Virtual Machine Homework (II) Report

Introduction

- Topic: a study of system virtual machine
- Team
  - b00902064 Hao-En Sung @ ntu csie
  - b00902107 Shu-Hung You @ ntu csie
- Abstract

  Our code can be found at https://github.com/suhorng/vm14hw2

  We studied the same ISA system virtual machine on a simulated ARM Cortex A15x1 platform. We built one host VM which ran one or more guest VMs, traced the hypervisor traps in the **kvm** module. We also ran serveral benchmarks on multiple guest VMs to observe the loadings.

- Source File Structure
  - `README.md`: some notes and explanation of the repo.
  - `report.pdf`: our study report
  - `linuv-kvm`: our **full** source code; build kernel image here.
  - `fs/`: where we put our `host_disk.img`; absent on the repo.
    - `fs/host_utils/`: the scripts we put on the host VM to help manipulating traces, launch the guest VM, etc.
  - `boot/`: our `params` file; the other `.dtb` files are not pushed
  - `report/`: the source of our report
- Virtual Machines' Screenshot

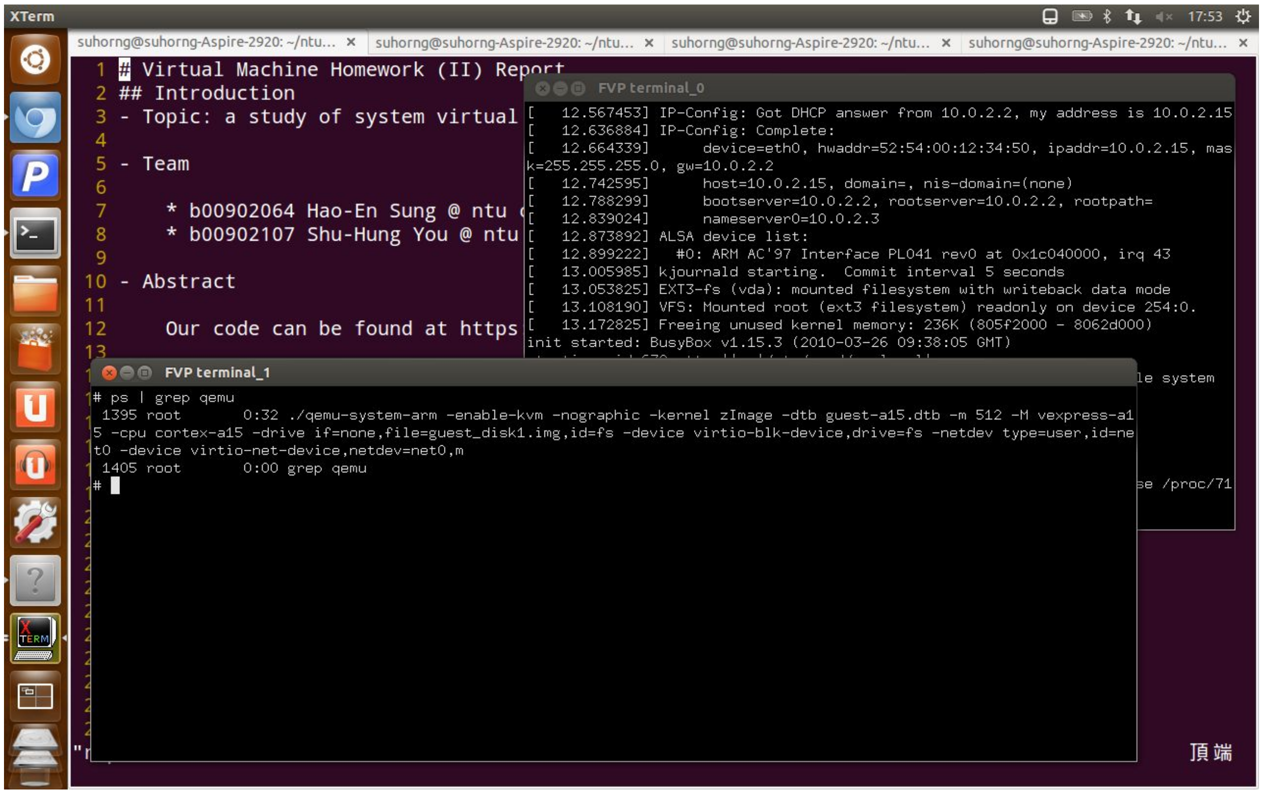
Part I - Trap Profiling

In this experiment, we traced several events in the KVM to see how many traps are there. We mainly focused on the HVC exception in the hypervisor mode and observed an unexpected result.

Virtual Machine Setup Difficulties

1. We chose to use the MMC file system instead of the NFS since there was little setup required to use a disk image, and its construction was similar to the guest machine. However, we initially only gave the host machine a 512MB disk, and was unable to put the guest inside the host since it was too large. Later, we rebuilt a host with enough disk space.
2. The given `qemu-system-arm` startup command was wrong. The given one was

```
#!/bin/sh
