MANCHESTER 1824
The University of Manchester

UK RISE 2025: SECCOM, Setting up QEMU and CXL for Composability/Security Research

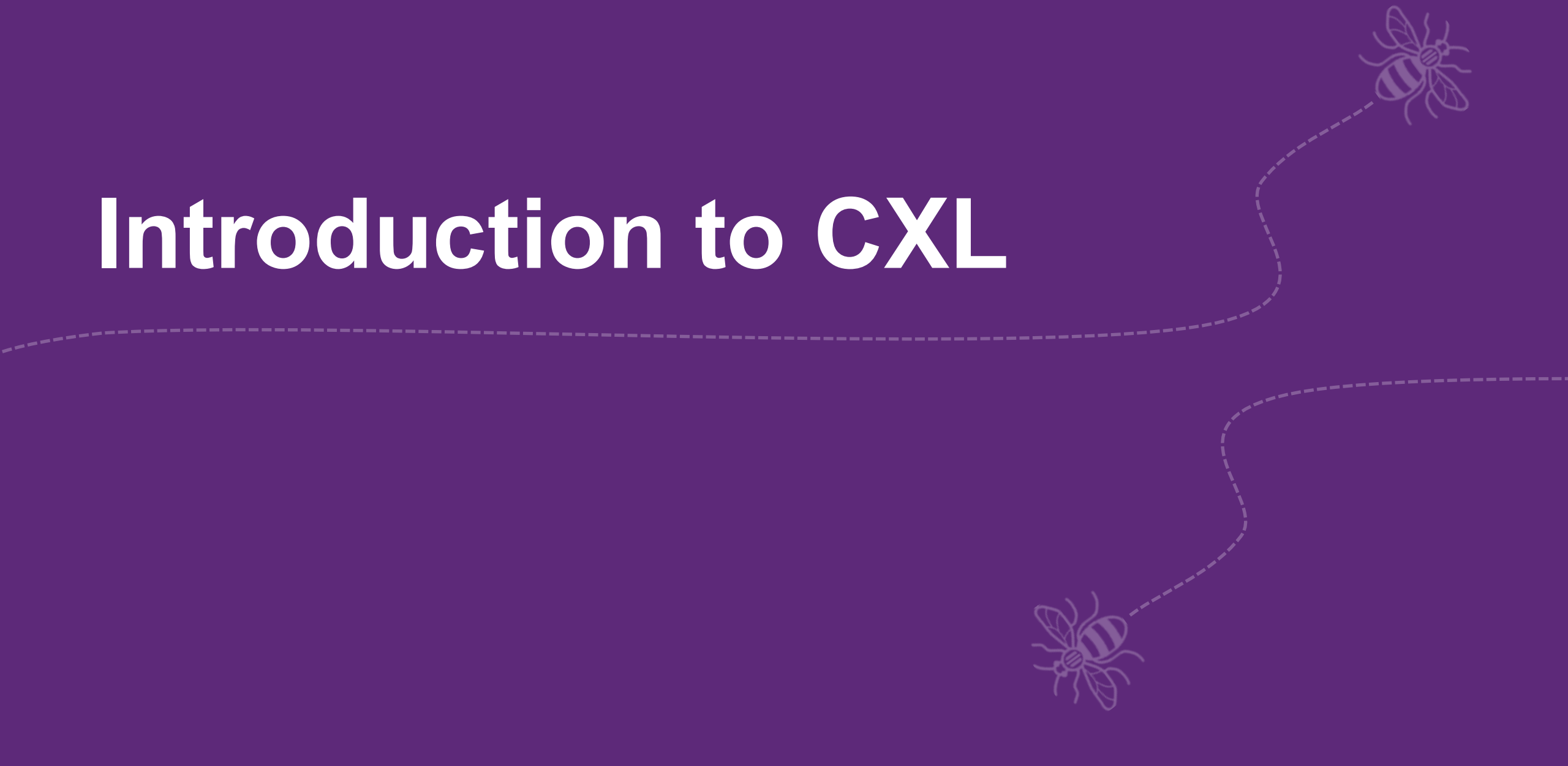
Dr. Joshua Lant, Research Fellow
joshua.lant@manchester.ac.uk

Topics of Presentation

- Introduction to CXL
  - General introduction
  - Fabric Management/Composability
  - Security Implications

- Setting up a QEMU emulated CXL system
  - Existing Tools
  - Steps for Manual Setup

Introduction to CXL



What is CXL?

