Implement a virtual development platform based on QEMU

XiaoXiao Bian
Jincheng college
Nanjing University of Aeronautics and Astronautics
Nanjing, China
e-mail: bianxiaox_njnu@163.com

***Abstract*—QEMU is widely used in Cloud environments, it powers both Xen and KVM. Basically it is a fast and portable dynamic translator and an embedded machine emulator that emulates multiple CPUs and many board models. It can also provide a virtual platform for quickly software development such as Android Emulator, which uses it to emulate the whole mobile platform. But QEMU only supports common hardware, so new virtual device module should be developed for QEMU in order to emulate new hardware. In this paper, we finish our research on full system emulation, study the architecture and internals of QEMU, propose detailed steps to create user-defined virtual hardware devices and also develop the Linux kernel driver for the new device. Research results show that the whole running, testing and debugging environment are built, and user level applications can be developed for the new virtual hardware without the physical device become available.**

Keywords—QEMU; Linux kernel; embedded device emulator; user-defined virtual hardware device

I. INTRODUCTION

Traditional embedded product development, usually the first to develop hardware, software developers need to be carried out after the hardware plate molding. Since hardware development depends on component availability and vendor support, it is a very time consuming process, which seriously affect the progress of software development. With the emulators available today, it is a big advantage to develop applications for hardware without the physical devices itself. Emulator can provide a virtual platform for quickly software development such as Android Emulator [1], which uses QEMU [2] to emulate the whole mobile platform. Application developers can develop and debug code on the emulator without the real mobile phones, easily and cost saving.

There are a few emulation products. SkyEye [3] is an embedded emulator and mainly for ARM processors. Bochs [4] is a platform emulator but only focus on X86. QEMU[2] is is a fast, portable and dynamic binary translator (DBT) [6] that supports a wide range of processors (X86, ARM, PowerPC, MIPS etc.), it can also emulate the whole platform not only PC but also embedded development board. In full system emulation, QEMU is approximately 30 times faster than Bochs [5]. In this work, we choose QEMU as target emulator.

QEMU only supports common hardware, if the product uses new devices, the new virtual hardware should be developed. In this paper, architecture and internals of QEMU were studied, and list steps to add user-defined virtual hardware. We also developed Linux kernel driver for the new device.

II. QEMU

QEMU uses dynamic recompilation to emulate a guest architecture. This technique makes it possible to execute binary code provided for a certain processor on another processor which has completely different opcodes. The general structure is as follows: when executing the guest code, it is read and disassembled, then translated into equivalent opcodes of the host processor. Finally, the dynamically generated host code is executed. However, since QEMU emulates many guest architectures on many host architectures, an intermediate phase has been added to the dynamic recompilation. Instead of translating the guest code directly to the host code, it is translated into an intermediate bytecode, and then translated into host code. The advantage of going through an intermediate bytecode is to facilitate the support of new emulated architectures and also that of new host architectures. If the translation was made directly from the code of the emulated architecture to the code of the host architecture, it would be necessary for each emulated architecture to write a new code translator for each specific host. If X emulated architectures are supported on Y host architectures, X * Y specific translators should be written. QEMU uses intermediate bytecode to save the effort.

Sequences of instructions are translated from the target to the host instruction by dynamic binary translator [7]. QEMU divides the target binary code into chunks of code called basic blocks (BB). A base block is a code portion that has a single entry point and a single exit point. This means that it does not contain any code that is the target of a jump instruction and that only the last statement can start executing another base block. As a result, when the first instruction of a base block is executed, all the other instructions of this basic block will be executed once, respecting their order. Indeed, it would be useless to continue translating guest code when it is not known if it will be executed.

Then a group of micro opcodes will be generated by tiny code generator (TCG) front-end operations after decoding the BB. Then TCG back-end operations hard encode the micro opcodes and generate host executable instructions called translated block (TB) and cache them in TB hash table. Each time the target code instruction pointer changes, TB hash table will be scanned whether contains the already generated TB, if it exists, the TB will be directly executed without generated second time for speed up the execution. Fig. 1 describes this process completely.

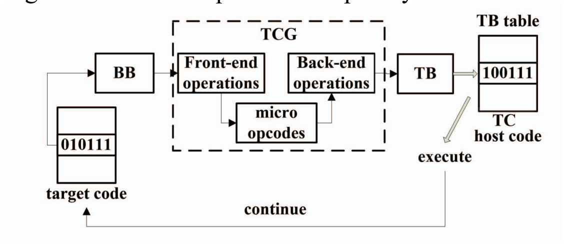


Figure 1. Dynamic binary translation.