./qemu-system-arm \
    -enable-kvm \
    -nographic \
    -kernel zImage \
    -dtb guest-a15.dtb \
```


XTerm
17:53
suhorng@suhorng-Aspire-2920: ~/ntu...
suhorng@suhorng-Aspire-2920: ~/ntu...
suhorng@suhorng-Aspire-2920: ~/ntu...
suhorng@suhorng-Aspire-2920: ~/ntu...
FVP terminal_0
FVP terminal_1
頂端

```
# Virtual Machine Homework (II) Report
## Introduction
- Topic: a study of system virtual

- Team

    * b00902064 Hao-En Sung @ ntu
    * b00902107 Shu-Hung You @ ntu

- Abstract

    Our code can be found at https
```

```
[   12.567453] IP-Config: Got DHCP answer from 10.0.2.2, my address is 10.0.2.15
[   12.636884] IP-Config: Complete:
[   12.664339]      device=eth0, hwaddr=52:54:00:12:34:50, ipaddr=10.0.2.15, mas
k=255.255.255.0, gw=10.0.2.2
[   12.742595]      host=10.0.2.15, domain=, nis-domain=(none)
[   12.788299]      bootserver=10.0.2.2, rootserver=10.0.2.2, rootpath=
[   12.839024]      nameserver0=10.0.2.3
[   12.873892] ALSA device list:
[   12.899222]   #0: ARM AC'97 Interface PL041 rev0 at 0x1c040000, irq 43
[   13.005985] kjournald starting.  Commit interval 5 seconds
[   13.053825] EXT3-fs (vda): mounted filesystem with writeback data mode
[   13.108190] VFS: Mounted root (ext3 filesystem) readonly on device 254:0.
[   13.172825] Freeing unused kernel memory: 236K (805f2000 - 8062d000)
init started: BusyBox v1.15.3 (2010-03-26 09:38:05 GMT)
```

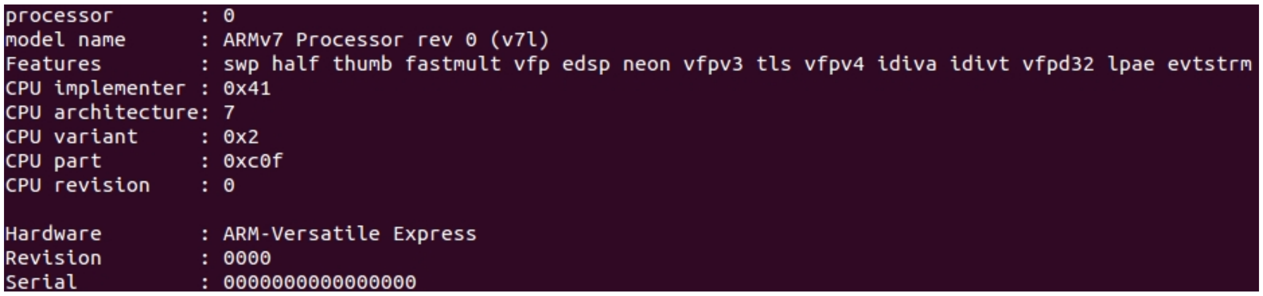
```
# ps | grep qemu
 1395 root       0:32 ./qemu-system-arm -enable-kvm -nographic -kernel zImage -dtb guest-a15.dtb -m 512 -M vexpress-a1
5 -cpu cortex-a15 -drive if=none,file=guest_disk1.img,id=fs -device virtio-blk-device,drive=fs -netdev type=user,id=ne
t0 -device virtio-net-device,netdev=net0,m
 1405 root       0:00 grep qemu
# 
```

