Development of Type-2 Hypervisor for MIPS64 Based Systems

July 15

2013

[1st Deliverable]

This document reports on the implementation challenges faced during 1st deliverable of "Development of Type 2 Hypervisor for MIPS64 based Systems" project, funded by National ICT R & D Fund Pakistan. This report starts with brief description of project objectives, technical details of our approach, challenges and their solutions. Complete description of testing infrastructure, test cases and test results are discussed later on. The report concludes with the impact of 1st deliverable on the overall project progress.

Test Cases Result Report

High Performance Computing and Networking Laboratory HPCNL
Al-Khwarizmi Institute of Computer Science, University of Engineering and Technology Lahore Pakistan

1.Project Description

The main objective of this project is to develop an open source Type 2 hypervisor, for Linux-based MIPS64 embedded devices. Type-2 means that it is a hosted hypervisor which runs on MIPS64 based Linux systems as a Linux process. It is intended that the hypervisor will (1) support installation and execution of un-modified MIPS64 Linux guest(s) on un-modified MIPS64 Linux host (2) take advantage of virtualization for improved hardware utilization and performance optimization, by using multiple MIPS cores. Our focus on MIPS is due to the fact that MIPS based systems are lagging behind in the use of virtualization. One of the reasons is that many MIPS based processors are used in low end consumer devices like TV set top box, GPS navigation system and printers. There isn't a clear cut use case for virtualization here. But few of the MIPS vendors target higher end embedded devices like network switches and routers, GSM/LTE base station equipment and MIPS based blade servers. There are clear-cut virtualization use cases for this higher-end MIPS segment.

The hypervisor is named "hypervisor2", where "2" stands for type 2. The development started on April 1, 2013 and first deliverable is due after 3.5 months i.e. July 15, 2013. In first deliverable, we need to build the required infrastructure. The infrastructure should print guest kernel banner on console, at the end of 1st deliverable.

2.Development Strategy

We are following a hybrid approach to develop hypervisor2. Executable binary is loaded in the address space of hypervisor2 and mapped to a known memory address. Traditional trap-and-emulate technique is used to take control of each instruction. Hybrid approach works as following:

1. If the instruction is privileged, it is emulated.
2. If the instruction manipulates `sp`, `gp` and/or `k0` registers, it is dynamically patched before execution.
3. Otherwise, the instruction is executed directly on hardware as it is.

3.1.Challenges and Solutions

Development of a hypervisor is quite challenging. Runtime systems like hypervisor are typically sensitive to runtime overhead. Runtime overheads, like that of emulation, result in significant performance degradation if not taken care of. To reduce runtime overhead, our initial strategy was to emulate privileged instructions only and execute rest of the instructions on

bare metal (hardware). On execution of privileged instruction in user mode, a trap is generated (i.e. SIGILL signal is raised). We implemented a signal handler that catches signal, fetch/decode the instruction and emulate its behavior.

Challenge 1
Standard C library (i.e. glibc) does not allow modification of sp ($29) and gp ($28) registers in user mode. Non-privileged instructions dealing with these registers can't be executed directly on hardware. Similarly, K0 ($26) and K1 ($27) registers produce unexpected results because they are interrupt handling registers used by kernel and potentially not used by user programs.

Solution 1
In addition to emulation of privileged instructions, we implemented the code for emulation of non-privileged instructions involving gp and sp register.

Challenge 2
The next challenge was that any instruction can potentially manipulate gp and sp registers and we may end up in emulating all instructions, resulting in poor performance.

Solution 2
We implemented code for dynamic code patching and patched all instructions involving sp($29), gp($28) and k1($27) registers. Patched instructions were harmlessly executed on hardware and contents of corresponding registers were updated later (in a trap handler).

Challenge 3
To ensure correct execution of guest code, we need to use debugger extensively during development. With the increasing number of executed instructions, debugging information becomes complex and hard to read. In case of an error condition, we need to determine the instruction that produced error. Searching the error-causing instruction between two states of emulator is not a trivial task.

Solution 2
In this stage, we generate trap on every instruction so that debugging and testing could be made easier. Now, the guest code is executed using a hybrid approach: privileged instructions are emulated, instructions involving sp, gp, k0 registers are patched and the rest are allowed to execute on hardware unchanged.

