QEMU: Architecture and Internals
Lecture for the Embedded Systems Course
CSD, University of Crete (April 30, 2015)

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

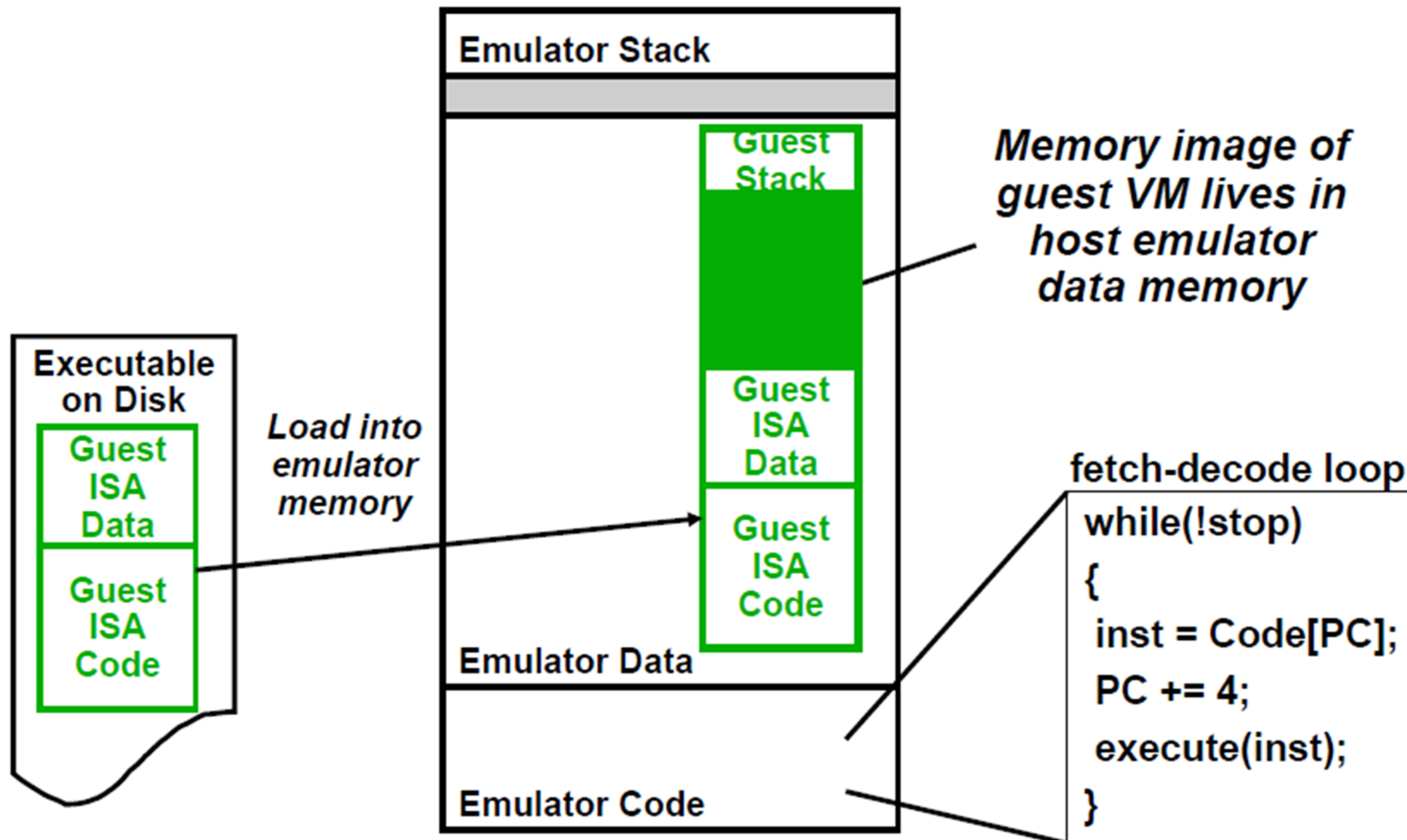
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