Figure 1: VM screenshot

```
    -m 512 -M vexpress-a15 -cpu cortex-a15 \
    -drive if=none,file=guest_disk.img,id=fs \
    -device virtio-blk-device,drive=fs \
    -netdev type=user,id=net0 \
    -device virtio-net-device,netdev=net0,mac="52:54:00:12:34:50" \
    -append "console=ttyAMA0 mem=512 root=/dev/vda ip=dhcp"
```

But the `mem=512` parameter should be `mem=512M`. If not, our guest would not have enough memory to run.

After booting up the host/guest virtual machine, we checked for its `cpuinfo` and happliy found that our virtual machines were indeed working.



```
processor       : 0
model name      : ARMv7 Processor rev 0 (v7l)
Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x2
CPU part        : 0xc0f
CPU revision    : 0

Hardware        : ARM-Versatile Express
Revision        : 0000
Serial          : 0000000000000000
```

Figure 2: cpuinfo

The KVM Architecture Overview

XXX TODO insert KVM architecture graph here

As depicted in the graph, the KVM was a kernel module running inside the host VM that helps the guest VM to virtualize the CPU. For the I/O part, guest-issued I/O requests would be forwarded to the QEMU emulation system.

After entering the hypervisor mode of the ARM processor, the KVM kernel module could fully realize its potential in simplifying and speeding up the CPU virtualization. The interface of the KVM kernel module to the outside world was, as usual, via the `ioctl` function. The handler `kvm_vm_ioctl` and `kvm_vcpu_ioctl` were at `virt/kvm/kvm_main.c` with architecture-dependent functionalities be forwarded to, said, `kvm_arch_vcpu_ioctl1` in `arch/arm/kvm/arm.c`.

It mainly had the following instructions:

1. Create a virtual CPU

```
static long kvm_vm_ioctl(/* ... */) {
  /* ... */
  switch (ioctl) {
  case KVM_CREATE_VCPU:
    r = kvm_vm_ioctl_create_vcpu(kvm, arg);
    break;
    /* ... */

static int kvm_vm_ioctl_create_vcpu(/* ... */) {
  /* ... */
  vcpu = kvm_arch_vcpu_create(kvm, id);
  /* ... */
  r = kvm_arch_vcpu_setup(vcpu);
  /* ... */

struct kvm_vcpu *kvm_arch_vcpu_create(/* ... */) {
  int err; struct kvm_vcpu *vcpu;
  vcpu = kmem_cache_zalloc(kvm_vcpu_cache, GFP_KERNEL);
  /* ... */
  err = kvm_vcpu_init(vcpu, kvm, id);
  /* ... */
```

2. Run on a virtual CPU

```
static long kvm_vcpu_ioctl(/* ... */) {
  /* ... */
  switch (ioctl) {
  case KVM_RUN:
    r = -EINVAL;
    if (arg)
      goto out;
    r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
    trace_kvm_userspace_exit(vcpu->run->exit_reason, r);
    break;
    /* ... */

int kvm_arch_vcpu_ioctl_run(/* ... */) {
  /* our main modifications here! */
  /* ... *virtualize* the cpu; enter guest mode and execute! */
```

3. Get/set virtual CPU register
4. Initialize a ARM CPU from use request

   ```
   long kvm_arch_vcpu_ioctl(/* ... */) {
     /* ... */
     switch (ioctl) {
     case KVM_ARM_VCPU_INIT: {
       /* ... */
   ```

To use CPU virtualization, we shall more or less go through the above two instructions.