4.Testing Infrastructure

Testing infrastructure involves MIPS64 evaluation board with multicore Octeon processor, hardware debugger (JTAG), development system and testing routines. We need rigorous testing to make sure that guest kernels run in complete isolation from each other and from host kernel. Similarly, on each instruction execution in virtualized environment, changes to system state should imitate the changes made by executing the same in real environment.

4.1.Test Cases

Hypervisor manipulates (i.e. emulation/code patching) guest code to use privileged hardware resources controlled by host kernel. Hence, various test cases are needed to make sure the consistency and integrity of guest code. During first deliverable, our focus is on four types of test cases.

4.1.1. Matching system states

In our case, system state consists of the values of general purpose registers and some of coprocessor 0 (CP0) registers at a particular instance. In order to verify the correct working of hypervisor, we run (same) executable binary directly on Cavium MIPS64 board and through hypervisor. We get real system state on each privileged instruction by using JTAG and compare both outputs (hypervisor and JTAG) for verification. JTAG provides the facility of setting hardware breakpoints at each privileged instruction to stop and take log of system state. Without setting breakpoints, it logs the state at every instruction execution.

4.1.2. Execution path

Due to emulation and code patching, guest code execution path may differ from that of the same binary running directly on board. Taking Log at breakpoints may fail due to unavailability of a priori information about execution path of guest code. For example, if guest code sway from the path containing some breakpoint, we would not be able to take system state at that breakpoint and state matching test result will be misleading.

Logging system state after each instruction execution could help in avoiding the situation of taking wrong execution path. This allows us to debug the potential causes of error (if any) by looking at system state before and after the execution of malfunctioning instruction. However, there is inherent overhead of logging state at each instruction execution. There were about 339351 instructions executed by u-boot. JTAG created a file of about 6MB in

approximately 7 hours. Generated file contains data (i.e. general purpose registers + CP0 registers content) of about 2600 states. To reduce state logging time, we decided to use a small binary (i.e. code for irrelevant external devices is commented out) and take log on Quick Emulator (QEMU). To take log on QEMU, we used the expertise of another HPCNL team working on a different project titled “System Mode Emulation in QEMU”.

4.1.3. Comparing Console Output

On reaching the stage where console is get attached with our hypervisor, the binaries, executing within hypervisor, starts emitting messages on console. It serves as another way of validation, whereby output of our hypervisor is compared with that of real MIPS system.

4.1.4. Progress

The progress is tracked by identifying labeled blocks, in binary code. The blocks are identified by following the control flow of binary. When the instructions in one block are executed, its label is noted and control is conditionally/unconditionally transferred to the next block in control flow. This way we measure the progress that how many block have been executed and how many left.

Emulation and code patching may lead to infinite loops in the code. For example, if emulation/patching changes system state in such a way that control is transferred to one of prior blocks of the current block, the hypervisor will enter into an infinite loop. We need to avoid the situations like this in order to make progress.

4.2.Test Results

The sample output of system state test, hypervisor console, and execution path test is elaborated in this section.

4.2.1. Output of System State Matching Test

We trap at every instruction and create a state-file. We match this state file with qemu log state-file to see if any register contains different contents. Mismatches are written in other file as shown in figure 1.