QEMU supports user mode emulation and full system emulation. User mode emulation allows running a Linux process compiled for one target CPU on another CPU. Full system emulation allows a complete and unmodified operating system run in a virtual machine, including processors and supporting peripherals. In this paper, our research was finished on full system emulation.

III. Operating Mechanism In QEMU

A. *Dynamic Binary Translation*

We focus on creating user-defined virtual hardware devices for QEMU. Take reading peripheral registers as an example. Fig. 2 shows the dynamic translation and calling the relevant I/O function when the target architecture code read external devices register in QEMU. When QEMU is started, function device_init will be called to initialize a peripheral, register read function and bind to a port number. Then cpu_exec() is called to execute guest instructions. The tb_find_fast() and tb_find_slow() is called first to check if guest instructions are in translation cache(TC). If not, gen_intermediate_code_internal() is called to translate a BB to intermediate code. The tcg_gen_code() continues to translate intermediate code to host code or a TB. The TB is executed by tcg_QEMU_tb_exec(). If BB is translated and stored in TC, tcg_QEMU_tb_exec() executes it without translating.

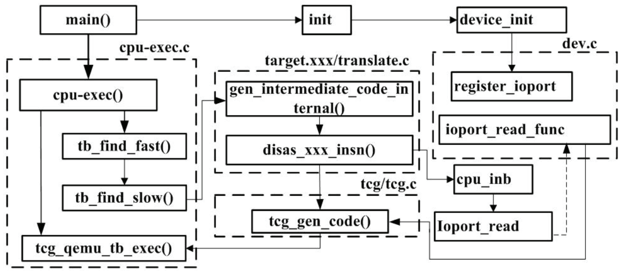
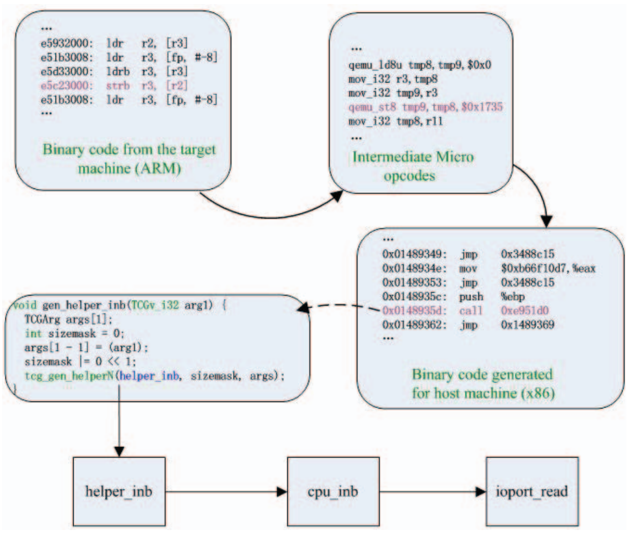


Figure 2. Read peripheral I/O register in QEMU.

When object code reads the I/O register, QEMU finds the function ioport_read_func() which registers the port number and add it to dynamically generated code. Then tcg_QEMU_tb_exec() is called to execute dynamically generated code. At this point we can see target instruction ' in al, addr' is executed by peripheral function ioport_read _func.

B. *Helper function*

An important part of DBT in QEMU is helper functions. QEMU uses them to implement uncommon but complex guest instructions [8], such as read or write from an I/O memory region, memory-mapped I/O method is used in this case. At start time, all virtual devices initialize by registering their port which was mapped to memory region, and also providing read/write callback functions, when tcg detects target code trying to do I/O operation, it inserts helper function in TB, helper function decides to call which processing function according to the port number, so complex blocks of code do not have to be implemented entirely at compile run-time. From these helper functions, callbacks that have been registered by the user's program are called.



```
...
e5932000:  ldr   r2,  [r3]
e51b3008:  ldr   r3,  [fp,  #-8]
e5d33000:  ldrb  r3,  [r3]
e5c23000:  strb  r3,  [r2]
e51b3008:  ldr   r3,  [fp,  #-8]
...
```

Binary code from the target machine (ARM)

```
...
qemu_ld8u tmp8, tmp9, $0x0
mov_i32 r3, tmp8
mov_i32 tmp9, r3
qemu_st8 tmp9, tmp8, $0x1735
mov_i32 tmp8, r11
...
```

Intermediate Micro opcodes

```
...
0x01489349:  jmp    0x3488c15
0x0148934e:  mov    $0xb66f10d7, %eax
0x01489353:  jmp    0x3488c15
0x0148935c:  push   %ebp
0x0148935d:  call   0xe951d0
0x01489362:  jmp    0x1489369
...
```

Binary code generated for host machine (x86)

```
void gen_helper_inb(TCGv_i32 arg1) {
  TCGArg args[1];
  int sizemask = 0;
  args[1 - 1] = (arg1);
  sizemask |= 0 << 1;
  tcg_gen_helperN(helper_inb, sizemask, args);
}
```

Figure 3. Helper functions of read I/O register in QEMU.

