QEMU: Architecture and Internals
Lecture for the Embedded Systems Course
CSD, University of Crete (Apr. 28 & 30, 2025)

- Manolis Marazakis (maraz@ics.forth.gr)

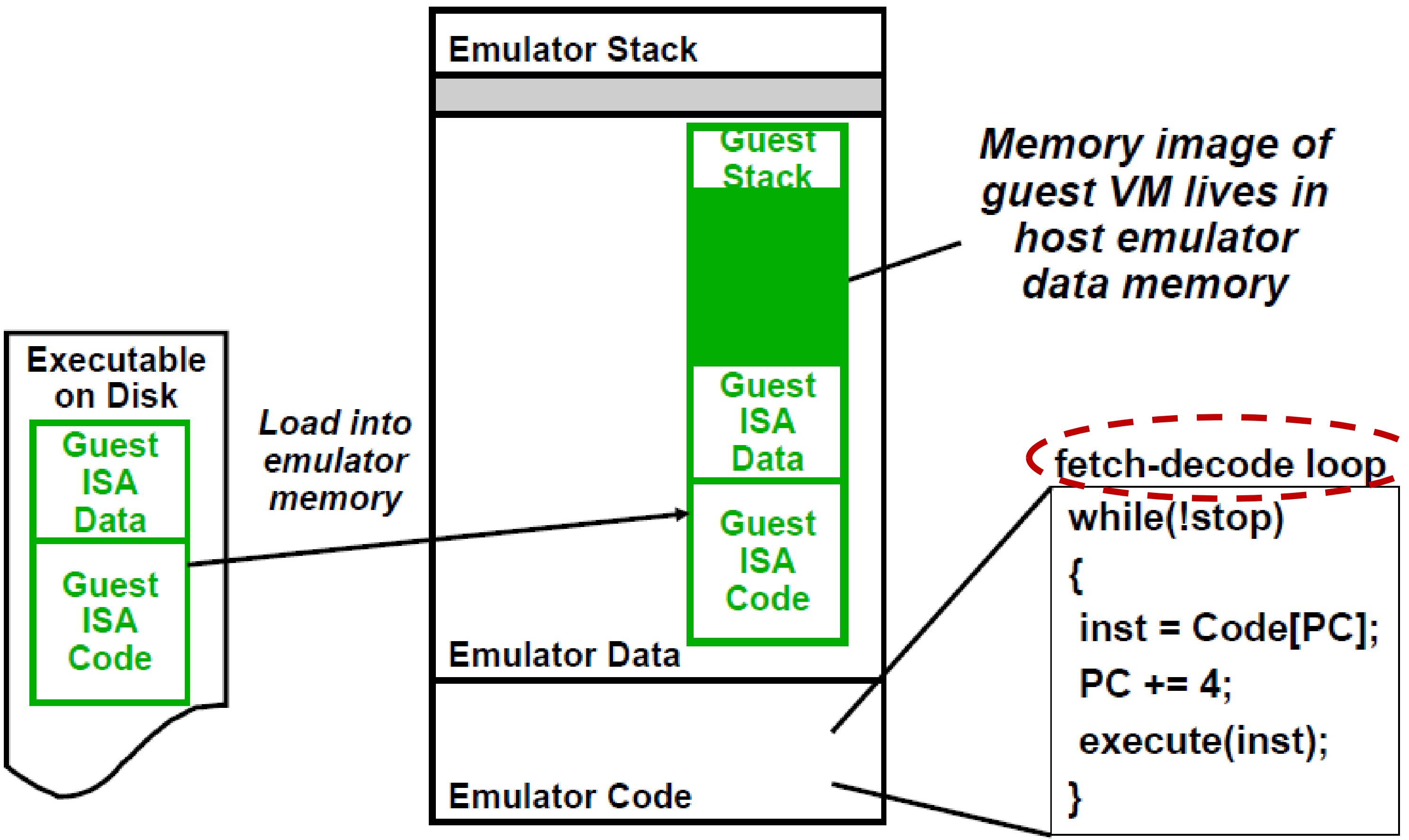
FORTH-ICS Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

System VMs (The OS implements VMs)

- VM := ISA + “Environment” (esp. I/O)
- VM specifications:
  - State available at process creation
  - ISA
  - Systems calls available (for I/O)
  - ABI: specification of the binary format used to encode programs
- At process creation, the OS reads the binary program, and creates an “environment” for it
  - … then begins to execute the code
  - … handling traps for I/O and emulation “sensitive instructions”
- Hypervisor (VMM): implements sharing of real H/W resources by multiple OS VMs

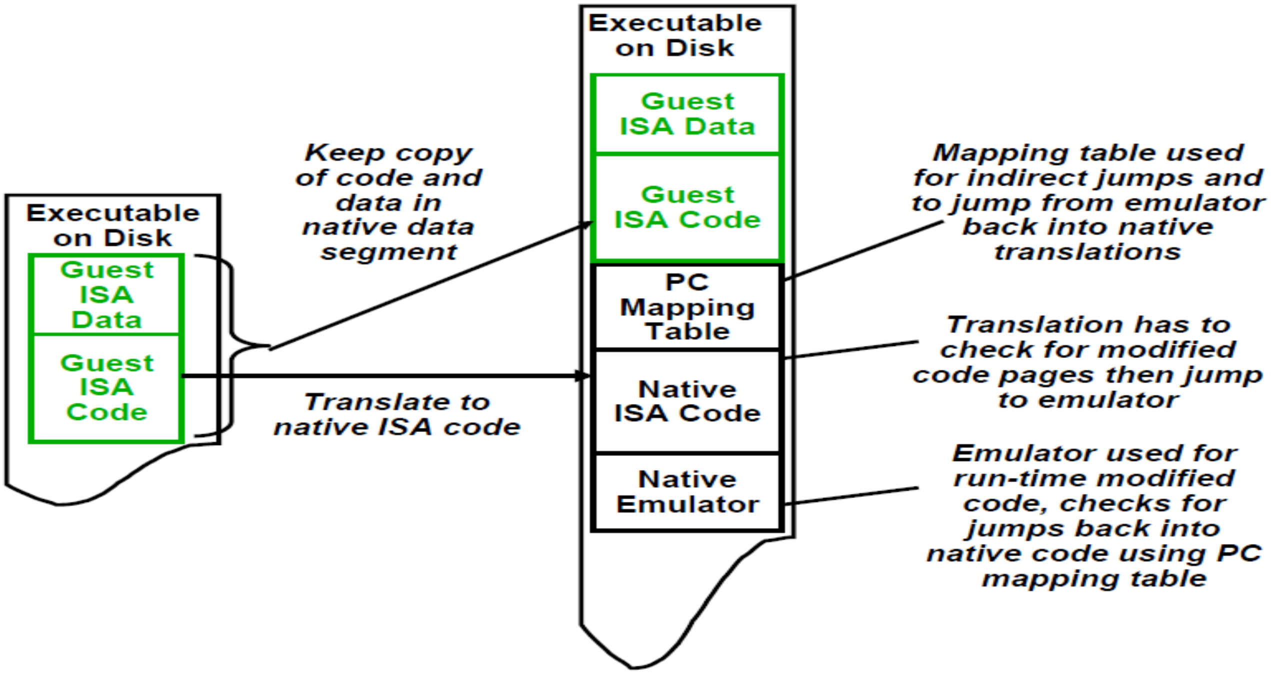
Emulation

- Interpreter fetches and decodes one instruction at a time



Static Binary Translation

- Translate <u>entire</u> binary program → create new native ISA executable
- Compiler optimizations on translated code
  - Register allocation, instruction scheduling, remove unreachable code, inline assembly …
- Complications: branch/jump targets → PC mapping table



basic block := a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching.