- Compute eXpress Link, version 3.2 (Nov ‘24)
- Open interconnect standard
  - Built on top of PCIe’s physical/data-link layer
  - Does not use normal PCIe TLPs (cache/mem)
- Addresses several modern datacentre challenges
  - Lowering TCO
    - Memory pooling/scaling/disaggregation
    - Increased resource utilization
  - Increasing memory capacity/bandwidth/data locality
    - Disaggregation
    - Peer to peer device transfer
  - Zero copy
    - Shared cpu/device memory space
    - Cache coherence from device to host
    - Memory-like access semantics

CXL Defines 3 Protocols

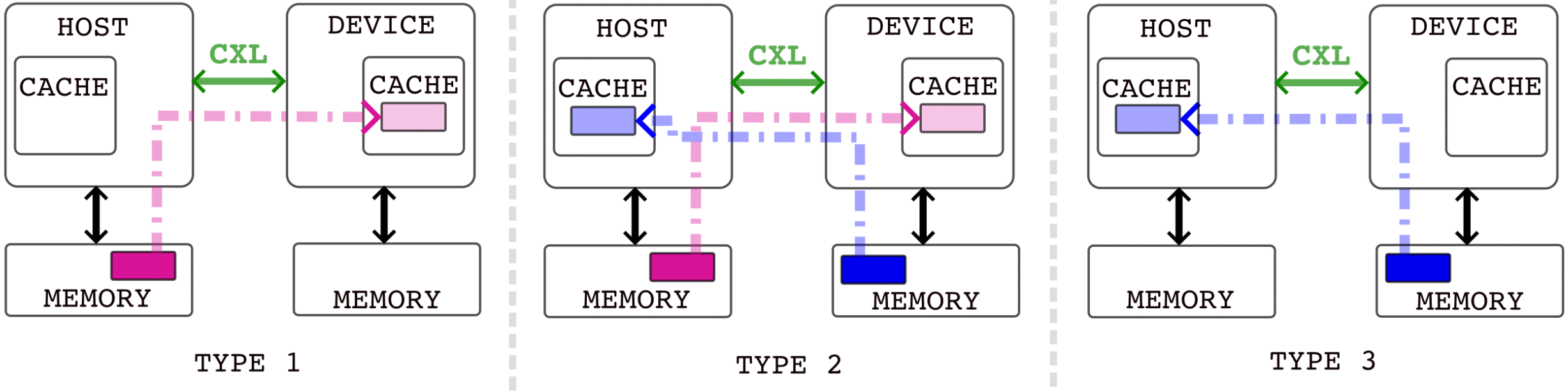
- Transaction layer protocols:
- CXL.io
  - Similar to PCIe configuration mechanisms.
  - Initialization, configuration, status, control etc.
  - I/O access for anything other than memory (CXL.mem) or cache (CXL.cache) operations.
  - The Linux CXL driver exposes access to .io functionality via:
    - sysfs interfaces
    - /dev/cxl/ devices (expose direct access to device mailboxes).
- CXL.cache
  - Allows a device to coherently access and cache host memory.
  - (Mostly) transparent to Linux once configured.
- CXL.mem
  - Allows a host to coherently access and cache device memory.
  - (Mostly) transparent to Linux once configured.

CXL Defines 3 Device Types

- Type 1
- (CXL.io + CXL.cache)
  - **Device caching of host-attached memory**
  - e.g Smart NIC.