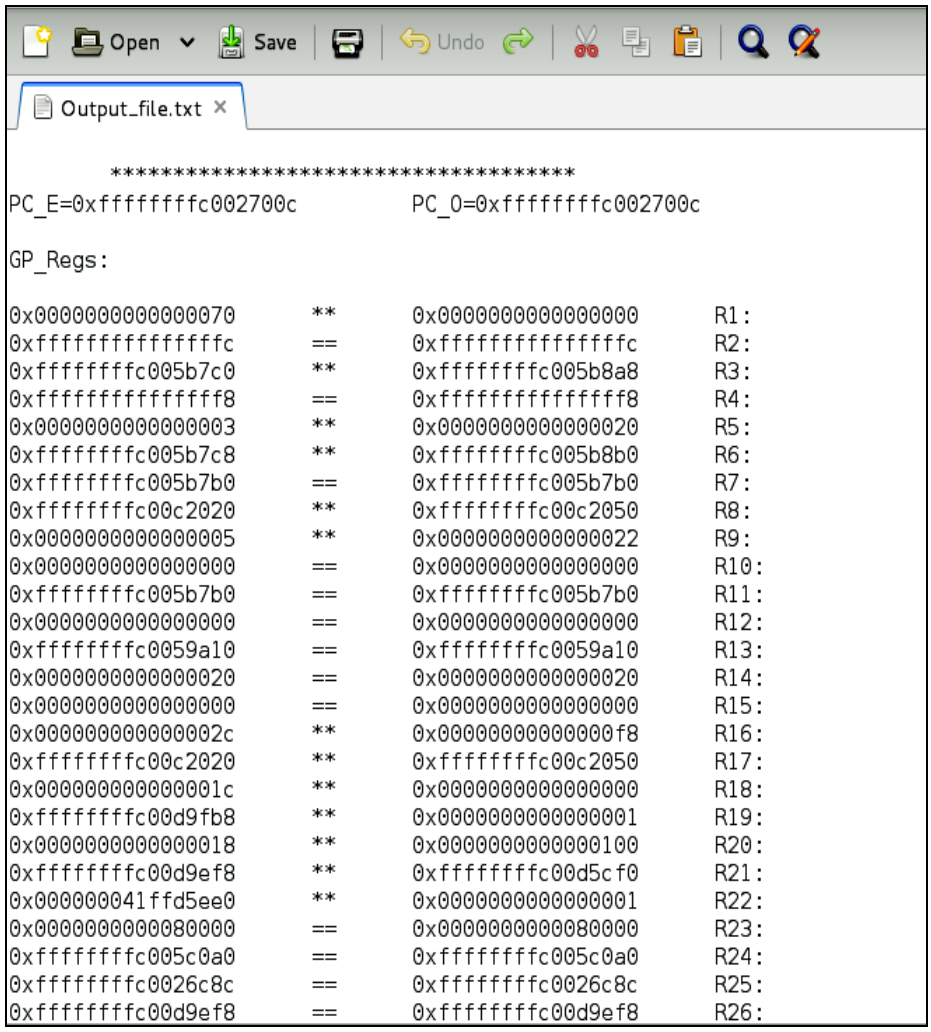

Figure 1: Output of system state matching test.

4.2.2. Output of Execution Path Test

We face difficulties in debugging if QEMU log is missing instruction log at different points. To ensure that the hypervisor is on the right track we match the Program Counter (PC) values taken by hypervisor and all the PC values taken in qemu log, as shown in figure 2.