Fig. 3 presents an example about target code (ARM) tries to read peripheral register, tcg first translate them into intermediate micro opcodes, opcode 'QEMU_st8 tmp9, tmp8, $0x1735' means load the value from peripheral register, this is a special opcode, tcg inserts the helper function gen_helper_inb() into generated host code. In this way, target code is compiled into native function calls, at last, the peripheral device's handler ioport_read() is called to return the register value.

IV. Create User- Defined Device For QEMU

In this paper, our goal is to create user-defined virtual hardware device for QEMU, and also develop Linux driver for the device. We describe the following detail steps to create a user-defined PCI device which supports interrupts, DMA, MMIO and PIO regions.

The first step is to register our device inside QEMU, macro 'type_init' with a parameter is the initialization function of a device, this macro is executed before main function. As we want to implement a PCI peripheral device, so extend the PCIDevice struct as Fig. 4 shows:

```
static const TypeInfo pci_new_device_info = {
        .name              = TYPE_PCI_NEW_DEV,
        .parent            = TYPE_PCI_DEVICE,
        .instance_size   = sizeof(PCINewDevState),
        .class_init        = pci_newdev_class_init,
};
```

Figure 4. C code of extend PCIDevice to create new device.

The parent and instance_size fields are for inheritance purpose. The important thing here is the class_init function which will be called after the device is registered, so that we can overwrite the virtual methods of the PCIDeviceClass with our device and set different PCI configuration bytes. When the whole virtual system is powered on, Bios and kernel will access PCI configuration space and get to know how to write commands in order to communicate with it. The macro invokes 'pci_new_register_types' which will register the type 'TYPE_PCI_NEW_DEV'. Function 'pci_ newdev_class _init' is the class static initialization. It sets up the static variables of the class and passes PCI specific information (e.g. vendor id and device id).

Before running QEMU, it's necessary to provide our device type by adding "-device TYPE_PCI_NEW_DEV" parameter, then QEMU loads 'pci_new_device_info', executes function 'pci_new_register_types' which registers our new device type 'TYPE_PCI_NEW_DEV' and calls the associated initialization function 'pci_newdev_class_init', sets up the PCI specific information so that Linux kernel can read, get the device id, then load our driver to manage it. The detail user level case is shown in Fig. 5.

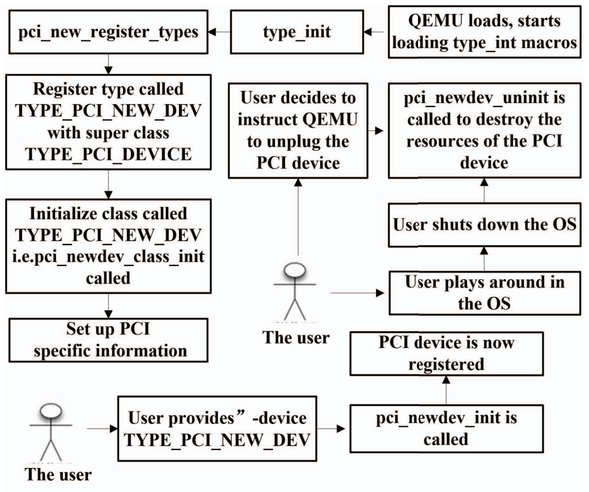

pci_new_register_types
type_init
QEMU loads, starts loading type_int macros
Register type called TYPE_PCI_NEW_DEV with super class TYPE_PCI_DEVICE
User decides to instruct QEMU to unplug the PCI device
pci_newdev_uninit is called to destroy the resources of the PCI device
User shuts down the OS
Initialize class called TYPE_PCI_NEW_DEV i.e.pci_newdev_class_init called
User plays around in the OS
Set up PCI specific information
The user
PCI device is now registered
User provides" -device TYPE_PCI_NEW_DEV
pci_newdev_init is called
The user

Figure 5. UML diagram of device creation.