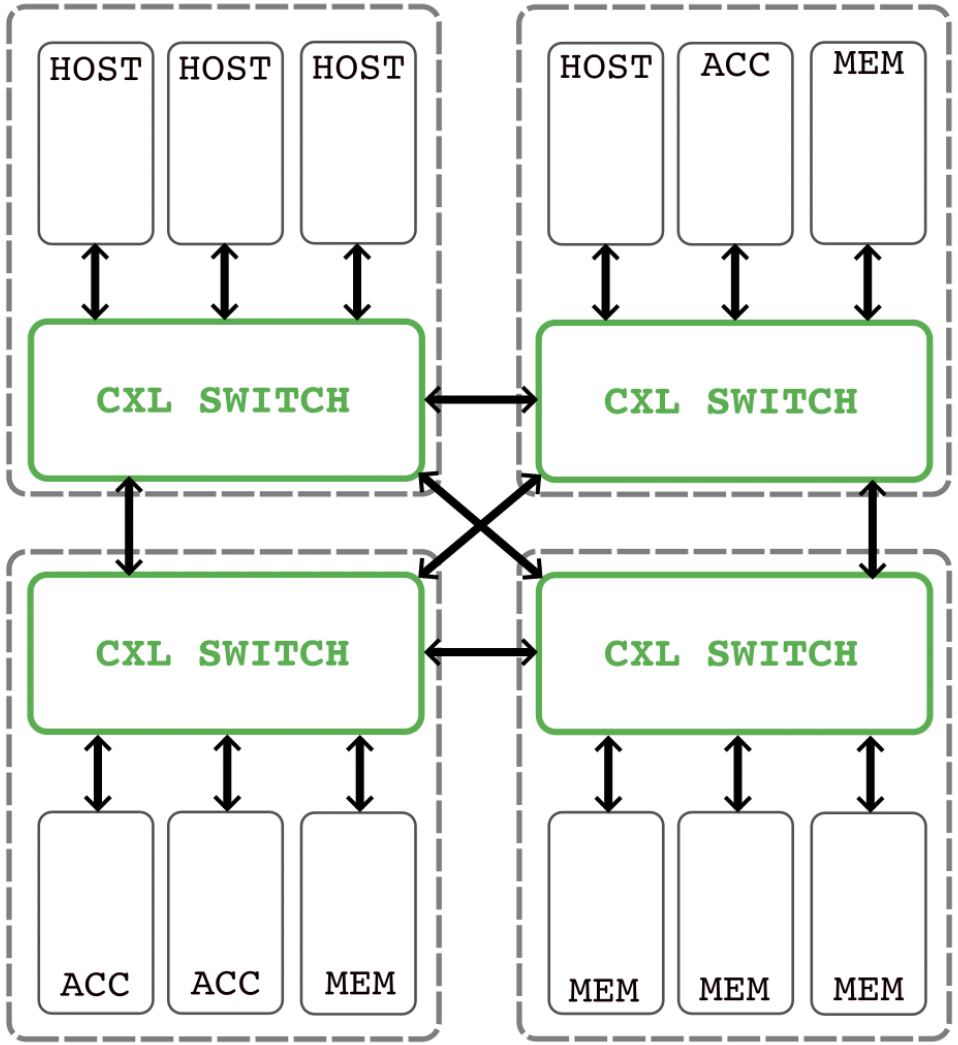
- Type 2
- (CXL.io + CXL.cache + CXL.mem)
  - **Device caching of host-attached memory**
  - **Host caching of device memory**
  - E.g. an accelerator with HBM

- Type 3
- (CXL.io + CXL.mem)
  - **Host caching of device memory**
  - e.g. memory expander modules.