FORTH-ICS Institute of Computer Science

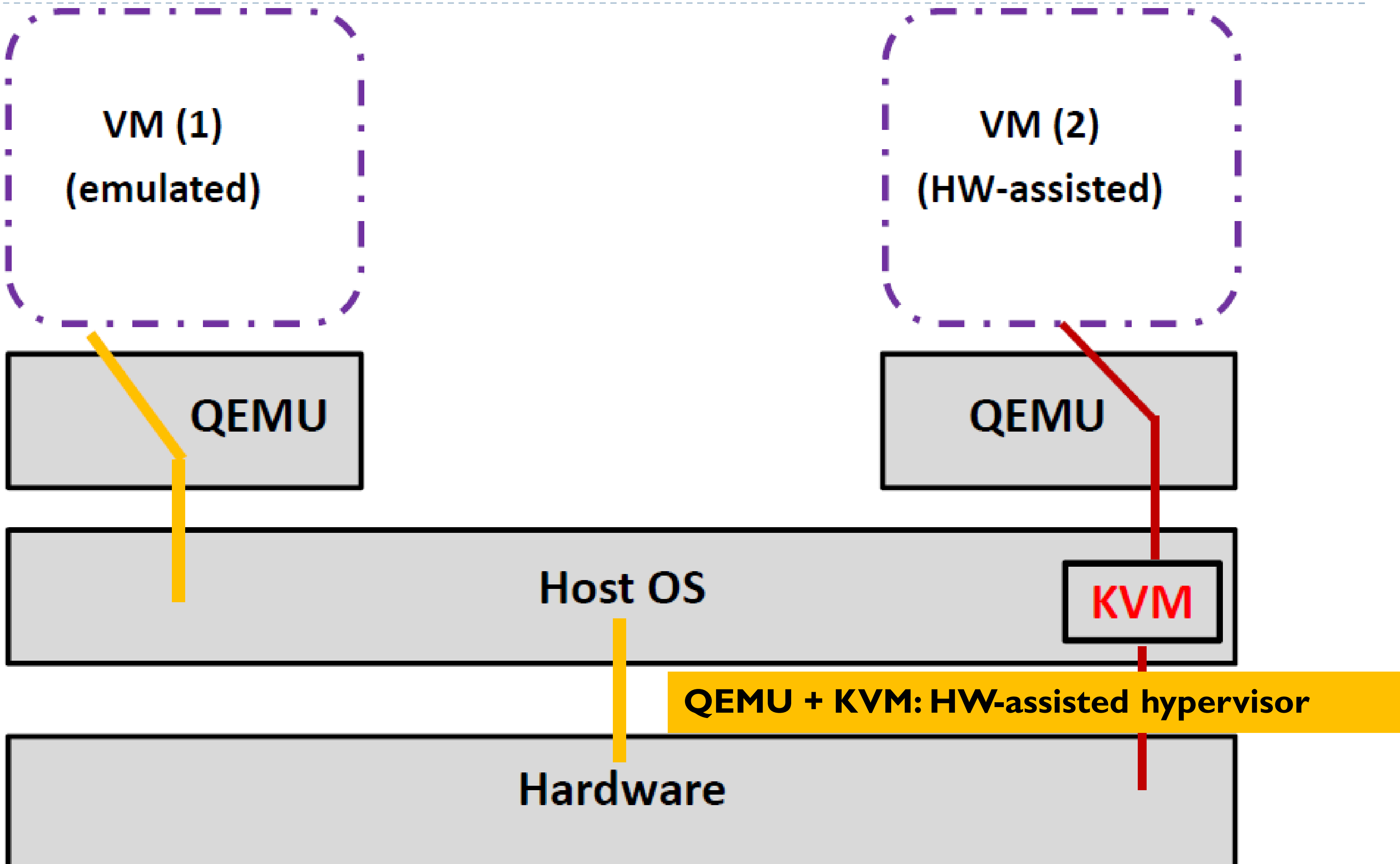
Dynamic Binary Translation

- Translate code sequences at run-time, and cache results
- Optimization based on dynamic info. (e.g. branch targets)
  - Tradeoff between optimizer run-time and time saved by optimizations in translated code
- Run-time translation and patching (chaining of blocks)
  - Use simplified Host instructions to describe Target instructions
  - Execution unit := basic code block
    - Space locality in translation cache
    - Chaining → temporal locality

Quick EMUlator (QEMU)

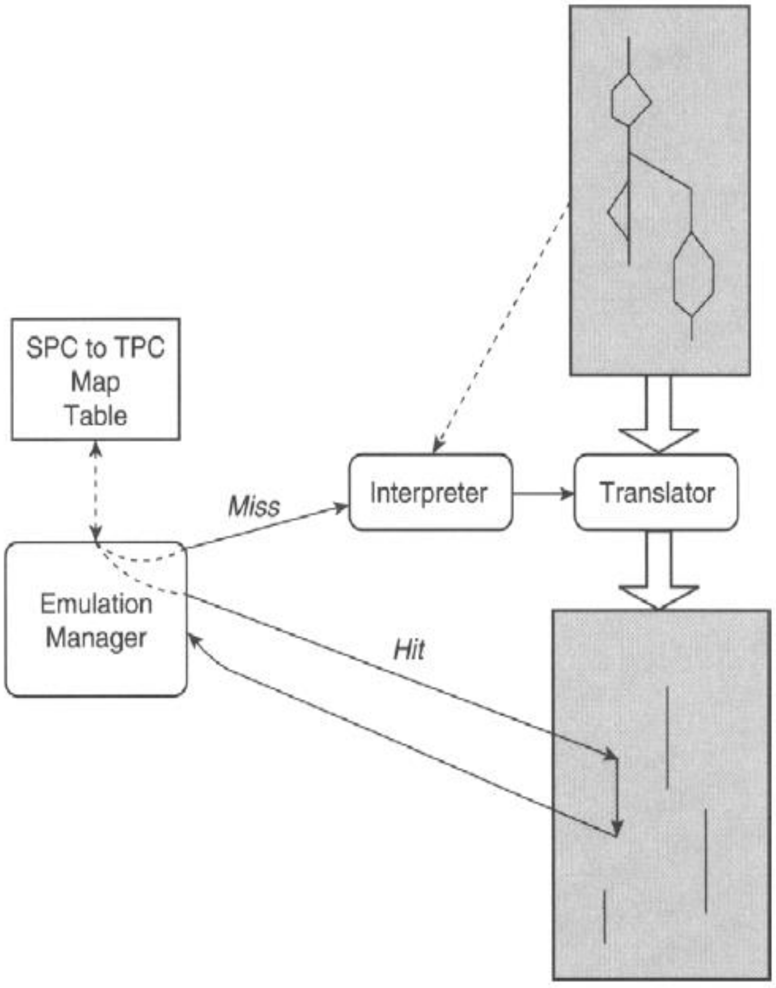
- Machine emulator + "Virtualizer" (device models)
- Modes:
  - **User-mode emulation**: allows a (Linux) process built for one CPU to be executed on another
    - QEMU as a "Process VM" for cross-compilation/cross-debugging
  - **System-mode emulation**: allows emulation of a full system, including processor and assorted peripherals
    - QEMU as a "System VM" (virtual host for system VMs)
- Popular uses:
  - Cross-compilation development environments
  - Virtualization, esp. device emulation, for xen and kvm hypervisors
  - Android Emulator (part of original SDK)
    - https://www.linaro.org/blog/running-64bit-android-l-qemu/
    - UI using emulated graphics (OpenGL), queue_pipe device for interaction with Host, misc. device models (keypad, screen, GSM, GPS, sensors)