Here, we create a new file named /hw/new_device.c, add class initialize function pci_newdev_class_init(), point the actually initialize device, eject device and reset device handler, also set the value of vendor_id and device_id for kernel to identify the new device. Function pci_new_dev will be executed when the new device is plugged and pci_new_dev_uninit is called when it's unplugged. For demonstration purposes, we list some code here to provide the detail, as shown in Fig. 6.

```
void pci_newdev_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
    k->init = pci_newdev_init;
    k->exit = pci_newdev_uninit;
    /* this pci id of our device */
    k->vendor_id = 0x1337;
    k->device_id = 0x0001;
    dc->desc = "New Virtual PCI Device";
    /* other user things for qemu */
    dc->props = newdev_properties;
    dc->reset = qdev_pci_newdev_reset;
    ......
}
```

(a) Snipped of C code used in new device initialization.

```
static int pci_newdev_init(PCIDevice *pci_dev)
{
     /* init the internal state of the device */
     PCINewDevState *d = PCI_NEW_DEV(pci_dev);
     ......
     memory_region_init_io(&d->mmio, OBJECT(d), &new_mmio_ops, d,
"new_mmio", NEW_MMIO_SIZE);
     memory_region_init_io(&d->io,     OBJECT(d),     &new_io_ops,     d,
"new_io", NEW_IO_SIZE);
     pci_register_bar(pci_dev, 0, PCI_BASE_ADDRESS_SPACE_IO, &d-
>io);
     pci_conf = pci_dev->config;
     ......
}
```

(b) Snipped of C code used in registering new PCI device and mapping memory region.

```
static const MemoryRegionOps new_mmio_ops = {
      .read = pci_new_dev_read,
      .write = pci_new_dev_write,
      .endianness = DEVICE_NATIVE_ENDIAN,
      ......
};
```

(c) Snipped of C code used in registering registering read and write call back handlers.

Figure 6. Use case creating and initialing a new PCI device

When QEMU tries to create the new device, we first cast the device to PCINewDevState at the beginning, initialize the internal state of the PCI device and use buffer pci_conf to represent the configuration space of the device.

QEMU uses MemoryRegion struct to represent MMIO and PIO regions, these regions can be hooked, so we call api memory_region_init_io() to register read and write handler here such as pci_new_dev_read() and pci_new_dev_write(). Before these regions to be accessible from outside of the device, another necessary step is to register it in a PCI bar.

A PCI device can use up to six address regions and each region can be either memory or Input/Output locations. A bar can be registered for these regions.

After all these have been done, if the CPU instruction tries to read or write these regions, our callback functions will be called and the data can be read or stored in the virtual PCI device.

The whole initialization process is a bit complex, so we list these function call path here for a clear view as shown in Fig. 7.

```
type_init(newdev_register_types)
-> type_register_static(&newdev_info)
-> newdev_class_init()
-> pci_newdev_init()
-> memory_region_init_io(&pci_newdev_ops)
```

Figure 7. Function call path of initialization process

When target code wants to access the new device, tcg inserts address_space_write() for write operation and helper _ret_ldub_mmu() with parameter function pointer address _space_read for read operation. The execution process is shown in Fig. 8, at last our new add function pci_new_dev _write() will be called for write while pci_new_dev_read() for read.

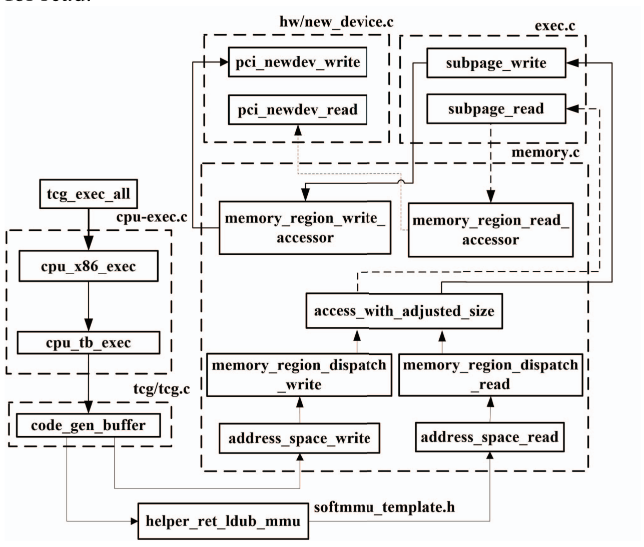


Figure 8. Execution processes of new device

V. Write Linux Driver and Verification

In this part, Linux kernel driver is developed to manage the new device. Linux has bus drivers and device drivers. The device drivers are what we commonly called as Linux drivers and the bus driver provides an API to the device driver for them to communicate with the hardware using a specific bus. Given the Linux device model is a complex data structure, we give the device creation process for simply. As PCI is a specific type of bus, so we can register our device driver create function inside the bus driver module. When kernel initializes device in PCI level, which interacts with the driver model, pci_bus_match() will be called if the driver can support the new device, if the match is successful, the probe function will be executed and our new driver will be loaded and initialized. Fig. 9 gives the driver creation process.