CXL Switches

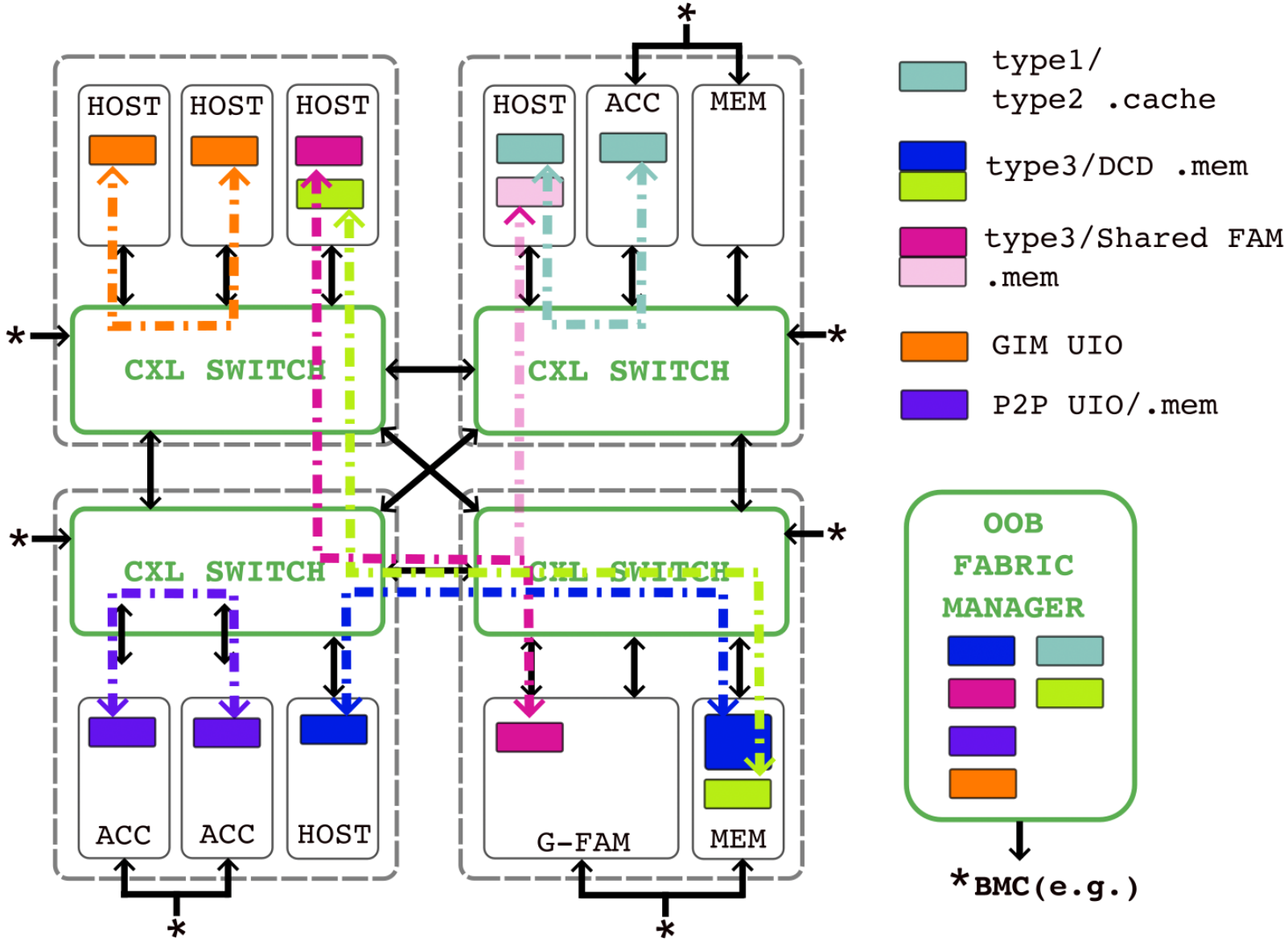
- 2 Switch types
  - PBR (Port Based Routing)
  - HBR (Hierarchy Based Routing); akin to standard PCIe enumeration techniques.
- PBR allows far greater scope for:
  - Memory Sharing
  - Composability/Disaggregation
  - Complexity (non-tree topologies)
- All work currently on HBR, no (public) software enablement for PBR…
- PBR routing *must* be set up by a Fabric Manager (FM)…



Fabric Manager, what and where is it?

- Gives us composability:
  - Endpoints/regions can be added/removed at runtime
  - Without this, static topology is needed pre-boot
- Separate, undefined device/controller/software which:
  - Controls the configuration of device memory regions, switch routing, etc.
  - What it does is defined (API), but not the how!
- Fabric manager can be:
  - In-band
    - PCIe MMIO to mailboxes (The host can be the Fabric Manager)
  - Out of band
    - MCTP messages
      - Confusing, can also be over the PCIe fabric, as a VDM. (The host can be the Fabric Manager). Tunnel commands through CXL switches using "Tunnel Management Command"
    - I2C/I3C/BMC/eth e.g. (Separate network/controller for Fabric Manager)

CXL: A Fully Composable System Architecture



Out-of-Band Fabric Manager. i.e. from BMC, SMBus, I2C,I3C or dedicated PCIe links

CXL Security Implications

- New technology
  - Many complex modes of communication/data sharing
  - CXL regions and protections are coarse grained
- Decisions left up to vendors/device manufacturers
  - Protections for peer-to-peer UIO/.mem
  - type1/2 endpoint drivers
  - Fabric Manager
  - ATS handling
- Protecting multiple workloads using same endpoint?
- DOS attacks
- CXL TSP cannot be used in switched setup (as of v3.2)

QEMU/CXL System Setup

QEMU

- Quick EMUlator
  - Full system emulation
    - CPU emulation, instruction translation
    - Emulation of full system hardware, peripherals etc.
    - Emulates many architectures
    - Uses internal TCG (Tiny Code Generator) translator
  - Hypervisor support
    - Manager for VMs under hypervisor (KVM/Xen etc.)
    - Provides device emulation (beyond simple virtio devices)
  - User-mode emulation
    - Binary translation for single userspace application
    - For rapid cross-compile/debug
- CXL fork for QEMU runs ahead of kernel, which runs ahead of mainline QEMU
  - Cant be upstreamed to QEMU until there is kernel support