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

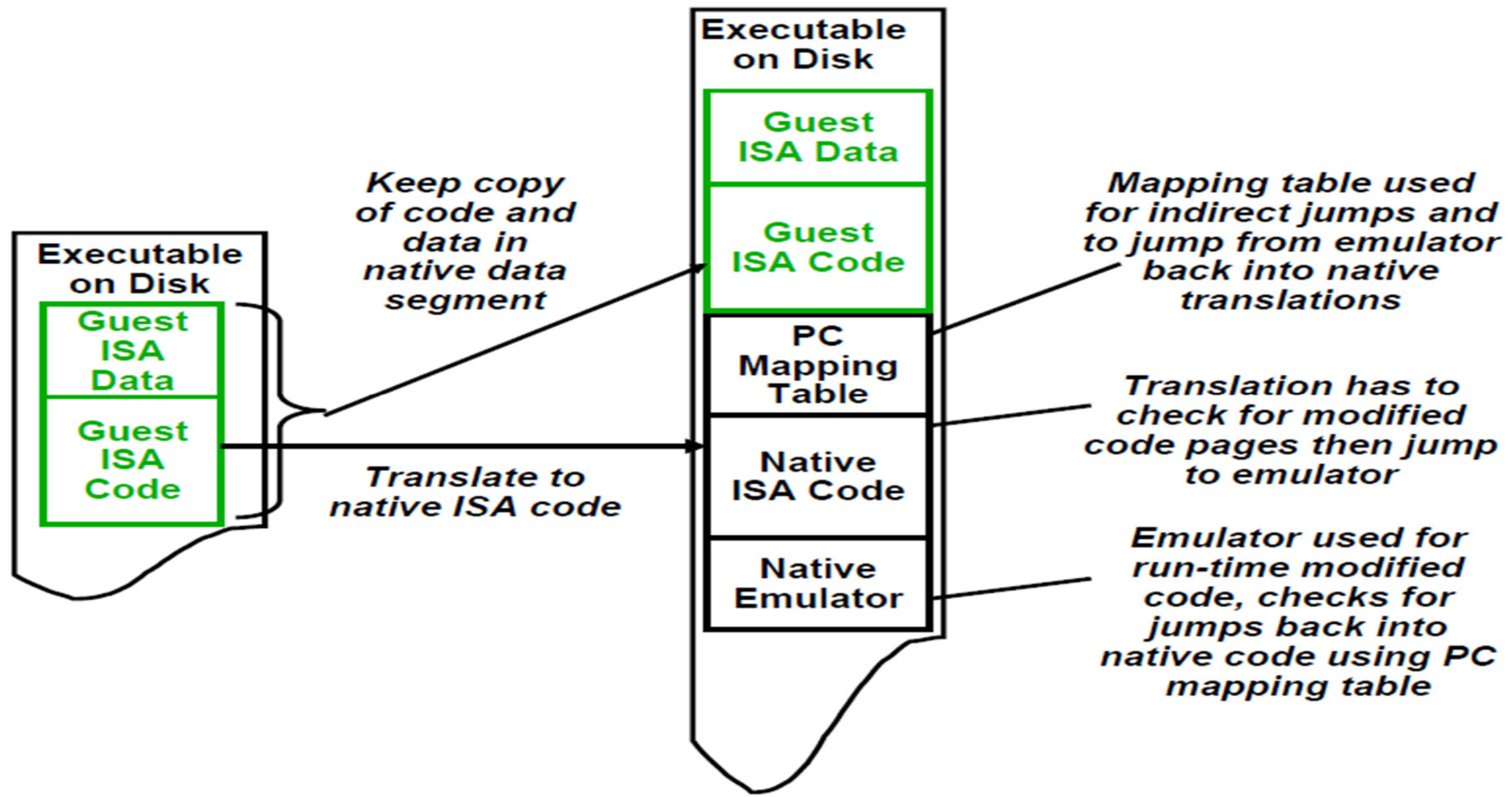
- Interpreter fetches and decodes one instruction at a time