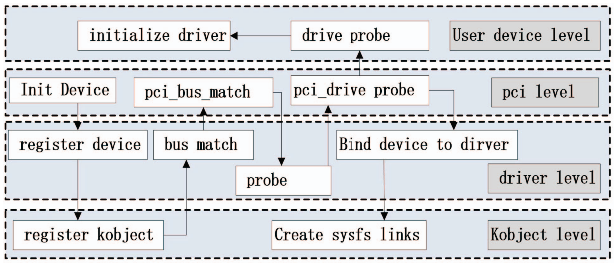


Figure 9. Driver creation process.

The new device driver is developed as a loadable kernel module. The new_dev_driver is a Linux kernel pci_driver struct which has a probe function pointer, points to the device probe function. We put our probe function newdev_probe() in this struct. When the module is loaded, newdev_probe() will be called, these process is shown in Fig.10.

```
module_init(newdev_init)
-> newdev_init()
-> pci_register_driver(&new_dev_driver)
```

(a) Snipped of C code used in registering new device module.

```
newdev_probe()
-> pci_iomap(pdev)
-> proc_create( “newdevice” , &proc_fops)
```

(b) Snipped of C code used in probing new device and loading module.

Figure 10. Driver register and load.

At this time, file /proc/newdevice will be created by kernel, and proc_fops is a variable for struct type file _operations, which has file operation function pointer like

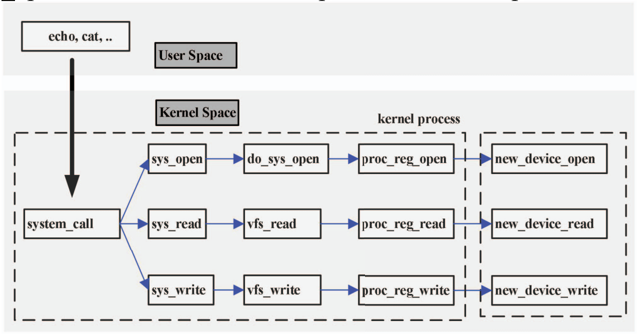
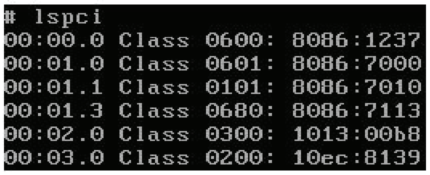


Figure 11. Code path of controlling the device.

open/read/write/close, we register these operations new _device_open/new_device_read/new_device/write at proc _fops, so when user mode program wants to read and write /proc/newdevice, the process is switched from user space to kernel space by system_call with file operation parameter, then the new device driver code will be called, execution process is show in Fig. 11.

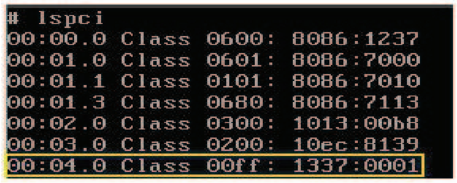
"lspci" [9] is a utility for displaying information about PCI buses in the system and devices connected to them. We can use it to check whether kernel detects the new device. Fig. 12 shows the result before inserting our new driver module.



```
# lspci
00:00.0 Class 0600: 8086:1237
00:01.0 Class 0601: 8086:7000
00:01.1 Class 0101: 8086:7010
00:01.3 Class 0680: 8086:7113
00:02.0 Class 0300: 1013:00b8
00:03.0 Class 0200: 10ec:8139
```