Implementation

Our implementation made two changes to the KVM, and we simply utilized other existing tracing points to make our observation. The first one was to add an *exit count* to the original `kvm_exit` trace event. `kvm_exit` traces when the processor leaves non- hypervisor mode and back to the KVM run loop:

```
 int kvm_arch_vcpu_ioctl_run(/* ... */) {
   /* ... */
   while (ret > 0) {
     /* ... */
     local_irq_disable();
     /* ... */
     trace_kvm_entry(*vcpu_pc(vcpu));
     kvm_guest_enter();
     vcpu->mode = IN_GUEST_MODE;

     ret = kvm_call_hyp(__kvm_vcpu_run, vcpu);

     vcpu->mode = OUTSIDE_GUEST_MODE;
     vcpu->arch.last_pcpu = smp_processor_id();
     kvm_guest_exit();
-    trace_kvm_exit(*vcpu_pc(vcpu));
+    ++vcpu->cnt_exit;
+    trace_kvm_exit(*vcpu_pc(vcpu), vcpu->cnt_exit);
     /* ... */
```

Then we modified the `kvm_exit` trace-event in `arch/arm/kvm/trace.h` to print exit count. The second one was to add our own trace event to record the HVC trap count, which was the hypervisor's trap in ARM hypervisor mode. According to the ARM architecture reference manual, this exception happened whenever an `HVC` instruction was issued (i.e. the guest requested to switch to hypervisor mode. This might happen in a `virtio` defice) or when some exceptions was routed to the hypervisor.

1. Add our trace event in `arch/arm/kvm/trace.h`

   ```
   /* tracing exception file */
   TRACE_EVENT(kvm_exchvc,
     TP_PROTO(unsigned long vcpu_pc, int cnt_exchvc, int kvm_cond_valid),
     TP_ARGS(vcpu_pc, cnt_exchvc, kvm_cond_valid),

     TP_STRUCT__entry(
       __field(unsigned long, vcpu_pc)
   ```

```
    __field(int, cnt_exchvc)
    __field(int, kvm_cond_valid)
  ),

  TP_fast_assign(
    __entry->vcpu_pc = vcpu_pc;
    __entry->cnt_exchvc = cnt_exchvc;
    __entry->kvm_cond_valid = kvm_cond_valid;
  ),

  TP_printk("PC: 0x%08lx; trap count: %d; kvm_cond_valid: %s",
    __entry->vcpu_pc,
    __entry->cnt_exchvc,
    __entry->kvm_cond_valid? "true" : "false")
);
```

2. Insert out trace events to appropiate points. We figured out that the main loop would check for exception upon every exit, i.e.

   ```
   int kvm_arch_vcpu_ioctl_run(/* ... */) {
       /* ... */
       kvm_guest_exit();
       /* ... */
       local_irq_enable();
       /* ... */
       ret = handle_exit(vcpu, run, ret);
     }
   ```

   Hence we could safely insert our trace point in exception dispatching function:

   ```
    int handle_exit(/* ... */) {
      exit_handle_fn exit_handler;

      switch (exception_index) {
      /* ... */
      case ARM_EXCEPTION_HVC:
        if (!kvm_condition_valid(vcpu)) {
    +       ++vcpu->cnt_exchvc;
    +       trace_kvm_exchvc(*vcpu_pc(vcpu), vcpu->cnt_exchvc, 0);
            kvm_skip_instr(vcpu, kvm_vcpu_trap_il_is32bit(vcpu));
            return 1;
        }

    +   ++vcpu->cnt_exchvc;
    +   trace_kvm_exchvc(*vcpu_pc(vcpu), vcpu->cnt_exchvc, 1);
        exit_handler = kvm_get_exit_handler(vcpu);

        return exit_handler(vcpu, run);
   ```

3. And the last step was to turn on our trace event.

To record the exit count and the HVC trap count, we added two additional counters in the `kvm_vcpu` structure defined in `include/linux/kvm_host.h`:

```
 struct kvm_vcpu {
   /* ... */
 #endif
     bool preempted;
     struct kvm_vcpu_arch arch;

+  int cnt_exchvc;
+  int cnt_exit;
 };
```

Upon tracing the code, we actually found a trace event `trace_kvm_hvc` in the HVC handleing function, `handle_hvc`. That was, we actually did not really need to create a brand new trace event. Anyway, we did it for fun.