FORTH-ICS
Institute of Computer Science

Static Binary Translation

- Translate entire 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



Dynamic Binary Translation

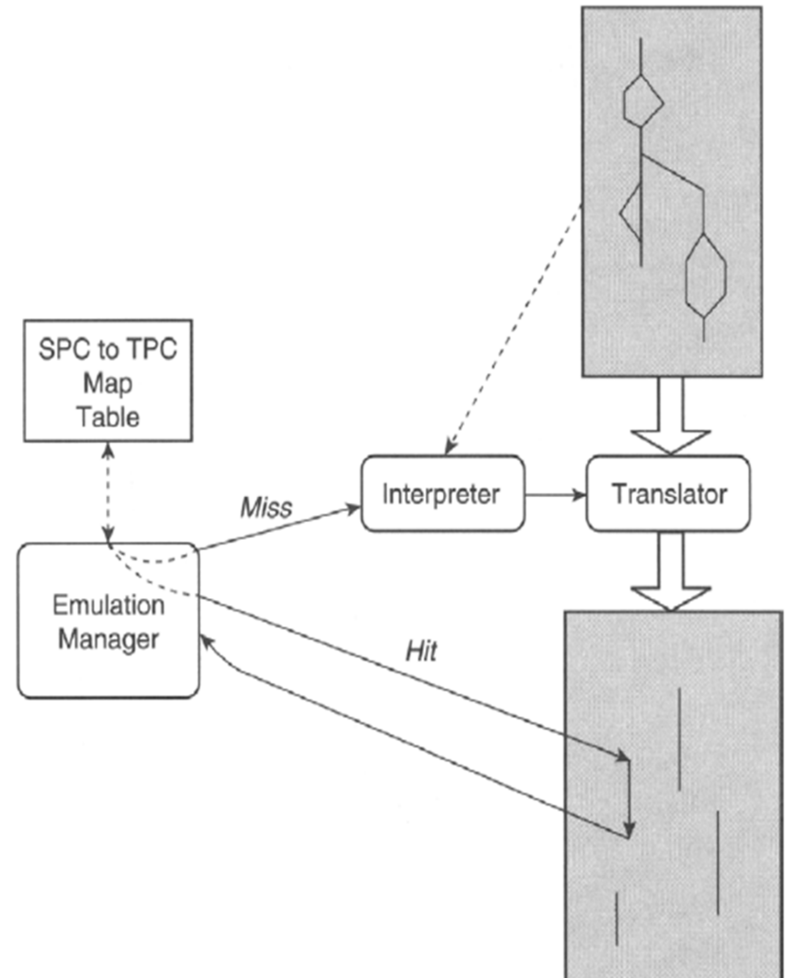
- Translate code sequences at run-time, and cache results
- Optimization based on dynamic info. (e.g. branch targets)
- Tradeoff between optimizer run-time and time saved by optimizations in translated code

Quick EMUlator (QEMU)

- Machine emulator + Virtualizer
- Modes:
  - User-mode emulation: allows a (Linux) process built for one CPU to be executed on another
    - QEMU as a “Process VM”
  - System-mode emulation: allows emulation of a full system, including processor and assorted peripherals
    - QEMU as a “System VM”
- Popular uses:
  - For cross-compilation development environments
  - Virtualization, esp. device emulation, for xen and kvm
  - Android Emulator (part of SDK)

Dynamic Binary Translation

- **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

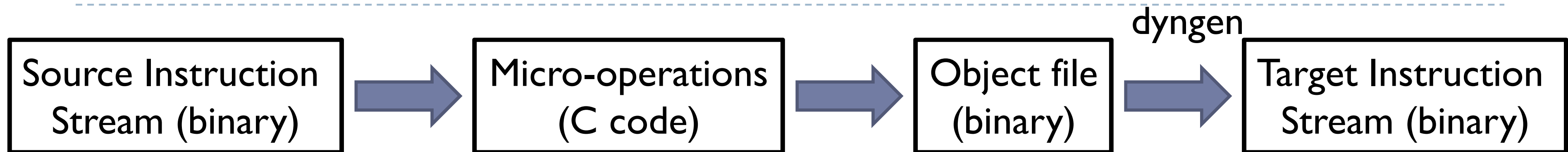


FORTH-ICS
Institute of Computer Science

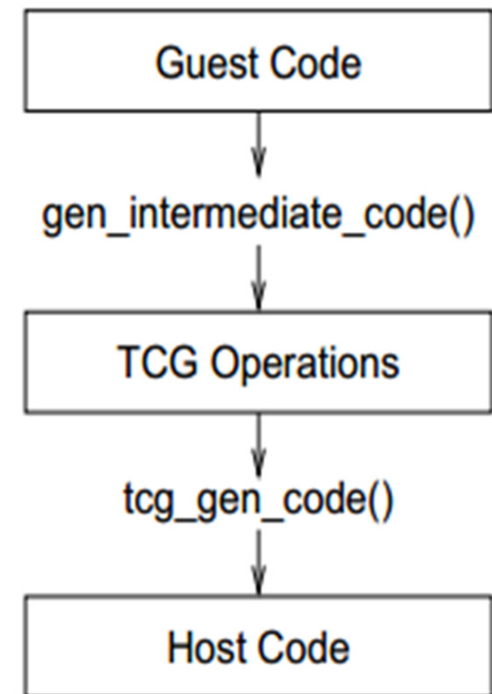
Dynamic Binary Translation (1/2)

- 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 in that the block is always executed as a whole (ie. 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 (ie. PC count), so they can be found easily.
  - Translation cache size can vary (32 MB by default)
  - Once the cache runs out of space, the whole cache is purged