Third Party Tools

- There are some tools aimed to make setup simpler
  - Internally, these use methods I will show, wrapped in nice scripts.
- If you need more control, build from scratch

- Many people use:
  - https://github.com/pmem/run_qemu

- This setup can do FM MCTP testing:
  - https://github.com/moking/cxl-test-tool/tree/main
  - See (https://lore.kernel.org/linux-cxl/20250408043051.430340-1-nifan.cxl@gmail.com/) for details

- N.B. Look at the forks. May have updates for what you want to do…

Overview

1. Build QEMU
2. Build Kernel
   - Install headers, modules, and CXL test .ko's
3. Create root-filesystem image
4. Build user-space CLI tools for CXL config/management/testing
   - ndctl/cxl-cli
   - libcxlmi
   - mctp
5. Run CXL tests/develop your own work…

Step 1, Build QEMU

1. Checkout CXL QEMU fork:
   - https://gitlab.com/jic23/qemu
   - N.B stable-ish branches are dated, regularly updated
   - N.B- There are some CXL components in the upstream QEMU, but not as many. More here…

2. Build QEMU:
   - Use the README.rst, and run "./configure --help | less" for info
   - Smaller build possible with --target-list=x86_64-softmmu (for our usecase)
   - Run ../configure with –enable-debug for more debug info...

3. Export build into PATH or add to your bash profile:
   - export PATH=</path/to/qemu/build>:$PATH

Step 2, Build Kernel

1. Checkout kernel (regular upstream or CXL):
   - git clone https://git.kernel.org/pub/scm/linux/kernel/git/cxl/cxl.git
   - N.B, upstream less features, but CXL fork breaking some of my tests
2. Generate initial .config, then modify for CXL dev/test
   - make defconfig
   - Add required CXL features to .config (end of presentation)
3. Build the kernel
   - make –j<n-procs> (choose default on remaining CLI .config options)
4. Build modules for tests: need to be installed in correct order (to avoid dep errors)
   - make M=tools/testing/nvdimm
   - sudo make M=tools/testing/nvdimm modules_install INSTALL_MOD_PATH=<host-guest/exposed/kmod/dir>
   - make M=tools/testing/cxl
   - sudo make M=tools/testing/cxl modules_install INSTALL_MOD_PATH=<host-guest/exposed/kmod/dir>
   - sudo make modules_install INSTALL_MOD_PATH=<host-guest/exposed/kmod/dir>
   - N.B- The dir can be seen in the QEMU command (slide <>), exported to the guest on boot and mounted in the guest /lib/modules dir…

Step 3, Create Root Filesystem

TL;DR- run create_image.sh

1. Generate disk image:
    1. qemu-img create -f raw rootfs.img 10G
2. Setup loop device and filesystem, then mount locally
    1. sudo losetup -fP rootfs.img
    2. losetup –a
    3. sudo mkfs.ext4 /dev/loop0
    4. sudo mount /dev/loop0 /mnt
3. Install a root filesystem (in this case Debian using debootstrap)
    1. sudo debootstrap --arch=amd64 --include=openssh-server,ifupdown,net-tools,iproute2,bash bookworm $MOUNT_POINT
4. Create a password for root, setup network, sshd, and virtfs module mounting
5. Unmount and remove loopback setup
    1. sudo umount /mnt && sudo losetup -d /dev/loop0
6. Resulting rootfs.img can be run under QEMU (-drive rootfs.img).

Step 4, Build Userspace CLI Tools

- Run QEMU
  - Example topologies can be found in qemu docs ./docs/system/devices/cxl.rst
- cxl-test
  - Kernel module for testing (didn't know what other slide to put this on).
  - Tests on insertion, creates virtual CXL topology
- ndctl
  - Tool for creating CXL/DAX/PMEM regions
  - https://github.com/pmem/ndctl.git
- libcxlmi
  - CXL Management Interface Library
  - Used to construct, send and decode CCI commands
  - Easily extensible to implement new commands
  - https://github.com/computexpresslink/libcxlmi
- MCTP
  - Userspace tools for managing MCTP network from linux
  - Used with I2C or USB device emulation
  - Emulating OOB Fabric Manager from QEMU
  - https://github.com/CodeConstruct/mctp

- N.B. Again, check the forks. Some have additional functionality beyond main repos…

Conclusion

Concluding Remarks

- Lots of the CXL ecosystem is still being developed
  - Composability for CXL is still not in a mature state
    - Most work until now is on DCD/MHD.
  - CXL.cache is still not in a mature state
    - Type 2 device support still not upstreamed, currently v17 in mailing lists
    - November '24 disussions of cxl.cache https://lore.kernel.org/linux-cxl/cc2525a6-0f6a-c1c8-83e1-6396661efc8a@amd.com/
- Development is tricky
  - Can require many out-of-tree patches from different repos.
  - Many codebases, Kernel/QEMU/userspace code, all under constant development
- This is not bad news!
  - Many opportunities to engage with community.
  - Many opportunities for research/development.
  - Time is correct for this work. Industry is coalescing around CXL.