Trace Result

The result was interesting. Since traps to the hypervisor mode was non-stopping, we could not count the exact number of traps during guest boot-up. We hence counted the trap number of guest boot-up followed by an immediate `poweroff`.

The result count was as follows.

```
179.182896: kvm_entry:  PC: 0x802c1f24
179.183041: kvm_exit:   PC: 0x802c21c8; exit count: 80671
179.183096: kvm_exchvc: PC: 0x802c21c8; trap count: 74322; kvm_cond_valid: true
179.183120: kvm_guest_fault: ipa 0x1c010000, hsr 0x93800006, hxfar 0xf80100a8, pc 0x802c21c8
179.183166: kvm_userspace_exit: reason KVM_EXIT_MMIO (6)
179.200710: kvm_userspace_exit: reason restart (4)
```

Running it again turned out to have similar trap counts:

```
181.829151: kvm_entry:  PC: 0x802c1f24
181.829296: kvm_exit:   PC: 0x802c21c8; exit count: 81072
181.829351: kvm_exchvc: PC: 0x802c21c8; trap count: 74382; kvm_cond_valid: true
181.829375: kvm_guest_fault: ipa 0x1c010000, hsr 0x93800006, hxfar 0xf80100a8, pc 0x802c21c8
```

Interestingly, we found a large number of HVC trap events without interleaving `kvm_guest_fault` (which we taken to be memory faults as indicated by the `kvm_userspace_exit` event in some places):

```
... repeated many many times ...
124.408340: kvm_entry:  PC: 0x8001086c
124.408389: kvm_exit:   PC: 0x80010868; exit count: 1777
124.408428: kvm_exchvc: PC: 0x80010868; trap count: 1760; kvm_cond_valid: true
124.408477: kvm_entry:  PC: 0x8001086c
124.408526: kvm_exit:   PC: 0x80010868; exit count: 1778
124.408565: kvm_exchvc: PC: 0x80010868; trap count: 1761; kvm_cond_valid: true
124.408628: kvm_entry:  PC: 0x8001086c
124.408675: kvm_exit:   PC: 0x80010868; exit count: 1779
124.408714: kvm_exchvc: PC: 0x80010868; trap count: 1762; kvm_cond_valid: true
124.408763: kvm_entry:  PC: 0x8001086c
124.408812: kvm_exit:   PC: 0x80010868; exit count: 1780
124.408851: kvm_exchvc: PC: 0x80010868; trap count: 1763; kvm_cond_valid: true
```