QEMU: Emulator + Hypervisor functionality



Dynamic Binary Translation [1/3]

- **Dynamic Translation**
- First Interpret
  - … perform code discovery as a by-product
- Translate Code
  - Incrementally, as it is *discovered*
  - Place translated blocks into *Code Cache*
  - Save source to target PC mapping in an *Address Lookup Table*
- Emulation process
  - Execute translated block to end
  - Lookup next source PC in table
    - If translated, jump to target PC
    - Else interpret and translate



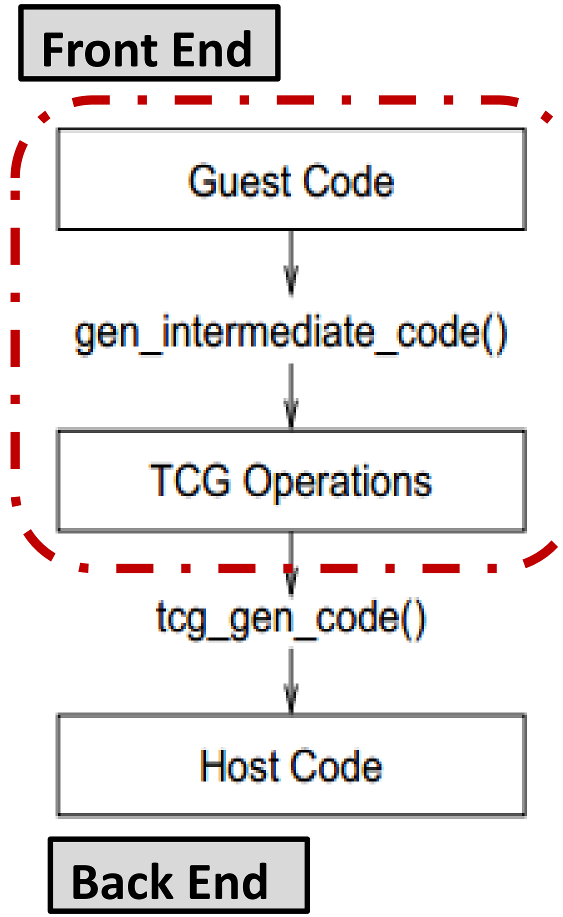
Dynamic Binary Translation [2/3]

- Works like a JIT compiler, but doesn't include an interpreter
- All guest code undergoes binary translation
  - Guest code is split into "translation blocks“
  - A translation block is similar to a basic block: the block is always executed as a whole (i.e. no jumps in the middle of a block).
- Translation blocks are translated into a single sequence of host instructions and cached into a translation cache.
  - Cached blocks are indexed using their guest virtual address (i.e. PC address value) for fast retrieval
  - Translation cache size can vary (32 MB by default)
    - Once the cache runs out of space, the whole cache is purged

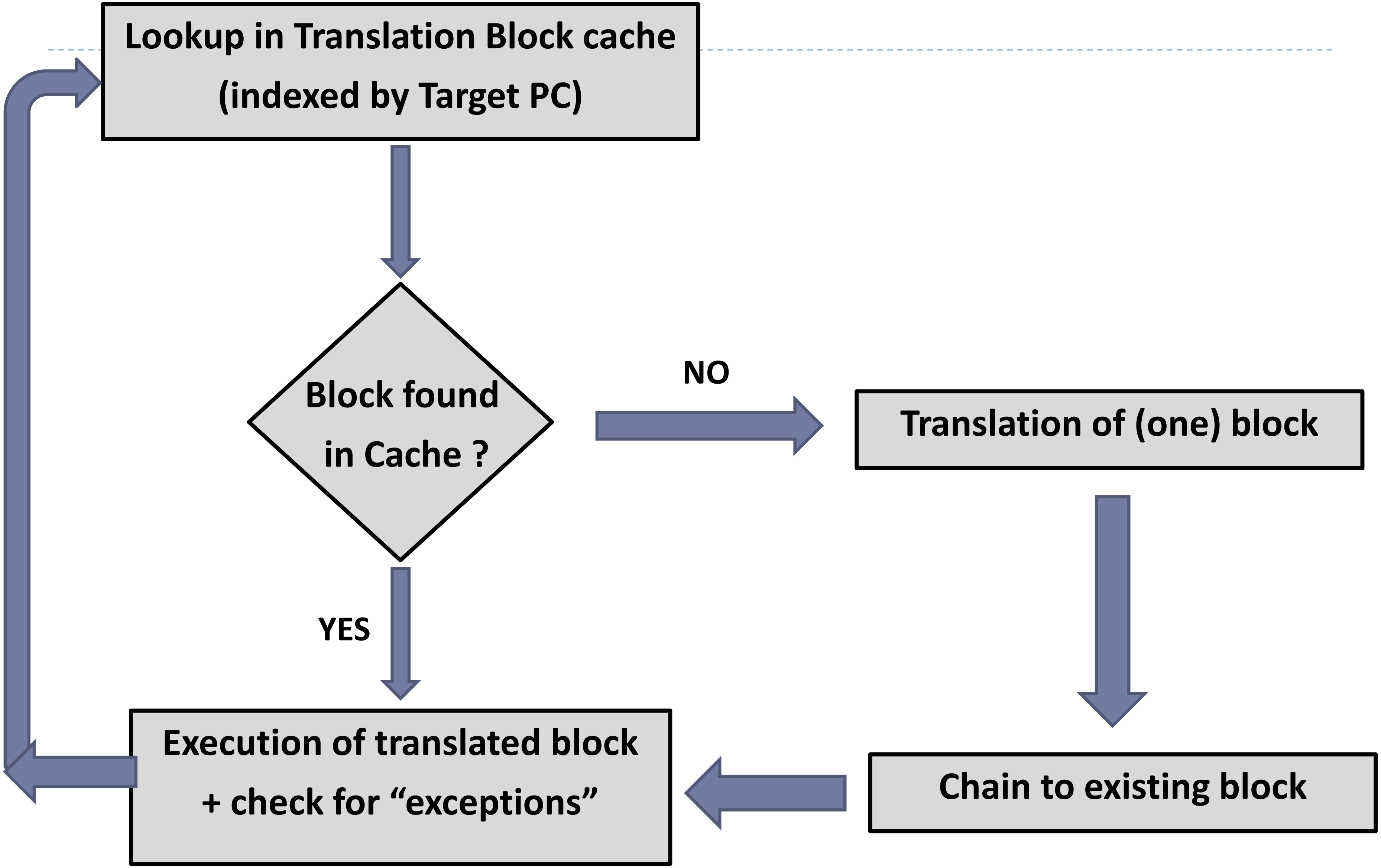
Dynamic Binary Translation [3/3]

