen müssen aktiviert sein:

```
CONFIG_ARCH_WANT_OPTIMIZE_DAX_VMEMMAP=y
CONFIG_CXL_BUS=m
CONFIG_CXL_PCI=m
CONFIG_CXL_MEM_RAW_COMMANDS=y
CONFIG_CXL_ACPI=m
CONFIG_CXL_PMEM=m
CONFIG_CXL_MEM=m
CONFIG_CXL_PORT=m
CONFIG_CXL_SUSPEND=y
CONFIG_CXL_REGION=y
CONFIG_CXL_REGION_INVALIDATION_TEST=y
CONFIG_CXL_PMU=m
CONFIG_ND_CLAIM=y
CONFIG_ND_BTT=m
CONFIG_ND_PFN=m
CONFIG_NVDIMM_DAX=y
CONFIG_DAX=m
```

[3] https://github.com/moking/moking.github.io/wiki/Basic:-CXL-Test-with-CXL-emulation-in-QEMU

```
CONFIG_DEV_DAX=m
CONFIG_DEV_DAX_PMEM=m
CONFIG_DEV_DAX_HMEM=m
CONFIG_DEV_DAX_CXL=m
CONFIG_DEV_DAX_HMEM_DEVICES=y
CONFIG_DEV_DAX_KMEM=m
```

Kernel kompilieren und installieren. Das Kernel-Image liegt anschließend am Pfad: `arch/x86/boot/bzImage`

```
$ make -j 16
$ sudo make modules\_install
```

Mit `$QEMUDIR` und `$LINUXDIR` meinen wir im Folgenden immer den Repository-Ordner für QEMU bzw. Linux.

Um den Kernel zu starten, verwenden wir den eingebauten Bootloader von QEMU. Hierzu erstellen wir ein Image mit Debian und mounten es in den Ordner `cxl-dir`.

```
$ $QEMUDIR/build/qemu-img create cxl-img 16G
$ sudo mkfs.ext4 cxl-img
$ sudo mount -o loop cxl-img cxl-dir
$ sudo debootstrap --arch amd64 stable cxl-dir
```

Jetzt kann in `cxl-dir` gechroot und weitere Einstellungen vorgenommen werden, wie einen User mit Root-Rechten erstellen und einige Pakete installieren.

```
$ sudo chroot cxl-dir /bin/bash
$ useradd -m $USER
$ usermod -aG sudo $USER
$ apt install kmod pciutils cxl ndctl sudo daxctl
```

Mit STRG-D kann aus der chroot-Shell zurückgekehrt werden. Es sollte nun noch `cxl-dir` unmounten werden.

```
$ sudo umout \$DIR
```

Der folgende Command startet QEMU mit dem kompilierten Linux-Kernel und einer CXL-Struktur mit einem Memory-Device. Außerdem wird dyndbg für einige CXL-Module aktiviert.

```
$ $QEMUDIR/build/qemu-system-x86_64 -s
    -kernel $LINUXDIR/arch/x86/boot/bzImage
    -append "root=/dev/sda rw console=ttyS0,115200 ignore_loglevel nokaslr \
    cxl_acpi.dyndbg=+fplm cxl_pci.dyndbg=+fplm cxl_core.dyndbg=+fplm \
    cxl_mem.dyndbg=+fplm cxl_pmem.dyndbg=+fplm cxl_port.dyndbg=+fplm \
    cxl_region.dyndbg=+fplm cxl_test.dyndbg=+fplm cxl_mock.dyndbg=+fplm \
    cxl_mock_mem.dyndbg=+fplm dax.dyndbg=+fplm \
    dax_cxl.dyndbg=+fplm device_dax.dyndbg=+fplm" \
    -smp 4 -serial mon:stdio -nographic -qmp tcp:localhost:4444,server,wait=off \
    -netdev user,id=network0,hostfwd=tcp::2024-:22 -device e1000,netdev=network0 \
    -monitor telnet:127.0.0.1:12345,server,nowait \
    -drive file=cxl-dir,index=0,media=disk,format=raw \
    -machine q35,cxl=on -m 4G \
    -virtfs local,path=./lib/modules,mount_tag=modshare,security_model=mapped \
    -virtfs local,path=/home/$USER,mount_tag=homeshare,security_model=mapped \
    -object memory-backend-file,id=cxl-mem1,share=on,mem-path=/tmp/cxltest.raw,size=512M \
    -object memory-backend-file,id=cxl-lsa1,share=on,mem-path=/tmp/lsa.raw,size=512M \
    -device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
    -device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
    -device cxl-type3,bus=root_port13,memdev=cxl-mem1,lsa=cxl-lsa1,id=cxl-pmem0 \
    -M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G,cxl-fmw.0.interleave-granularity=8k
```