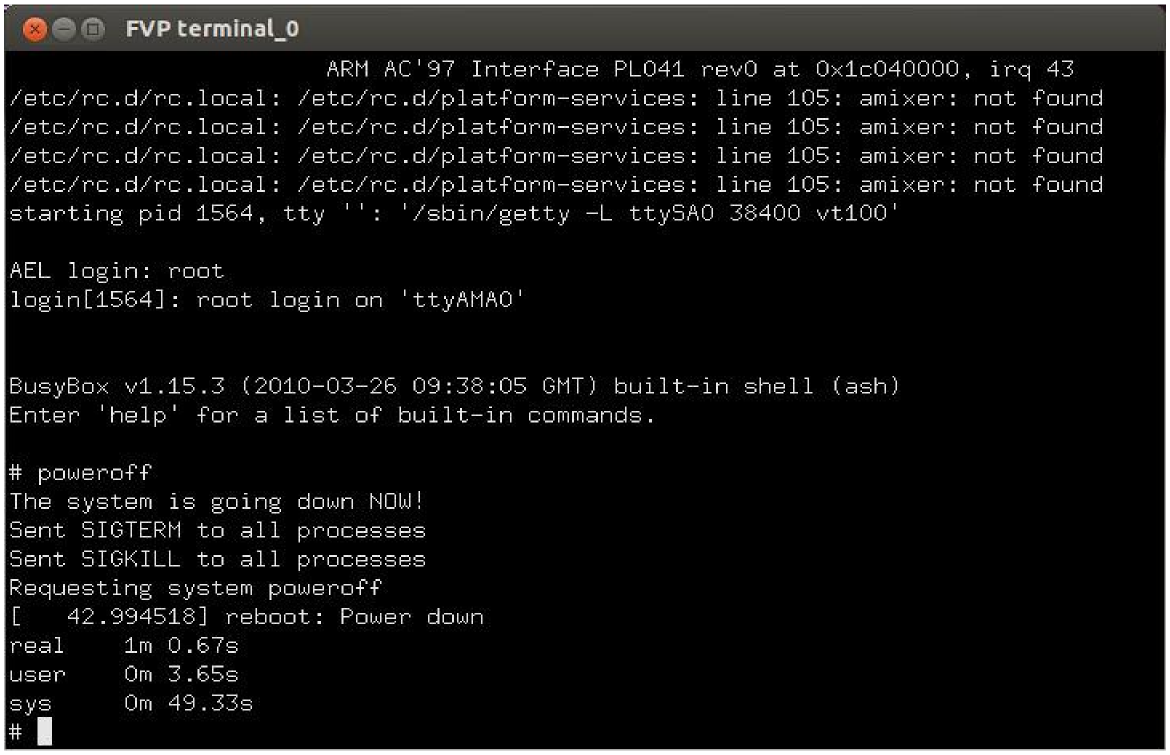
```
124.408900: kvm_entry:  PC: 0x8001086c
124.408949: kvm_exit:   PC: 0x80010868; exit count: 1781
124.408988: kvm_exchvc: PC: 0x80010868; trap count: 1764; kvm_cond_valid: true
... repeated many many times ...
```

Similar places occured many times, each time having different PC. Currently we had no explanation about it.

Discussions

When we booted either the host or the guest, there were some places where the boot process hung very long. We did not know if it was caused by some I/O failure or wrong device settings, but we suspected that it was where the strange trace messages get logged.

Apart from the strange HVC trap events, we also found that the time in the guest virtual machine differed from that of the actual running time on the host machine. This might be an indicator on whether the virtualization was effecient.



```
                      ARM AC'97 Interface PL041 rev0 at 0x1c040000, irq 43
/etc/rc.d/rc.local: /etc/rc.d/platform-services: line 105: amixer: not found
/etc/rc.d/rc.local: /etc/rc.d/platform-services: line 105: amixer: not found
/etc/rc.d/rc.local: /etc/rc.d/platform-services: line 105: amixer: not found
/etc/rc.d/rc.local: /etc/rc.d/platform-services: line 105: amixer: not found
starting pid 1564, tty '': '/sbin/getty -L ttySA0 38400 vt100'

AEL login: root
login[1564]: root login on 'ttyAMA0'


BusyBox v1.15.3 (2010-03-26 09:38:05 GMT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# poweroff
The system is going down NOW!
Sent SIGTERM to all processes
Sent SIGKILL to all processes
Requesting system poweroff
[   42.994518] reboot: Power down
real    1m 0.67s
user    0m 3.65s
sys     0m 49.33s
# 
```

Figure 3: guest running time

In the above screenshot, the guest was running to 42.9s where the host actually ran 1 minute. So the speed in the guest was only about 2/3 of the host in this case. Note that the guest time differed from the above trace results since we were not enabling tracing this time. If so, the running time will be much larger.

We once tried to find the trace event `kvm_emulate_insn` on our machine. However, later we discovered that it only exists on x86 machines. This meant that the ARM processor was actually doing well in supporting virtualization so we did not need much emulation.