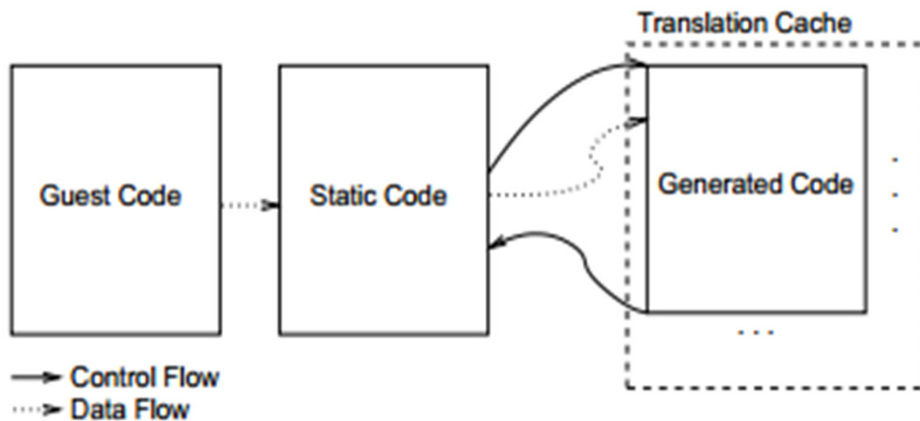
Dynamic Binary Translation (2/2)



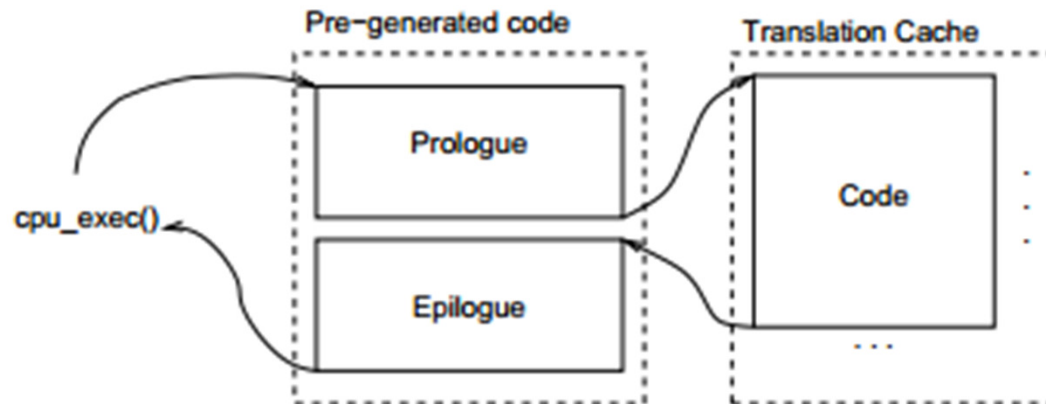
- Functional simulation
  - Simulate what a processor does, not how it does it
- Dynamic binary translation
  - Interpreters execute instructions one at a time
    - Significant slowdown from constant overhead
  - Instead, QEMU converts code as needed:
    - translate basic blocks → generate native host code
    - store translated blocks in translation cache
- Tiny Code Generator (TCG)
  - Micro-operations
  - (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



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



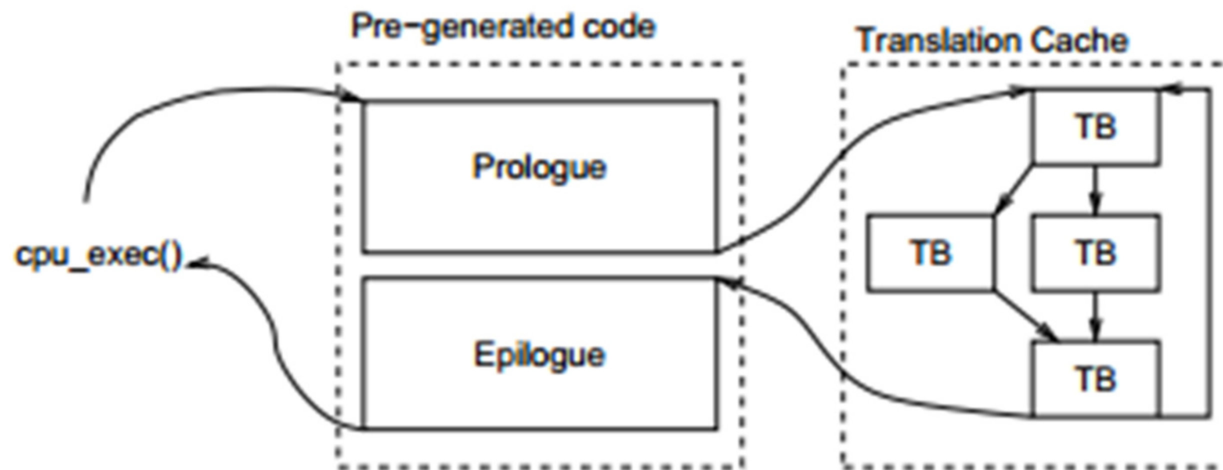
FORTH-ICS Institute of Computer Science

Block Chaining (1/5)