Source Instruction Stream (binary) → Micro-operations (C code) → Object file (binary) $\xrightarrow{\text{dyngen}}$ Target Instruction Stream (binary)

- Functional simulation
  - Simulate what a processor does, not how it does it
- Dynamic binary translation
  - Interpreters execute instructions one at a time
    - Significant slowdown from constant overhead
  - Instead, QEMU converts code as needed:
    - translate basic blocks → generate native host code
    - store translated blocks in translation cache
- Tiny Code Generator (TCG)
  - Micro-operations
    - **RISC-style, CPU-independent**
    - Basic functions + Temporary variables (+ helpers)
  - (Fixed) Register mapping to reduce load/store instr's
  - Translation blocks
    - A TCG "basic block" corresponds to a list of instructions terminated by a branch instruction
  - Block chaining

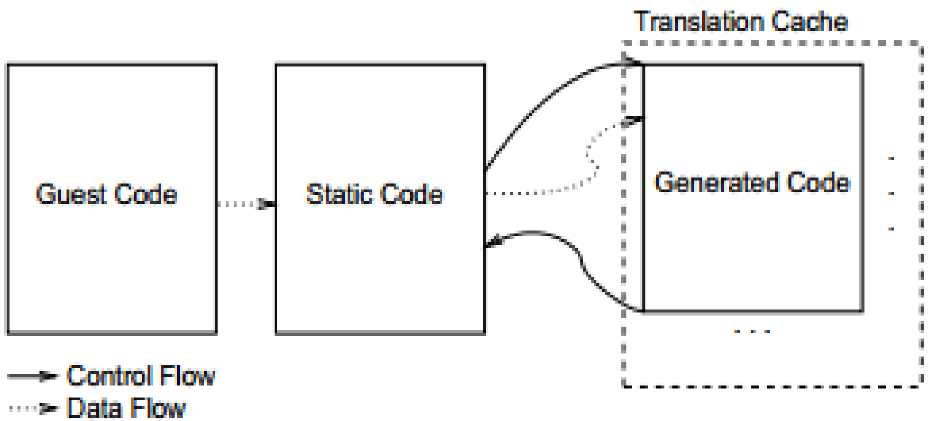


QEMU CPU Emulation Flow (Just-In-Time)

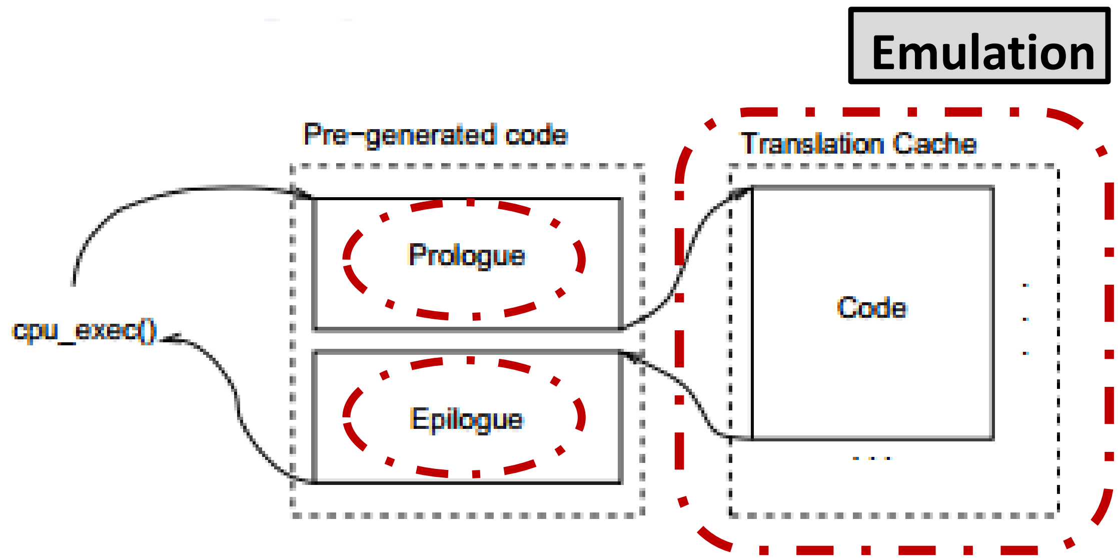


FORTH-ICS Institute of Computer Science

Dynamic translation + Cache



- cpu_exec() called in each step of main loop
- Program executes until an unchained block is encountered
  - Returns to cpu exec() through epilogue



Host

<u>Main loop:</u>

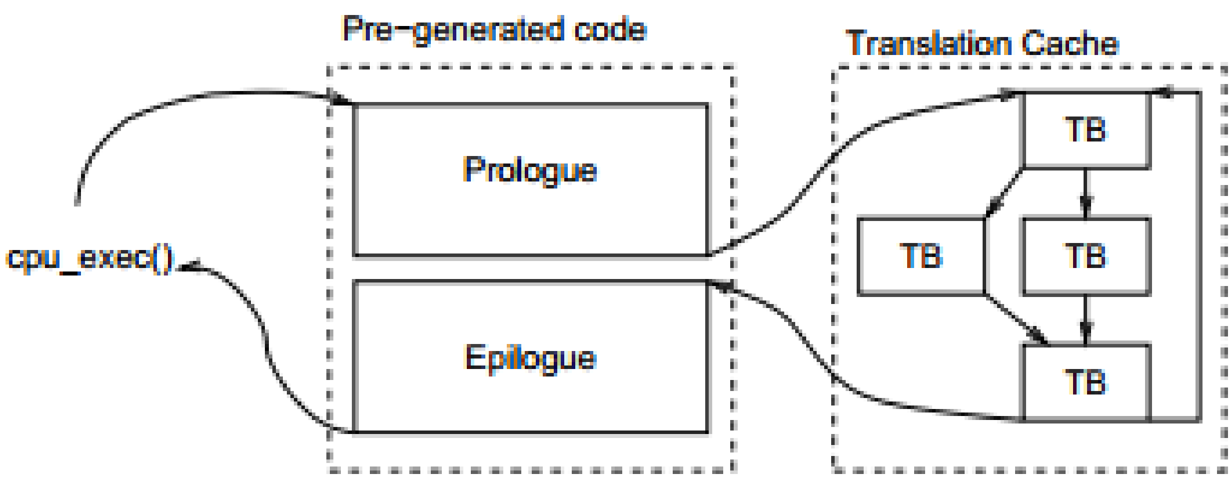
- Handling of interrupts
- Code translation
- Run guest code