To support this argument, we might note that the exit count (from `kvm_exit`) and the HVC trap count (from `kvm_exchvc`) were not equal, but the former number was very close to the latter one. This meant that for most of the traps (or any other reason) that caused the virtualization to get back to the hypervisor, only a small amount of nubmer was caused by other routed exceptions. We spent most of the time on hypervisor calls, and for other times we almost avoided other overheads.

Part II - Experimenting Multiple Guests

Difficulties Setting Up The Environment

We were confused about the purpose of Part IV requirement initially. Furthermore, we did not know how to build up the host system. We assumed that the `qemu-debootstrap` was an extension program which was appended on the previous host system. However, it was not the case. This type of file system was different from those based on image file, instead, it was stored in a folder. After storing the whole file system into an image file, we were able to boot up the system.

A next interesting point was encountered when we testing benchmarks in both host and guest system, we found out that some binary which was based on ARM architecture in host system could not be executed in guest system, but some of them could. After doing some experiments, we finally figured out the reason. Some benchmarks used dynamic library linking instead of static library linking. Thus, once host system library was different from the guest one, the binary file would encounter a load error.

Benchmarking Guests

We used five benchmarks from the MiBench (http://www.eecs.umich.edu/mibench/source.html) to test the performance of each virtual machine, including `telecomm/adpcm`, `telecomm/gsm`, `telecomm/CRC32`, `telecomm/FFT`, and `comsumer/jpeg`. The following table shows the experiment result. There were five major rows, each standed for a benchmark.

There were five columns, including local system, host system, one guest system on host system, and two guest systems on host system. Two-guest-systems experiment was designed to test system performance with heavy loading. To be specific, two same benchmarks ran on two guest system individually, and simultaneously.

Discussions

There was a huge performance gap between local and host system. However, it was surprising that guest system has little overhead when compared with host system. The reason might be that ARM-to-X86 simulation was harder than ARM-to-ARM one.

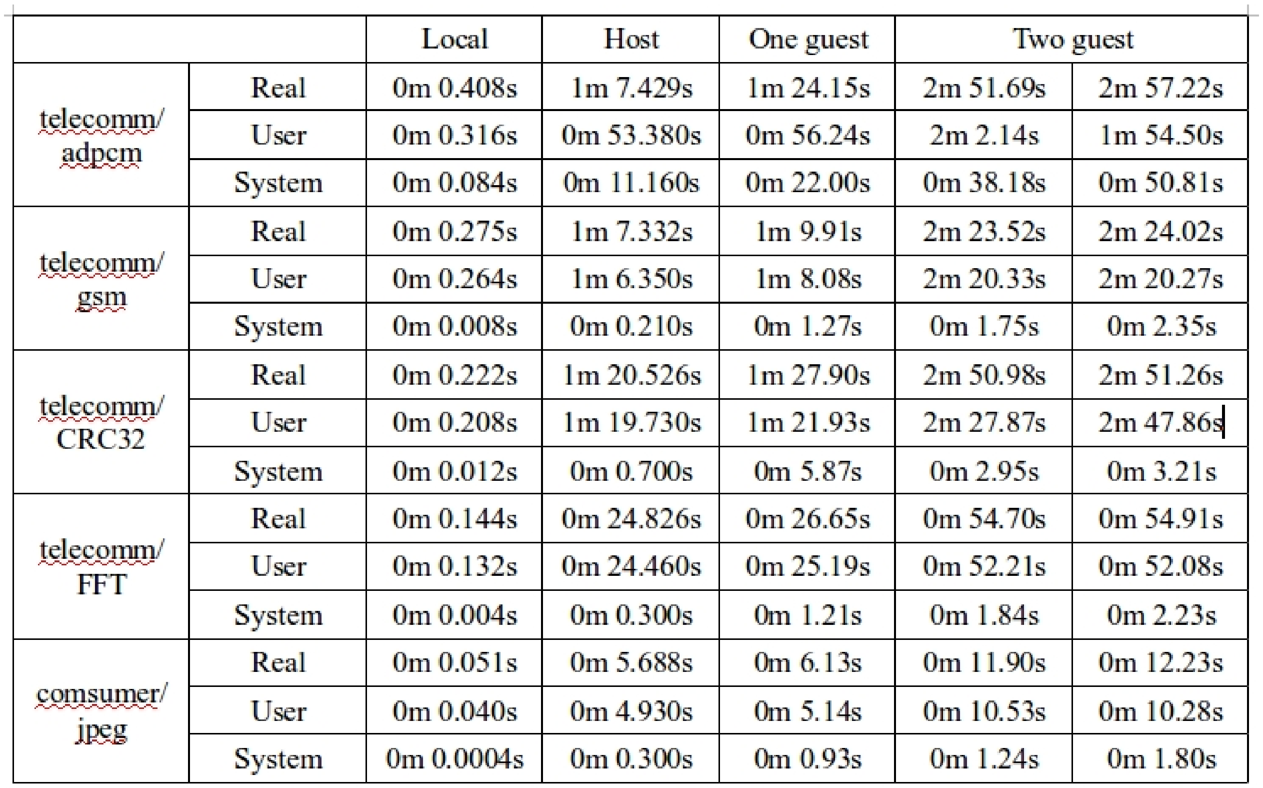
When two guest system was running on one same host system, performance was about half of the original one, which sounded pretty reasonable. The slightly difference on execution time between two guest systems may be caused by context-switch priority.