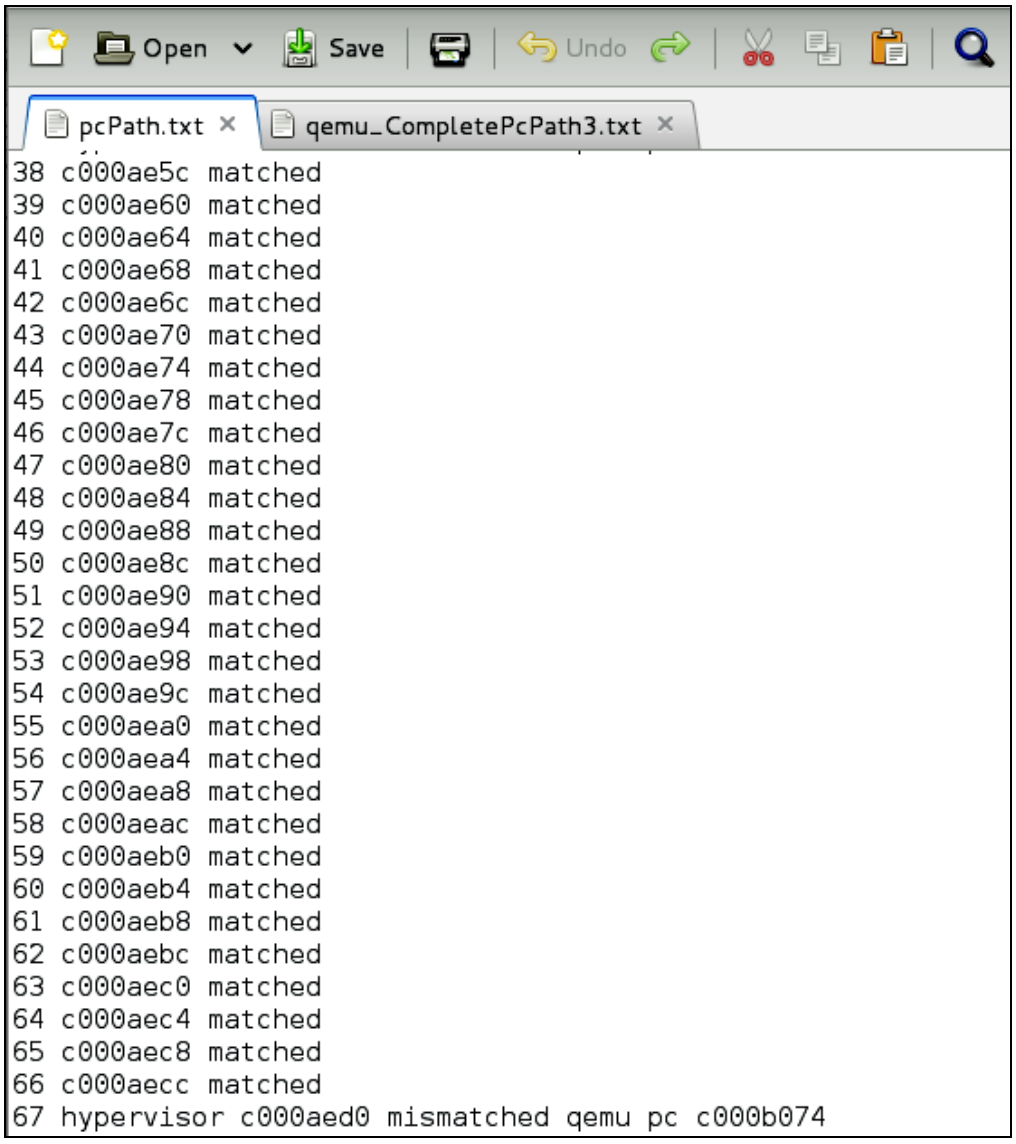

Figure 2: Output of Execution Path Test.

4.2.3. Output on Hypervisor Console

During execution, hypervisor makes a call to the code written for console I/O. On console attachment, the binaries, executing within hypervisor, can start printing on the hypervisor console. To validate virtual execution of binaries, hypervisor console output (shown in Figure 3) was compared with that of real host system console.

5.Impact on Project Progress

Although development constraints forced us to change the order of couple of milestones, we are not expecting any impact on overall progress of project. Our progress is quite satisfactory and according to the expectations.

```
root@octeon:/home/Asad_data/hypervisor2-clone# ./dist/Debug/GNU-Linux-x86/hypervisor2-clone

        ********************   Main():Start Here  ************************
        Loaded binary address...: 0x000000004b883000
REGION_ADDR...: 0x000000002ba4c000 REGION_SIZE = 0x80
REGION_ADDR...: 0x000000002ba4d000 REGION_SIZE = 0x200
REGION_ADDR...: 0x000000004bc83000 REGION_SIZE = 0x10000000
REGION_ADDR...: 0x000000005bc83000 REGION_SIZE = 0x80000
REGION_ADDR...: 0x000000005bd03000 REGION_SIZE = 0x80000
                    :
                    :
U-Boot 1.1.1 (Development build, svnversion: u-boot:exported, exec:exported) (Bu

BIST check passed.
Warning: Board descriptor tuple not found in eeprom, using defaults
EBH5610 board revision major:1, minor:0, serial #: unknown
OCTEON CN56XX-NSP pass 2.0, Core clock: 0 MHz, DDR clock: 0 MHz (0 Mhz data rate
DRAM:  1024 MB
Clearing DRAM........ done
Flash boot bus region not enabled, skipping NOR flash config
                    :
                    :
```

Figure 3: Output on hypervisor console.

~*~*~*~