Block Chaining (1/5)

- The execution of every translation block is surrounded by the execution of special code blocks
  - **Prologue:** initializes the processor for generated host code execution and jumps to the code block
  - **Epilogue:** restores normal state and returns to the main loop.
- Returning to the main loop after each block adds significant overhead … which adds up quickly
  - <u>When a block returns to the main loop and the next block is known and already translated,</u> QEMU can patch the original block to jump directly into the next block (instead of jumping to the epilogue)

Block chaining (2/5)

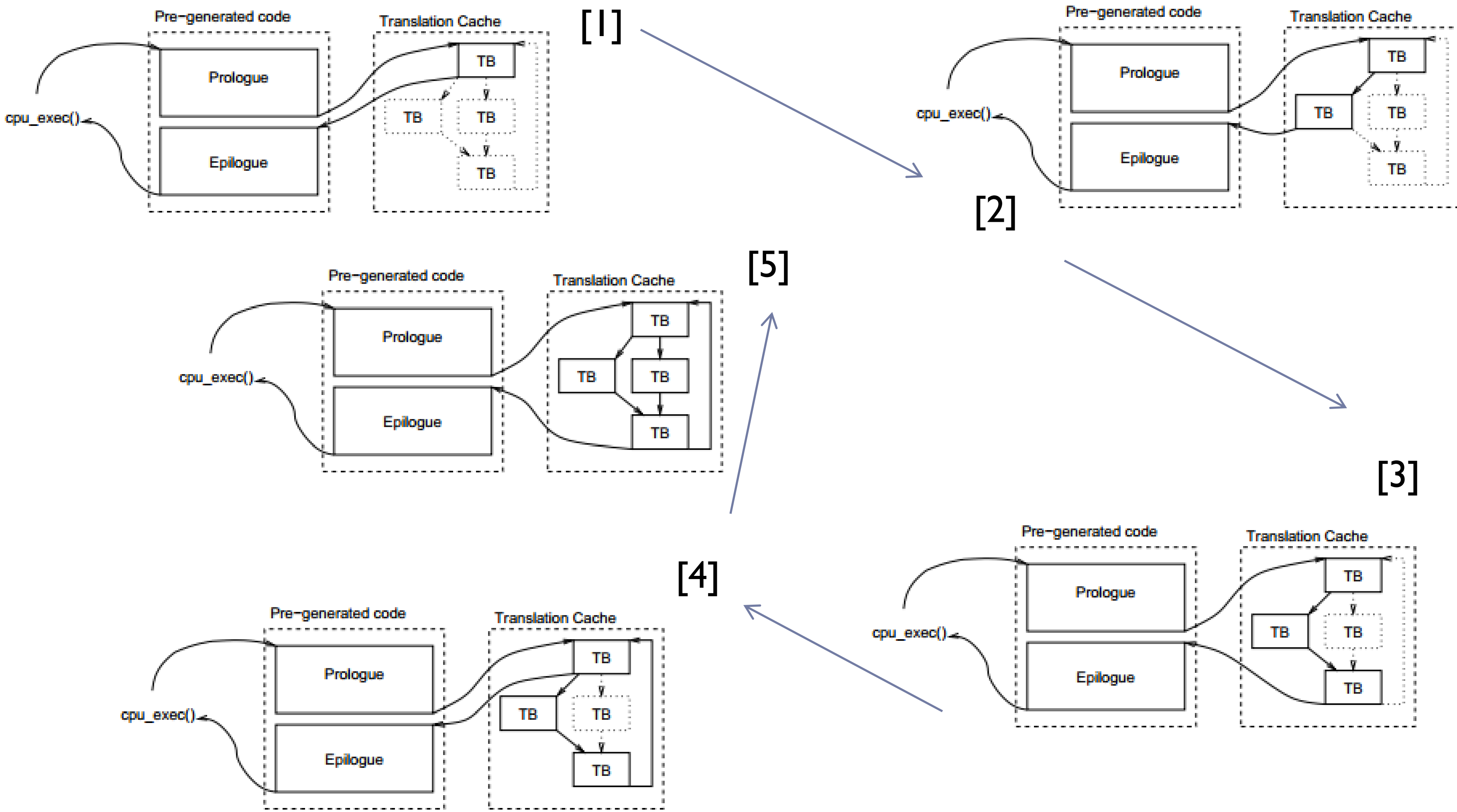
- Jump directly between basic blocks:
  - Make space for a jump, follow by a return to the epilogue.
  - Every time a block returns, try to chain it (i.e. jump directly between blocks)



Block Chaining (3/5)

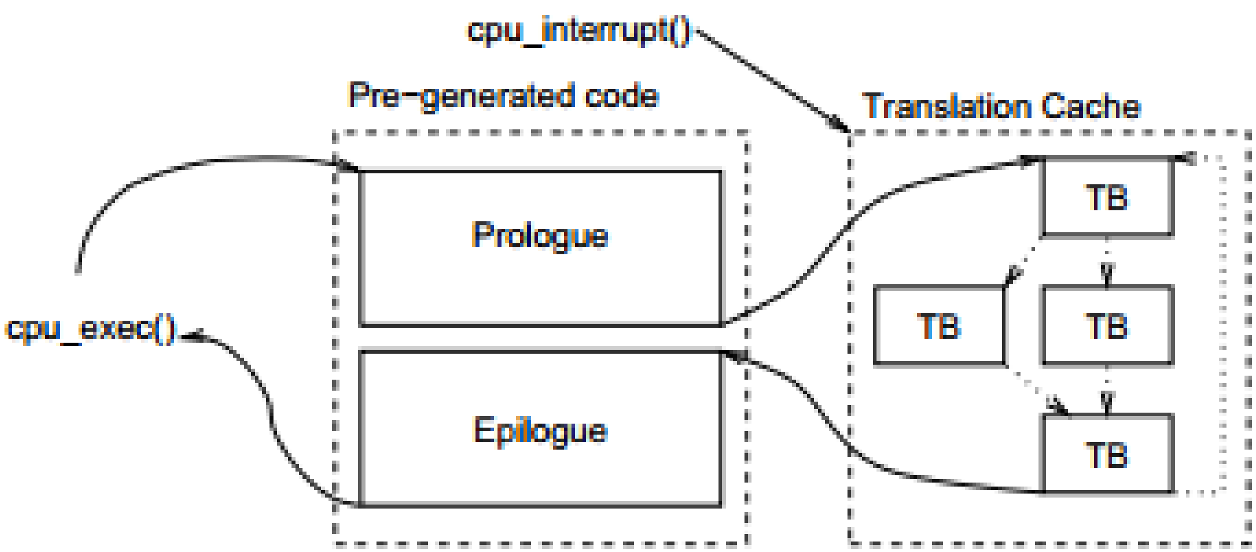
- Consecutive blocks can form chains and loops.
  - This allows QEMU to emulate tight loops without running any extra code in between.
  - In the case of a loop, this also means that the control will not return to QEMU unless an untranslated or otherwise un-chainable block is executed.
- Asynchronous interrupts:
  - QEMU does not check at every basic block if a hardware interrupt is pending. Instead, device models must asynchronously call a specific function to give notice that an interrupt is pending.
  - This function resets the chaining of the currently executing basic block → return of control to the main loop of the CPU emulator