- Normally, the execution of every translation block is surrounded by the execution of special code blocks
  - The **prologue** initializes the processor for generated host code execution and jumps to the code block
  - The **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
  - When a block returns to the main loop and the next block is known and already translated QEMU can patch the original block to jump directly into the next block instead of jumping to the epilogue.

Block chaining (2/5)

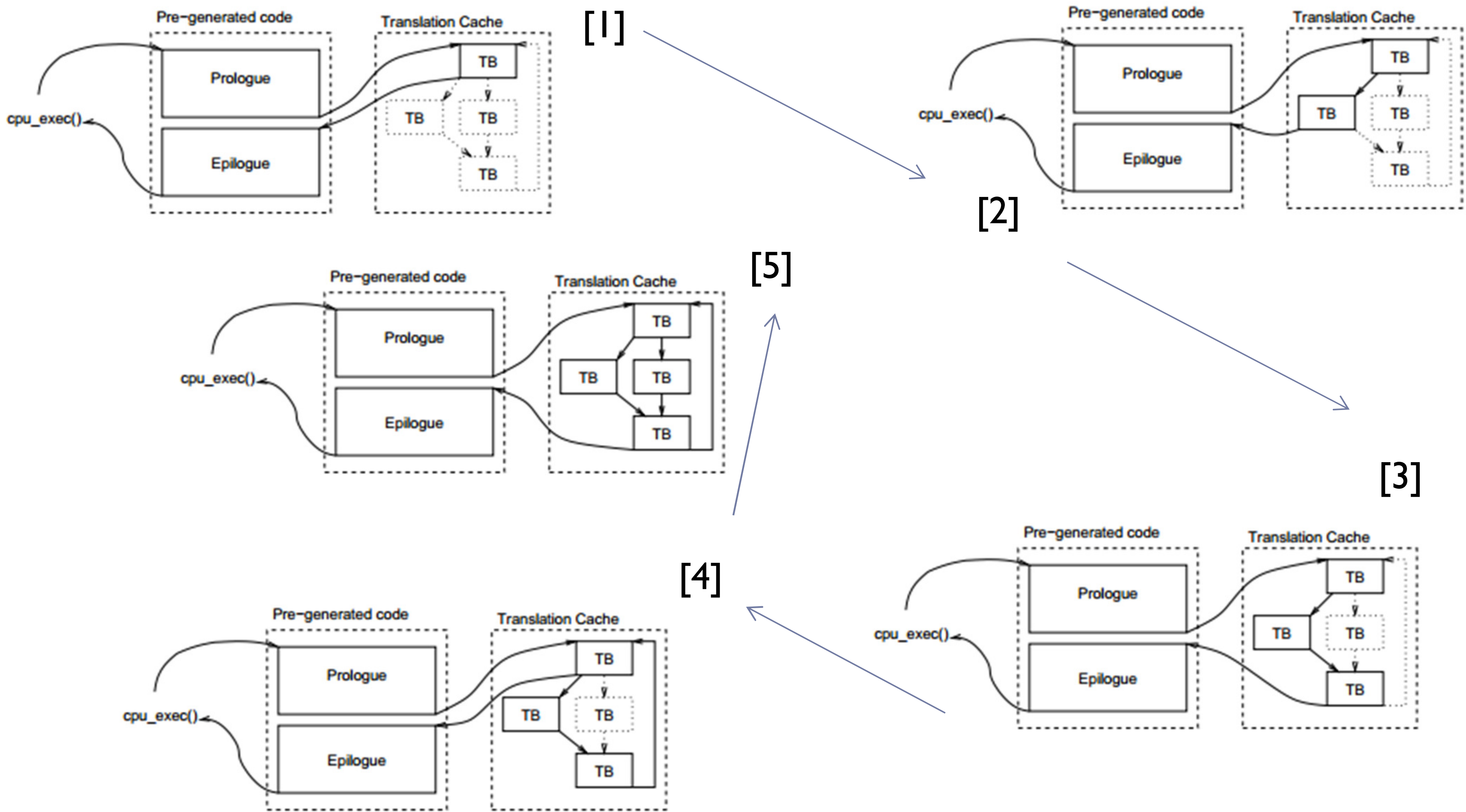
- Jump directly between basic blocks:
  - Make space for a jump, follow by a return to the epilogue.
  - Every time a block returns, try to chain it