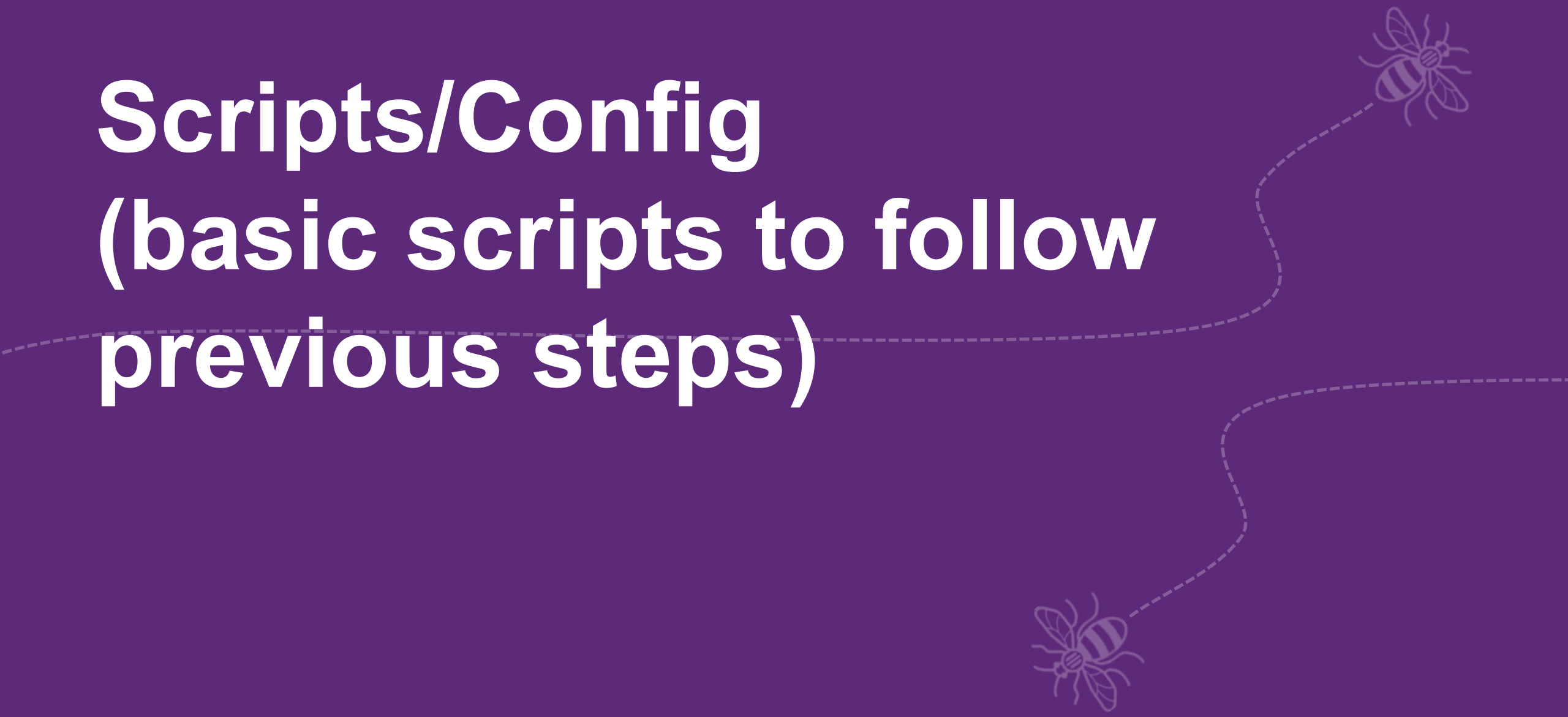
A Note About This Talk…

- CXL is too complex for 15 minutes…
- Many important concepts have been omitted or skimmed over:
  - DCD (Dynamic Capacity Devices)
  - MHD/MLD (Multi-Headed and Multi-Logical Devices)
  - G-FAM (Global Fabric-Attached Memory)
  - CCI Mailboxes and MCTP Endpoints
  - Peer-to-peer & Global Integrated Memory (Unordered IO vs .mem)
  - ACPI Tables/Firmware
  - CFMWs/Interleaving
  - …