Nun kann sich mit dem erstellten User eingeloggt werden. Nach jedem Start müssen die CXL-Module geladen werden:

```
$ sudo modprobe -a cxl\_acpi cxl\_core cxl\_pci cxl\_port cxl\_mem cxl\_pmem
```

Um den Speicher auf dem CXL-Memory-Device nutzen zu können, muss zunächst eine Region und einen Namespace für diese Region erstellt werden. Anschließend kann der Namespace dem RAM hinzufügt werden. Der Speicher sollte dann in `lsmem` zu sehen sein.

```
$ sudo cxl create-region -m -d decoder0.0 -w 1 mem0 -s 512M
$ sudo ndctl create-namespace -m dax -r region0
$ sudo daxctl reconfigure-device --mode=system-ram --no-on
```

Zusätzliche Informationen:

- Namespace kann auch als fsdax erstellt werden, dieser kann als Dateisystem formatiert und mountet werden.
- Veränderungen sind in der /tmp/cxltest.raw und /tmp/lsa.raw zu sehen.
- Diese Veränderungen in den beiden Dateien bleiben bis zum Löschen der Dateien bestehen.
- Beim Neustarten wird der Namespace nicht mehr erkannt und das Erstellen eines neuen Namespaces schlägt fehl.

Auf echter Hardware

Am Lehrstuhl haben wir einen Gigabyte R263-Z30-AAH2 Server, der CXL 1.1 unterstützt. In dem Server ist ein Smart Modular CXA-4F1W CXL Memory Expander, dieser unterstützt CXL 2.0. Auf dem Server läuft zurzeit Linux Debian.

Unter Linux wird das CXL-Gerät als PCI-Gerät erkannt.

```
lspci
c1:00.0 CXL: SMART Modular Technologies Device c241
```

In den DVSEC (Designated Vendor-Specific) Capabilities ist als Capability `Mem+` angegeben, heißt, es kann als Memory Device verwendet werden, aber es ist in den Control-Registern nicht aktiviert `Mem-`

```
sudo lspci -vvv
c1:00.0 CXL: SMART Modular Technologies Device c241 (prog-if 10 [CXL Memory Device (CXL 2.x)])
Capabilities: [224 v1] Designated Vendor-Specific: Vendor=1e98 ID=0000 Rev=2 Len=60: CXL
    CXLCap: Cache- IO+ Mem+ Mem HW Init+ HDMCount 1 Viral+
    CXLCtl: Cache- IO+ Mem- Cache SF Cov 0 Cache SF Gran 0 Cache Clean- Viral-
```

Des Weiteren wird die CXL-Karte nicht von der `cxl`-Utility erkannt.

```
cxl list
Warning: no matching devices found

[
]
```

In den Kernelmeldungen wird ein Fehler wegen nicht gefundener Register angezeigt. Es ist möglich, dass diese Meldungen erscheinen, weil das Mainboard nur CXL 1.1 unterstützt. Die Mailbox ist z.B. erst ein CXL 2.0 Feature.

```
sudo dmesg | grep cxl
cxl_pci 0000:c1:00.0: enabling device (0000 -> 0002)
cxl_pci 0000:c1:00.0: Mapped CXL Memory Device resource 0x0000030020f10000
cxl_pci 0000:c1:00.0: registers not found: status mbox memdev
```