Block chaining (4/5)

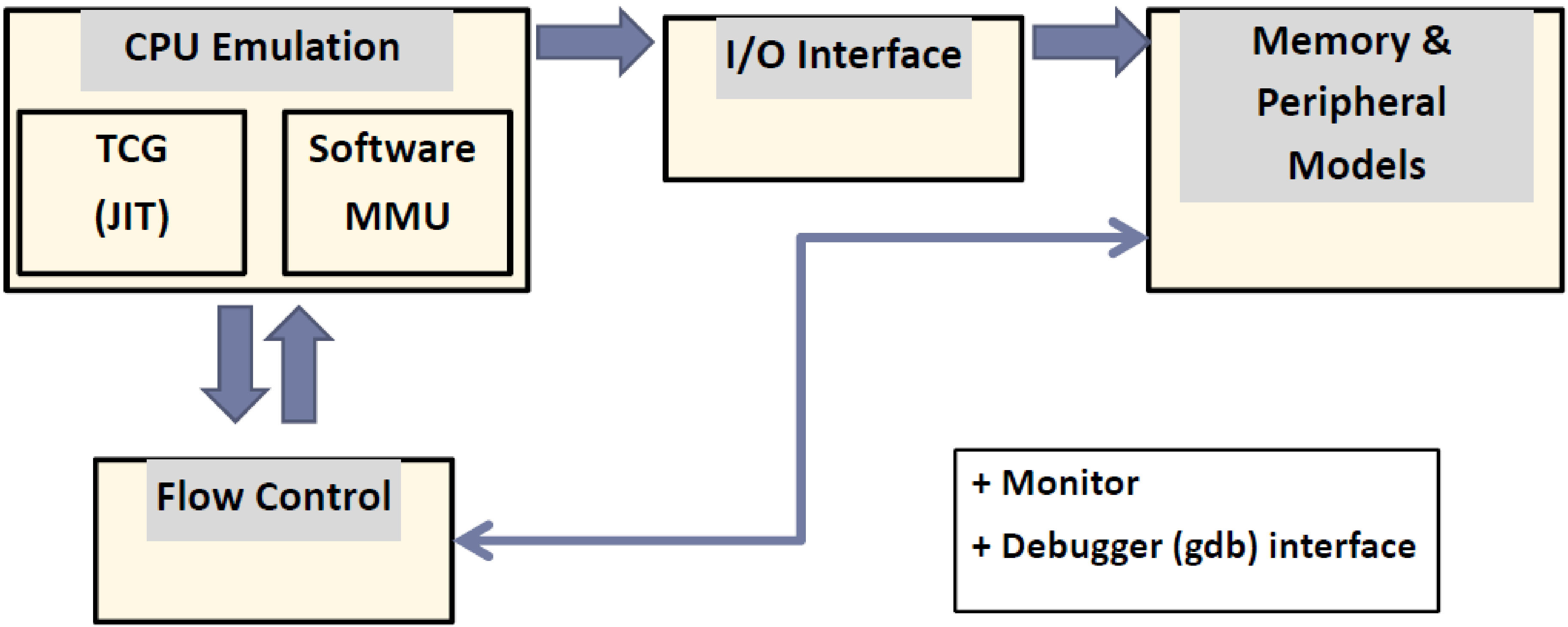


Block chaining (5/5)

- Interrupt by unchaining (from another thread)
  - Also for exceptions – e.g. I/O.



Architecture of QEMU-based Emulation



<u>QEMU "board" initialization</u>: (1) Instantiates CPU(s); (2) Allocates memories; (3) Populates memories with content.; (4) Instantiates device models & connects them; (5) Sets up default system state.

Register mapping (1/2)

- Easier if

  Number of target registers > number of source registers.
  (e.g. translating x86 binary to RISC)
- Done on a per-block, or per-trace, or per-loop, basis

  (if the number of target registers is not enough)
- Infrequently used registers (Source) may not be mapped

Register mapping (2/2)

- How to handle the Program Counter ?
  - TPC (Target PC) is different from SPC (Source PC)
  - For indirect branches, the registers hold source PCs → must provide a way to map SPCs to TPCs !
    - The translation system needs to track SPC at all times