Figure 12. Execute lspci before inserting new device driver

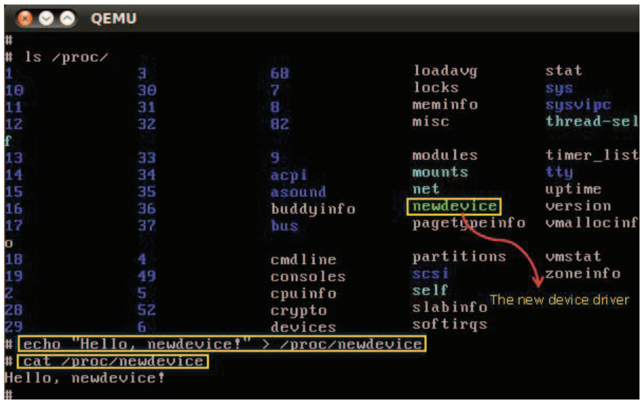
Fig. 13 shows the result after inserting the new device driver, the new device information is showed at the last line and the vender id is 0x1337, device id is 0x0001 which are defined in our QEMU virtual device module.



```
# lspci
00:00.0 Class 0600: 8086:1237
00:01.0 Class 0601: 8086:7000
00:01.1 Class 0101: 8086:7010
00:01.3 Class 0680: 8086:7113
00:02.0 Class 0300: 1013:00b8
00:03.0 Class 0200: 10ec:8139
00:04.0 Class 00ff: 1337:0001
```

Figure 13. Execute lspci after inserting new device driver

Then we can control the new device at user space. Use 'echo' [10] command to write data to the device and 'cat'[10] command to read data from the device. Fig. 14 shows that we write string "Hello, newdevice!" to file /proc/newdevice which is the device driver interface and the string is exactly written to virtual device emulated in QEMU. The data can also be read at the same time.



```
#
# ls /proc/
1              3              68             loadavg        stat
10             30             7              locks          sys
11             31             8              meminfo        sysvipc
12             32             82             misc           thread-sel
f
13             33             9              modules        timer_list
14             34             acpi           mounts         tty
15             35             asound         net            uptime
16             36             buddyinfo      newdevice      version
17             37             bus            pagetypeinfo   vmallocinf
o
18             4              cmdline        partitions     vmstat
19             49             consoles       scsi           zoneinfo
2              5              cpuinfo        self
28             52             crypto         slabinfo
29             6              devices        softirqs
# echo "Hello, newdevice!" > /proc/newdevice
# cat /proc/newdevice
Hello, newdevice!
#
```

Figure 14. Screen shot of write and read the device.

From the above test, our driver works successfully, the whole running, testing and debugging environment are built, and user level applications can be developed for the new virtual hardware without the physical device become available.

VI. CONCLUSIONS

In this paper, we build an unbeatable price tag and convenient environment which can be used to develop, test and debug target code without access to target hardware. Make software development take place before the hardware is ready. For future work, we will optimize QEMU to run faster for embedded software development.

ACKNOWLEDGMENT

This work is supported by the Natural Science Foundation of Jiangsu colleges and universities under grant No. 14KJD510004. The author would like to thank all the editors and anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] Alex Bennée, "Running Android L Developer Preview on 64-bit ARM QEMU," https://www.linaro.org/blog/core-dump/running-64bit-android-l-qemu/, August 2014.

[2] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," Proceedings of USENIX Annual Technical Conference, pp.41–46, June 2005.

[3] Y. Chen, J. Ren, H. Zhu and Y. C.Shi, "Dynamic binary translation and optimization in a whole-system emulator –SkyEye,"International Conference on Parallel Processing Workshops, pp. 329-336, August 2006.

[4] M. T. Jones, "Platform emulation with Bochs," http://www.ibm.com/developerworks/library/l-bochs/, January 2011.

[5] D. Mihocka, S. Shwartsman, "Virtualization without direct execution or jitting: designing a portable virtual machine infrastructure," Proceedings of 1st Workshop on Architectural and Micro architectural Support for Binary Translation, Beijing, vol. pp.1-16, 2008.

[6] R. L. Sites, A. Chernoff, M. B. Kirket, "Binary translation", Commun ACM, vol.36, pp. 69–81, 1993.

[7] M. Gligor, N. Fournel and F. Ptrot, "Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation", Proc. IEEE/ACM International conference on Hardware/software codesign and system synthesis, pp. 71-80, 2009.

[8] T. V. Dung, I. Taniguchi, H. Tomiyama, "Cache Simulation for Instruction Set Simulator QEMU," Proc. IEEE Symp. International Conference on Dependable, Autonomic and Secure Computing, August 2014, pp.441-446, doi:10.1109/DASC.2014.85.

[9] M. Mareš, "lspci," unpublished.

[10] Daniel J. Barrett, Linux Pocket Guide: Essential Commands 3rd ed., O'Reilly Media, pp.15-30, June 2016.