CXL-Register manuell auslesen

Wir haben ein C Programm geschrieben, dass eine PCI-BAR mit `mmap` in den Speicher mapped und einen Hex-Dump erstellt. Die Base Address und den Offset haben wir mit `lspci` ausgelesen.

```
$ sudo lspci -vvv
c1:00.0 CXL: SMART Modular Technologies Device c241 (prog-if 10 [CXL Memory Device (CXL 2.x)])
Region 0: Memory at 30020f00000 (64-bit, prefetchable) [size=256K]
Capabilities: [274 v1] Designated Vendor-Specific: Vendor=1e98 ID=0008 Rev=0 Len=28: CXL
    Block1: BIR: bar0, ID: component registers, offset: 0000000000000000
    Block2: BIR: bar0, ID: CXL device registers, offset: 0000000000010000
```

Die Struktur der *component register* stimmt mit der Spezifikation (CXL 2.0 Sektion 8.2.4) überein. Jedoch passt der Dump der *CXL device register* nicht zur Spezifikation (CXL 2.0 Sektion 8.2.8).

D3OS auf dem CXL-Server

Um D3OS auf dem Server zu testen, ohne Linux löschen zu müssen, steckt in dem Server ein USB-Stick, der als Boot-Device verwendet werden kann. Auf den USB-Stick haben wir mit `dd` ein D3OS-Image installiert. Der Server hat auch auf den ersten Blick ohne Fehler in D3OS gebootet. Leider war in der Remote-KVM auf der Management-Weboberfläche nur folgende Ausgabe zu sehen:

```
[ INFO]: towboot/src/boot/video.rs@027: setting up the video...
[ WARN]: towboot/src/boot/video.rs@038: color depth will be 24-bit, but the kernel wants 32
[ INFO]: towboot/src/boot/video.rs@096: set (800, 600) as the video mode
[ INFO]: towboot/src/main.rs@096: booting D3OS...
```

In der Management-Web-Oberfläche gibt es auch eine Option Serial-over-LAN, hier werden jedoch beim Booten nur zufällige Zeichen angezeigt. Vermutlich stimmen die Baudraten nicht überein. Wir haben schon die Baudraten 9600 und 115200 ausprobiert. Eine andere Möglichkeit wäre, ein CLI-Tool wie `ipmitool` zu verwenden, um die Seriellen vom Server zu lesen.

Fehler Suche

Im BIOS haben wir im Reiter *CXL Common Options* alles bis auf die CXL-Encryption aktiviert. Die Web-Version des BIOS ist sehr vereinfacht und es fehlen eineige CXL-Einstellungen. Deshalb ist es zu empfehlen, über Remote-KVM auf das BIOS zuzugreifen.

Nach dem Austausch mit dem Yussuf Khalil vom KIT (Karlsruher Institut für Technologie) stellte sich heraus, dass sie ein ähnliches Setup wie wir verwenden und bei ihnen funktioniert es. Infolgedessen sind wir noch einmal gemeinsam alle BIOS-Optionen durchgegangen und haben CXL SMP deaktiviert, das hat aber zu keiner Veränderung geführt

Außerdem wurde das BIOS auf die Version `R29_F43` geupdatet.

Durch den Austausch mit Yussuf Khalil ist uns aufgefallen, dass die CXL-Karte im falschen PCI-Slot gesteckt hat. Der Server besitzt nämlich sowohl PCI-4.0 als auch PCI-5.0 Slots. Die Karte steckte in einem der PCI-4.0 Slots. Nach dem Umstecken der Karte lässt sich der Server jedoch nicht mehr booten. Beim Einschalten des Servers, landet dieser in einem Boot-Loop und und das BIOS ist nicht mehr erreichbar.

Ausblick

Folgende Fragestellungen sind mögliche Ansätze, um an CXL weiter zu arbeiten:

- Was wurde durch dcd-Patches in QEMU bzw. Linux geändert?
- Funktioniert CXL mit einem “normaler” Linux kernel in dem gepachten QEMU?
- Ist die CXL Karte wirklich der Grund für den Boot-Loop?