Block Chaining (3/5)

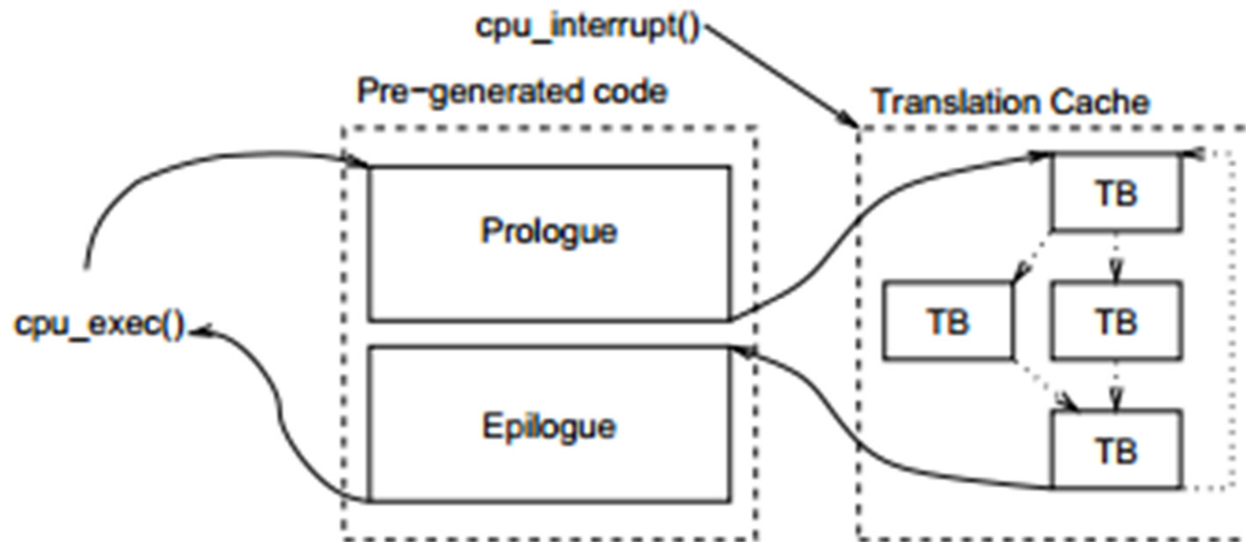
- When this is done on several consecutive blocks, the blocks will form chains and loops.
  - This allows QEMU to emulate tight loops without running any extra code in between.
  - In the case of a loop, this also means that the control will not return to QEMU unless an untranslated or otherwise un-chainable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

Block chaining (4/5)



Block chaining (5/5)

- Interrupt by unchaining (from another thread)



Register mapping (1/2)

- Easier if
  - Number of target registers > number of source registers. (e.g. translating x86 binary to RISC)
- May be 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