Conclusion

In this homework, we studied how the kernel module KVM runs in hypervisor mode to support efficient CPU virtualization as well as benchmarked the virtualized CPU on one or more guest virtual machines.

We learnt the KVM kernel module by modifying its code, adding our own trace functions and modifying the existing ones. From the trace result, we speculated that the hypervisor mode was doing a good job of virtualizing the CPU. We can also estimate the running-speed ratio of the guest from the timestamp in the log.

For multiple guest benchmarking, we guessed that the guest shall share their time with none having priorities over the others. The guest machines ran almost as fast as the host virtual machine, yet the host virtual machine ran much slower than our **real** machine. The simulator was faithfully doing its job, but simulating is nonetheless slow.



		Local	Host	One guest	Two guest	
telecomm/ adpcm	Real	0m 0.408s	1m 7.429s	1m 24.15s	2m 51.69s	2m 57.22s
	User	0m 0.316s	0m 53.380s	0m 56.24s	2m 2.14s	1m 54.50s
	System	0m 0.084s	0m 11.160s	0m 22.00s	0m 38.18s	0m 50.81s
telecomm/ gsm	Real	0m 0.275s	1m 7.332s	1m 9.91s	2m 23.52s	2m 24.02s
	User	0m 0.264s	1m 6.350s	1m 8.08s	2m 20.33s	2m 20.27s
	System	0m 0.008s	0m 0.210s	0m 1.27s	0m 1.75s	0m 2.35s
telecomm/ CRC32	Real	0m 0.222s	1m 20.526s	1m 27.90s	2m 50.98s	2m 51.26s
	User	0m 0.208s	1m 19.730s	1m 21.93s	2m 27.87s	2m 47.86s
	System	0m 0.012s	0m 0.700s	0m 5.87s	0m 2.95s	0m 3.21s
telecomm/ FFT	Real	0m 0.144s	0m 24.826s	0m 26.65s	0m 54.70s	0m 54.91s
	User	0m 0.132s	0m 24.460s	0m 25.19s	0m 52.21s	0m 52.08s
	System	0m 0.004s	0m 0.300s	0m 1.21s	0m 1.84s	0m 2.23s
comsumer/ jpeg	Real	0m 0.051s	0m 5.688s	0m 6.13s	0m 11.90s	0m 12.23s
	User	0m 0.040s	0m 4.930s	0m 5.14s	0m 10.53s	0m 10.28s
	System	0m 0.0004s	0m 0.300s	0m 0.93s	0m 1.24s	0m 1.80s

Figure 4: benchmark