Some Useful CXL Resources

- There is no real substitute for reading the specification, however:
- CXL Kernel and QEMU Developer Mailing List (untold wealth of information/discussions):
  - https://lore.kernel.org/linux-cxl/
- Documentation in the CXL forks of the Kernel and QEMU. (Updated/extended frequently, more than upstream documentation).
  - Search the mailing lists for new additions as well…
- Jonathan Cameron's Intro to Fabric Management emulation in CXL, and how to poke with MCTP:
  - https://gitlab.com/jic23/cxl-fmapi-tests
- Github repo with Fabric Manager information and diagrams, along with libcxlmi:
  - https://github.com/computexpresslink
- Gregory Price's Boot to bash docs (ACPI tables and early boot stuff):
  - https://github.com/gourryinverse/cxl-boot-to-bash
- Debendra Das Sharma et al. An Introduction to the Compute Express Link (CXL) Interconnect (good intro, cache trans. flows esp.)
  - https://dl.acm.org/doi/pdf/10.1145/3669900

Scripts/Config (basic scripts to follow previous steps)



Additions to Kernel .config

```
CONFIG_NVDIMM_KEYS=y
CONFIG_DEV_DAX_PMEM=m
CONFIG_DEV_DAX_HMEM=y
CONFIG_DEV_DAX_HMEM_DEVICES=y
CONFIG_DEV_DAX_KMEM=m
CONFIG_FS_DAX=y
CONFIG_FS_DAX_PMD=y
CONFIG_X86_PMEM_LEGACY_DEVICE=y
CONFIG_X86_PMEM_LEGACY=m
CONFIG_ACPI_TABLE_LIB=y
CONFIG_ACPI_HOTPLUG_MEMORY=y
CONFIG_ACPI_HMAT=y
CONFIG_TRANSPARENT_HUGEPAGE=y
CONFIG_TRANSPARENT_HUGEPAGE_ALWAYS=y
CONFIG_THP_SWAP=y
CONFIG_MM_ID=y
CONFIG_NET_DEVMEM=y
CONFIG_HOTPLUG_PCI_PCIE=y
CONFIG_PCIEAER=y
CONFIG_DYNAMIC_DEBUG=y
CONFIG_DYNAMIC_DEBUG_CORE=y
CONFIG_ND_PFN=m
CONFIG_ZONE_DEVICE=y
CONFIG_BLK_DEV_PMEM=m
CONFIG_BTT=y
CONFIG_NVDIMM_PFN=y
CONFIG_NVDIMM_DAX=y
CONFIG_EXT4_FS=m
CONFIG_XFS_FS=m
CONFIG_DAX=m
CONFIG_ENCRYPTED_KEYS=y
CONFIG_NVDIMM_SECURITY_TEST=y
CONFIG_STRICT_DEVMEM=y
CONFIG_IO_STRICT_DEVMEM=y
CONFIG_ACPI_NFIT=m
CONFIG_NFIT_SECURITY_DEBUG=y
CONFIG_MEMORY_FAILURE=y
CONFIG_MEMORY_HOTPLUG=y
CONFIG_MEMORY_HOTREMOVE=y
CONFIG_CXL_BUS=m
CONFIG_CXL_PCI=m
CONFIG_CXL_ACPI=m
CONFIG_LIBNVDIMM=m
CONFIG_CXL_PMEM=m
CONFIG_CXL_MEM=m
CONFIG_CXL_PORT=m
CONFIG_CXL_REGION=y
CONFIG_CXL_REGION_INVALIDATION_TEST=y
CONFIG_CXL_FEATURES=y
CONFIG_DEV_DAX=m
CONFIG_DEV_DAX_CXL=m
CONFIG_CXL_MEM_RAW_COMMANDS=y
CONFIG_VIRTIO_BLK=y
CONFIG_VIRTIO_PCI=y
CONFIG_BLK_DEV=y
CONFIG_EXT4_FS=y
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y
CONFIG_TMPFS=y
CONFIG_TMPFS_POSIX_ACL=y
CONFIG_PROC_FS=y
CONFIG_SYSFS=y
CONFIG_BLK_DEV_SD=y
CONFIG_SCSI=y
CONFIG_SCSI_VIRTIO=y
```