FORTH-ICS Institute of Computer Science

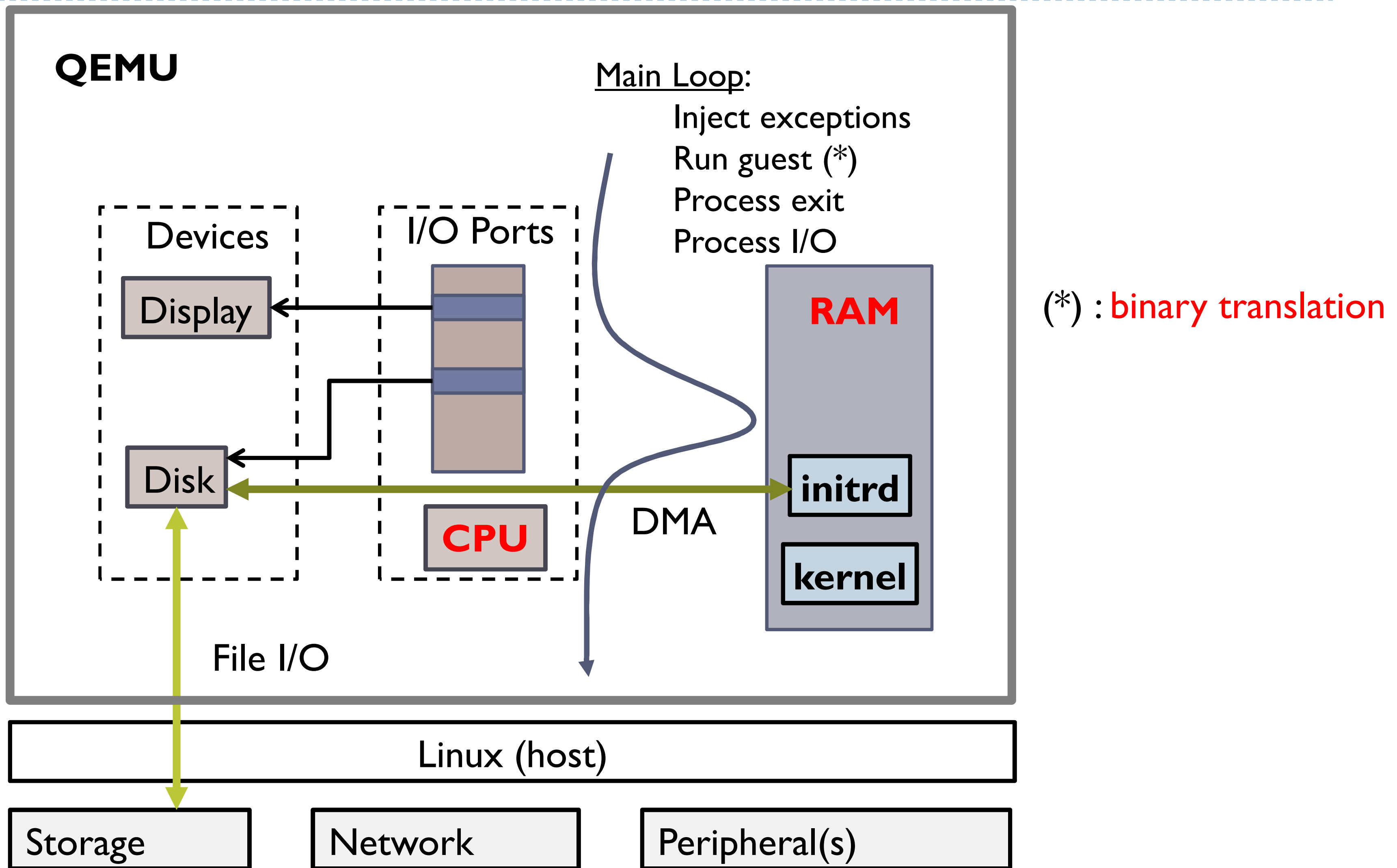
Other (major) QEMU components

- 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) that does 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

- The MMU 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 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 one).

QEMU Overview



FORTH-ICS Institute of Computer Science

QEMU Storage Stack

Application
File system & block layer
Driver
Hardware emulation
Image format (optional)
File system & block layer
Driver