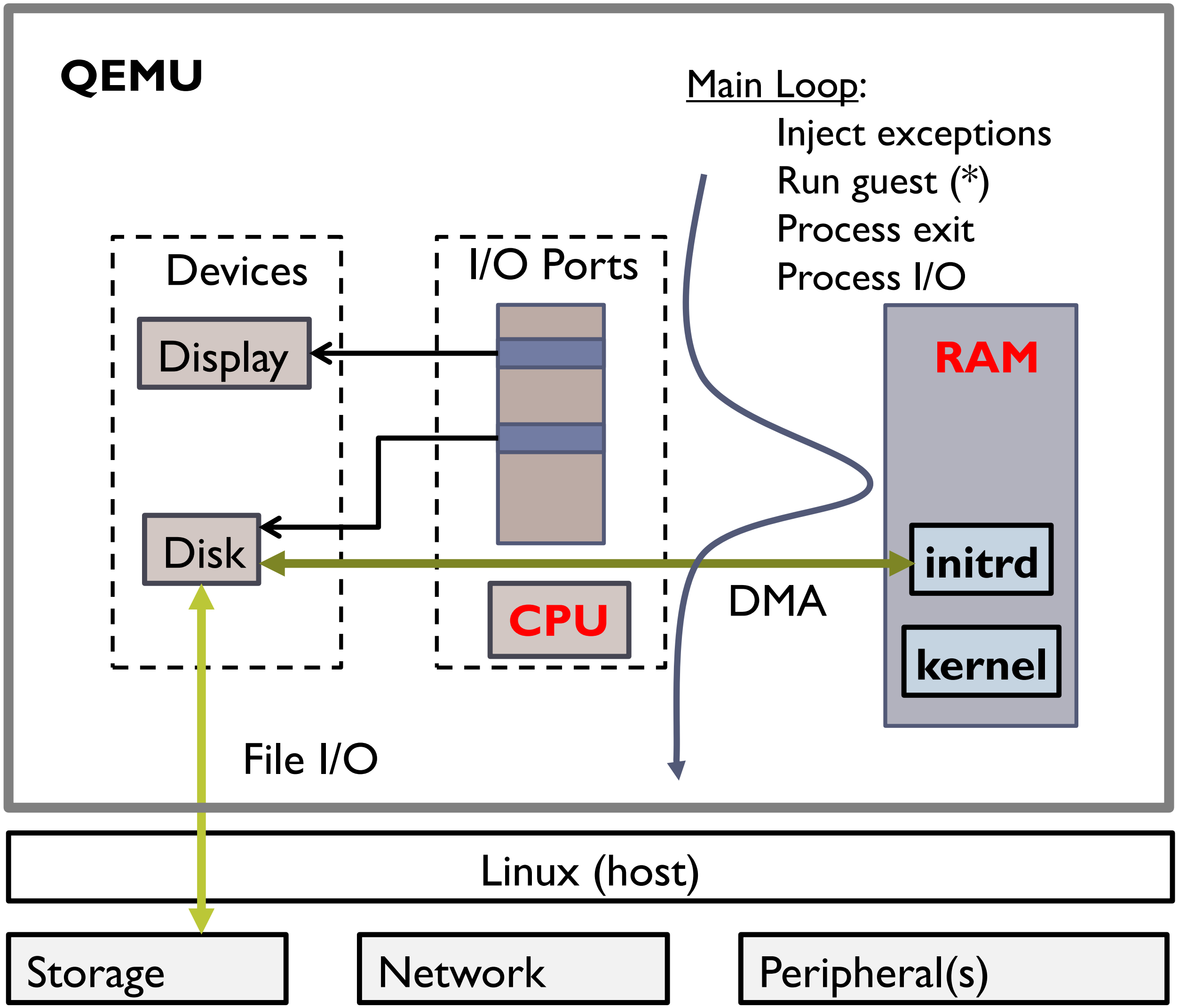
Other major QEMU components

- Memory address translation
  - Software-controlled MMU (model) to translate target virtual addresses to host virtual addresses
    - Two-level guest physical page descriptor table
  - Mapping between Guest virtual address and host virtual addresses
    - Address translation cache (tlb_table) for direct translation from target virtual address to host virtual address
  - Mapping between Guest virtual address and registered I/O functions for that device
    - Cache used for memory mapped I/O accesses (iotlb)
- Device emulation
  - i440FX host PCI bridge, Cirrus CLGD 5446 PCI VGA card , PS/2 mouse & keyboard, PCI IDE interfaces (HDD, CDROM), PCI & ISA network adapters, Serial ports, PCI UHCI USB controller & virtual USB hub, …

SoftMMU

- Virtual-to-physical address translation is done at every memory access
  - Address translation cache to speed up the translation.
  - In order to avoid flushing the cache of translated code each time the MMU mappings change, QEMU uses a physically indexed translation cache.
    - Each basic block is indexed with its (Target) physical address.
    - When MMU mappings change, only the chaining of the basic blocks is reset (i.e. a basic block can no longer jump directly to another).
    - MMU flush unlinks translation blocks.

QEMU: Overview of Linux System Emulation



(*) : binary translation

QEMU itself is single-threaded.

Overall speed of emulation depends on the number & complexity of device models.

Device emulation:

Devices registered at "board" init.

Callback upon IO range access

Device model tracks internal state

QEMU user-mode emulation example

- **arm-linux-gnueabihf-gcc** -o hello-armv7 hello-armv7.c
  - file ./ hello-armv7
  - ./hello-armv7: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.2.0, BuildID[sha1]=5d06bc699218e9d976be9b3ebb007ac6d99185df, not stripped
- qemu/arm-linux-user/**qemu-arm -L gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/libc** ./hello-armv7

QEMU system emulation example (u-boot)

make vexpress_ca9x4_defconfig CROSS_COMPILE=arm-linux-gnueabihf-

make all CROSS_COMPILE=arm-linux-gnueabihf-

qemu-system-arm -machine vexpress-a9 -nographic -no-reboot -kernel u-boot-2018.03/u-boot

……………….. …………………….. …………………….. …………………….. …………………….. …………………….. …….

U-Boot 2018.03 (Apr 30 2018 - 15:46:51 +0300)

DRAM: 128 MiB
WARNING: Caches not enabled
Flash: 256 MiB
MMC: MMC: 0
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Net: smc911x-0
Hit any key to stop autoboot: 0
⇒ reset
resetting …

Creation of root filesystem image (BusyBox)

- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make defconfig
- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make menuconfig
- --> build BusyBox as a static binary (no shared libs)
- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j4
- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make install

- <u>Creation of root filesystem image</u>:
  - dd if=/dev/zero of=bbrootfs.img bs=4k count=1024
  - mkfs.ext4 -b 4096 bbrootfs.img
  - mount -o loop bbrootfs.img ./bbrootfs
  - rsync -a busybox/_install/ ./bbrootfs
  - chown -R root:root ./bbrootfs/
  - umount ./bbrootfs

QEMU system emulation example (ARM Versatile)

- **ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-** make vexpress_defconfig
- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make menuconfig
- --> set: ARM EABI, enable: ramdisk default size=16MB, enable ext4, …
- ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j4