qemu-command.sh

```
KERNEL_SRC_DIR=
ROOTFS_IMAGE=
GUEST_MODDIR=
SHARED_DATA_DIR=

qemu-system-x86_64 -s -kernel $KERNEL_SRC_DIR/arch/x86_64/boot/bzImage \
-append "root=/dev/sda rw console=ttyS0,115200 loglevel=7 nokaslr
cxl_acpi.dyndbg=+fplm cxl_pci.dyndbg=+fplm cxl_core.dyndbg=+fplm cxl_mem.dyndbg=+fplm
cxl_pmem.dyndbg=+fplm cxl_port.dyndbg=+fplm cxl_region.dyndbg=+fplm
cxl_test.dyndbg=+fplm cxl_mock.dyndbg=+fplm cxl_mock_mem.dyndbg=+fplm dax.dyndbg=+fplm
dax_cxl.dyndbg=+fplm device_dax.dyndbg=+fplm pci=earlydump pci=trace" \
-smp 1 -accel kvm -serial mon:stdio -nographic -qmp tcp:localhost:4444,server,wait=off \
-netdev user,id=network0,hostfwd=tcp::2024-:22 -device e1000,netdev=network0 \
-monitor telnet:127.0.0.1:12345,server,nowait \
-drive file=$ROOTFS_IMAGE,index=0,media=disk,format=raw \
-machine q35,cxl=on -m 8G,maxmem=32G,slots=8 \
-virtfs local,path=$GUEST_MODDIR,mount_tag=modshare,security_model=none \
-virtfs local,path=$SHARED_DATA_DIR,mount_tag=datashare,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 \
-serial file:qemu.log \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G,cxl-fmw.0.interleave-granularity=8k
```

create-image.sh

```
MOUNT_POINT=
IMAGE_NAME=
DATA_SHARE_PATH=
PASSWORD=’’
HASH=$(openssl passwd -6 "$PASSWORD")

qemu-img create -f raw $IMAGE_NAME 10G
sudo losetup -fP $IMAGE_NAME
LOOPBACK_DEVICE=$(losetup -a | grep $IMAGE_NAME | awk -F: '{print $1}')
sudo mkfs.ext4 $LOOPBACK_DEVICE
sudo mount $LOOPBACK_DEVICE $MOUNT_POINT
# Minimal install to SSH in... Doesnt install user packages for ndctl tests...
#sudo debootstrap --arch=amd64 --include=openssh-server,ifupdown,net-tools,iproute2,bash bookworm $MOUNT_POINT
# setup so ndctl tests should run with no missing deps.
sudo debootstrap --arch=amd64 --include=openssh-server,ifupdown,net-tools,iproute2,bash,git,meson,build-essential,pkg-config,cmake,libkmod-dev,libudev-dev,uuid-dev,libjson-c-
dev,libtraceevent-dev,libtracefs-dev,asciidoctor,libkeyutils-dev,libiniparser-dev,keyutils,bash-completion,jq,bsdmainutils,xxd,parted,uuid-runtime bookworm $MOUNT_POINT
# Setup networking at boot on the fs.
sudo sh -c "echo 'auto enp0s2' >> \"$MOUNT_POINT/etc/network/interfaces\""
sudo sh -c "echo 'iface enp0s2 inet dhcp' >> \"$MOUNT_POINT/etc/network/interfaces\""
# apply the password to the root user on the guest
sudo chroot "$MOUNT_POINT" /bin/bash -c "usermod -p '$HASH' root"
# Make modifications to allow root to SSH in, in case QEMU config gives no terminal (i.e. output to log file).
sudo sh -c "echo 'PermitRootLogin yes' >> \"$MOUNT_POINT/etc/ssh/sshd_config\""
# Mount module sharing and data share dirs from boot
sudo sh -c "echo 'modshare /lib/modules 9p trans=virtio,version=9p2000.L,msize=262144 0 0' >> \"$MOUNT_POINT/etc/fstab\""
sudo sh -c "mkdir -p \"$MOUNT_POINT/$DATA_SHARE_PATH\""
sudo sh -c "echo 'datashare $DATA_SHARE_PATH 9p trans=virtio,version=9p2000.L,msize=262144 0 0' >> \"$MOUNT_POINT/etc/fstab\""
# cleanup before qemu can boot
sudo umount $MOUNT_POINT
sudo losetup -d $LOOPBACK_DEVICE
```