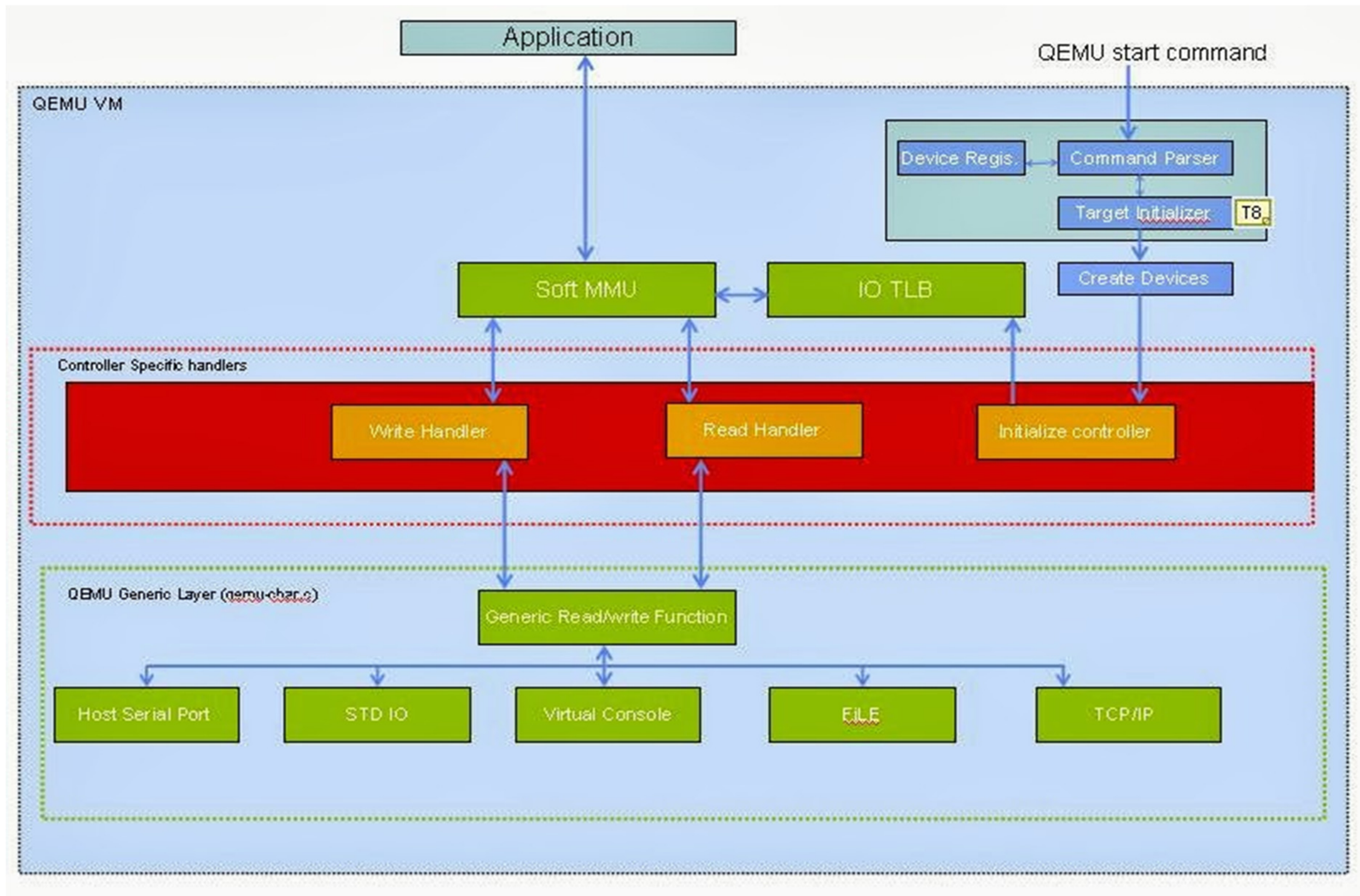
- Application and **guest** kernel work similar to bare metal.
- Guest talks to QEMU via emulated hardware.

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

Guest QEMU Host

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

QEMU I/O Control Flow



FORTH-ICS
Institute of Computer Science

QEMU user-mode emulation example

- arm-none-linux-gnueabi-gcc -o hello hello.c
  - file ./hello
  - ./hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.16, not stripped
- qemu-arm

-L Sourcery_CodeBench_Lite_for_ARM_GNU_Linux/arm-none-linux-gnueabi/libc

./hello

FORTH-ICS Institute of Computer Science

QEMU system emulation example (1/2)

- qemu-system-arm

-M versatilepb -smp 1 -m 128

-nographic -serial stdio

-kernel ../u-boot-2014.01/**u-boot.bin**

-no-reboot

-append "console=ttyAMA0 root=/dev/ram panic=5 user_debug=31"

FORTH-ICS Institute of Computer Science

Creation of root filesystem image (BusyBox)

- make **ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- defconfig**
- make **ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- menuconfig**
  - (build options -> **static**)
- make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- install

- <u>Creation of compressed root filesystem image</u>:
  - cd _install
  - find . | cpio -o --format=news > ../rootfs.img
  - cd ..
  - gzip -c rootfs.img > rootfs.img.gz

QEMU system emulation example (2/2)

- make ARCH=arm CROSS_COMPILE=arm-none-eabi- versatile_defconfig
- make ARCH=arm CROSS_COMPILE=arm-none-eabi- menuconfig
- --> set: ARM EABI, enable: ramdisk default size=16MB, enable ext4
- make ARCH=arm CROSS_COMPILE=arm-none-eabi- uImage
  - file linux-3.13.2/arch/arm/boot/uImage

linux-3.13.2/arch/arm/boot/uImage: u-boot legacy uImage, Linux-3.13.2, Linux/ARM, OS Kernel Image (Not compressed), 2072832 bytes, Thu Feb 13 16:36:27 2014, Load Address: 0x00008000, Entry Point: 0x00008000, Header CRC: 0x481719E8, Data CRC: 0x5792BBD1

- qemu-system-arm

-kernel linux-3.13.2/arch/arm/boot/uImage

-initrd bbrootfs.img.gz

-m 128 -M versatilepb

-no-reboot

-append "console=ttyAMA0 root=/dev/ram panic=5 rootfstype=ext4 rw"

-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/