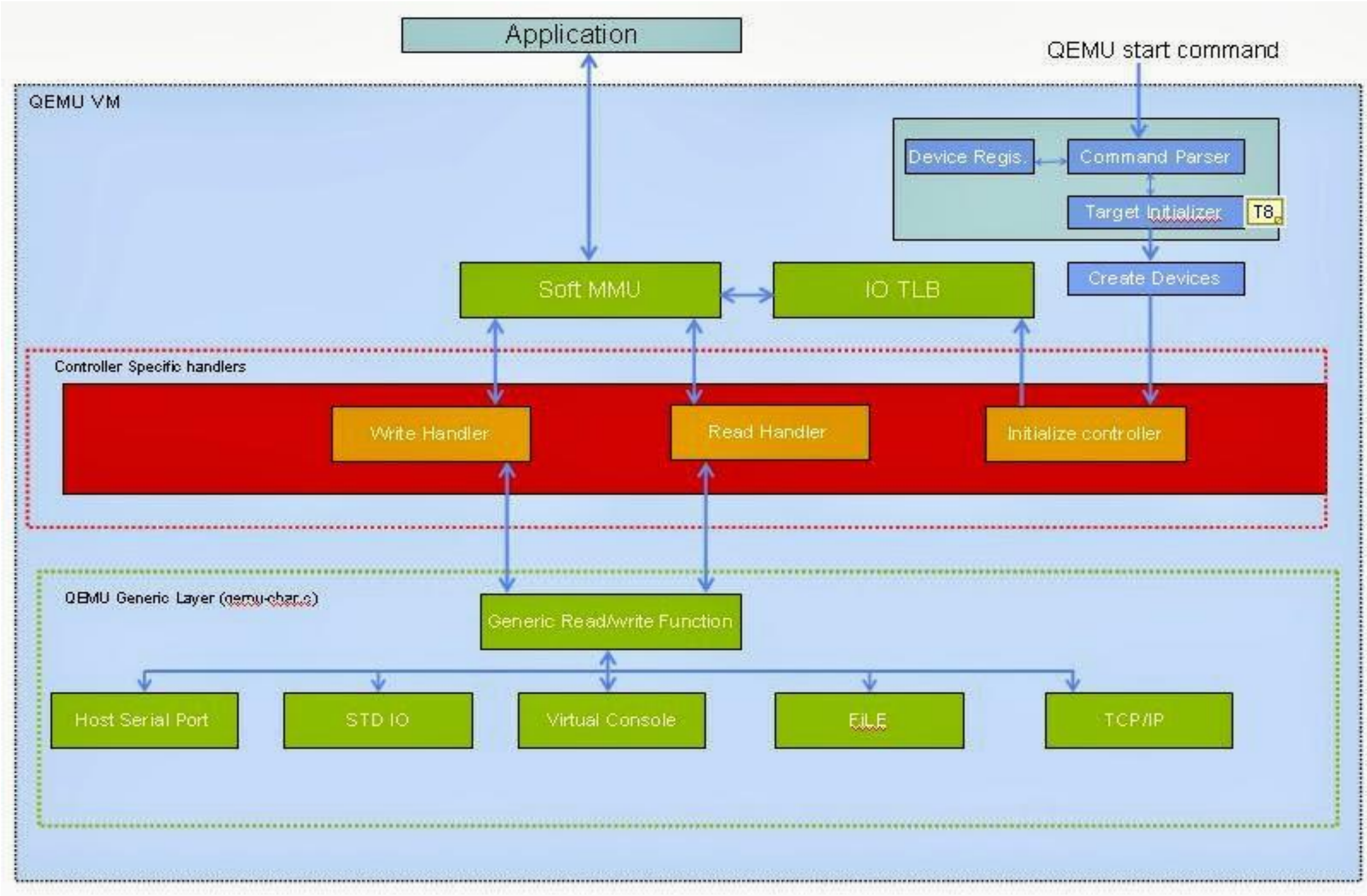
  # file linux/arch/arm/boot/zImage

  linux/arch/arm/boot/zImage: Linux kernel ARM boot executable zImage (little-endian)
- **qemu-system-arm** -M vexpress-a9 -cpu cortex-a9 -smp 4 -m 256 \
- **-dtb** ./linux/arch/arm/boot/dts/**vexpress-v2p-ca9.dtb** \
- **-kernel** ./linux/arch/arm/boot/**zImage** \
- **-append** "root=/dev/mmcblk0 rootfstype=ext4 rw rootwait earlyprintk loglevel=8 console=ttyAMA0" \
- **-drive** if=sd,driver=raw,cache=writeback,file=./**bbrootfs.img** \
- --nographic

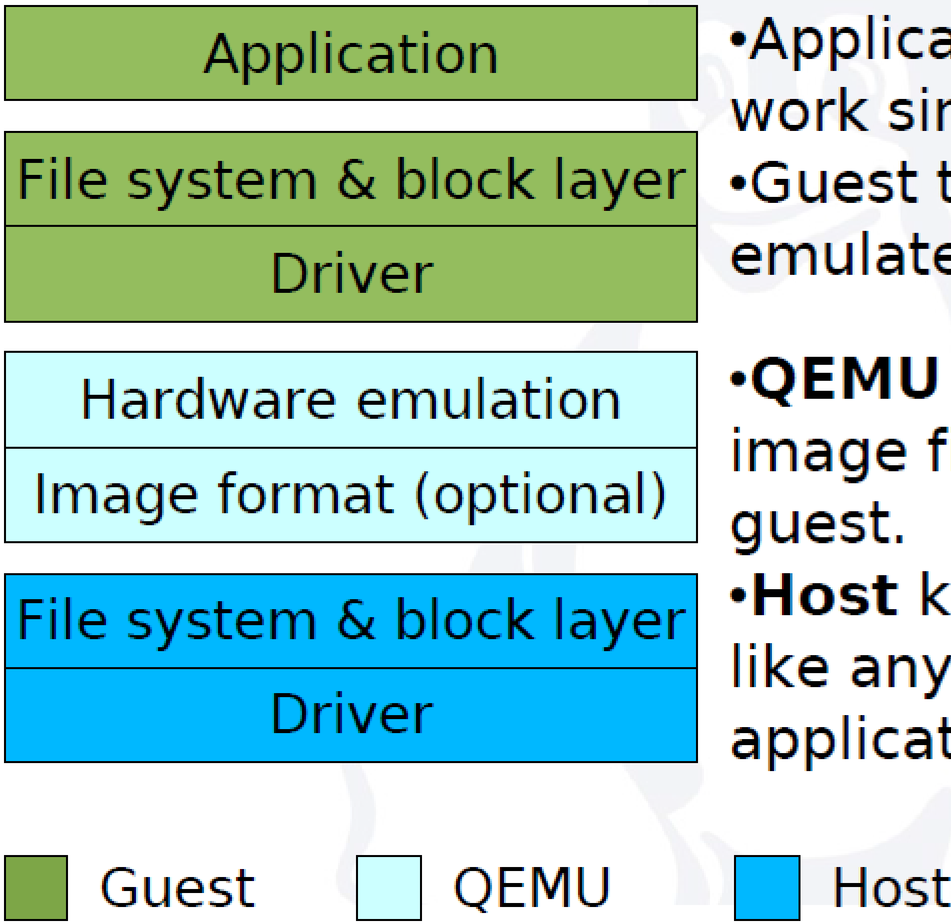
Sources

- Fabrice Bellard, **QEMU: A Fast and Portable Dynamic Translator**, USENIX Freenix 2005, http://www.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf
- Chad D. Kersey, **QEMU internals**, http://lugatgt.org/content/qemu_internals/downloads/slides.pdf
- M. Tim Jones, **System emulation with QEMU**, http://www.ibm.com/developerworks/linux/library/l-qemu/

QEMU Control Flow



QEMU Storage Stack



- Application and **guest** kernel work similar to bare metal.
- Guest talks to QEMU via emulated hardware.

- **QEMU** performs I/O to an image file on behalf of the guest.
- **Host** kernel treats guest I/O like any userspace application.

[source: Stefan Hajnoczi, - IBM Linux Technology Center, 2011]