HTTM - Design and Implementation of a Type-2 Hypervisor for MIPS64 Based Systems

Qurrat ul Ain*, Usama Anwar, Muhammad Amir Mehmood and Abdul Waheed

Al-Khwarizmi Institute of Computer Science, University of Engineering and Technology, Lahore, Pakistan

*a qurratulain@kics.edu.pk

Abstract. Virtualization has emerged as an attractive software solution for many problems in server domain. Recently, it has started to enrich embedded systems domain by offering features such as hardware consolidation, security, and isolation. Our objective is to bring virtualization to high-end MIPS64 based systems, such as network routers, switches, wireless base station, etc. For this purpose a Type-2 hypervisor is a viable software solution which is easy to deploy and requires no changes in host system. In this paper we present the internal design HTTM - A Type-2 hypervisor for MIPS64 based systems and demonstrate its functional correctness by using Linux Testing Project (LTP) tests. Finally, we performed LMbench tests for performance evaluation.

1. Introduction

Recent years have seen manifold increase in the processing power of computing devices. However, these hardware advancements become ineffective if the software running on top of such hardware is not able to fully utilize its processing capabilities. Virtual environment enables us to safely use a single hardware for multiple purposes. Virtualization is an effective software solution for multiple fields of computing. For instance, in the form of Type-1/ bare-metal hypervisors (e.g., Xen [1]), it has revolutionized the sever domain while Type-2/ hosted hypervisors (e.g., VMWare workstation [2]) have proved their benefits for client domain.

Virtualization has attracted developers of embedded systems to consider it as a viable solution for their problems such as security, consolidated infrastructure, isolated debugging environment, etc., [3-5]. Companies like Motorola and VMware have used virtualization as a possible solution by creating Evoke [6] and Mobile Virtualization Platform (MVP) [7] respectively. Gu et al. [8] have implemented a real-time application on integrated automotive gateway platform. They used KVM (Type-2 hypervisor) as their virtualization layer and created two guest systems. One guest executes security critical tasks e.g., parking assistance while the other run applications that can compromise security due to connection to external network e.g., GPS navigation or mobile office.

There are number of MIPS based devices already installed and functional in the current network infrastructure including security gateways, network routers, switches, base station, etc. To benefit from the valuable aspects of virtualization, these devices can use a Type-2 hypervisor which can be easily installed as a user process without altering existing infrastructure.

In this paper, we have designed HTTM -- Hypervisor Type-2 MIPS-On-MIPS which support full-virtualization i.e., it does not need the host or guest Linux OS to be patched or changed. In Section 2,
we present the internal design of hypervisor while in Section 3 we describe the results of its functional testing. In Section 4 we conclude this presented work.

2. Design of HTTM

HTTM is a user space process for Linux based MIPS64 host system. It creates an abstract layer of virtual resources for the MIPS64 guest Linux system running on it. For development and debugging purposes we used Cavium Octeon MIPS64 evaluation boards. Octeon II CN6860 board with 12 cores, 1.2 GHz, and 4GB RAM was used as a host system, which runs their customized 2.6.32 Linux kernel. Octeon CN5750 was the guest/ target system with 12 cores, 800 MHz, and 1GB RAM.

The virtual resources consist of CPU cores, memory management Unit (MMU) and some I/O devices. Figure 1 shows an overview of HTTM which is running on host Linux system along with the other user space processes. HTTM can broadly be categorized in two sections: i) Virtual resource and ii) Management. The virtual resource section is responsible for correct representation of guest/ target system and its on-board units that includes cores, Translation Look-aside Buffer (TLB), General Purpose (GP) and Co-Processor (CP) registers, etc. While the management section helps that virtual resource to complete its tasks correctly and efficiently. It includes Dynamic Binary Translation (DBT) unit, caches, guest-to-host translation unit, and signal handler.

Multithreaded approach was implemented in HTTM with cores and devices executing in separate threads. The basic execution cycle of a core thread is simple fetch-translate-execute. A set of instructions with control shifting instruction at the end (forming a block) is fetched from the guest code and translated into set of host-safe instructions which is then directly executed on the host. If interrupts or exceptions are generated during the execution, control is transferred to the management section which handles it accordingly. Figure 2 shows a simplified version of execution cycle in a core thread. Next we give a brief description of different units of HTTM and virtual guest system.

2.1. DBT for MIPS64 instruction set

As HTTM executes in the user space rather than directly on hardware, particular constraints of user space hinders in direct execution of guest MIPS instruction over host system. For instance host Linux system inhibits the user space process to write over its ra (return address) and sp (stack pointer) registers, therefore we cannot directly map the guest GP register to host’s GP registers. Hence, we need DBT to translate the guest assembly instructions into host-safe assembly instructions and then execute them directly on hardware. Other than basic translation mechanism for ISA virtualization, correct execution of guest code requires addition of some set of instructions in the translated block e.g., privilege check for privileged instructions, Program Counter (PC) increment at every instruction, etc. Due to this fact the translated block becomes larger than the guest code block. We have provided
translation for 184 MIPS instructions out of which 159 were standard while 25 were Cavium specific instructions. Cache is maintained for translated blocks to avoid translation overhead.

2.2. Memory management unit
Standard MIPS64 memory map is divided into different regions depending on i) whether the target address requires TLB for translation, ii) the current mode of execution (kernel or user), and iii) whether the address is cached or not. Also there was an additional Cavium specific memory segment called CVMSEG, whose implementation was inevitable for the correct execution of guest system. The guest uses it as a scratch pad memory during context switching. In Cavium Linux system the memory management is done through TLB which provides virtual to physical address translation. TLB is managed by the kernel with four basic assembly instructions. i) tlbp probes the TLB for a entry, ii) tlbr reads the found entry, iii) tlbwr writes the TLB at random index, and iv) tlbwi write the entry on a particular index. HTTM implements the required functionality by making the intended changes in guest’s TLB against these instructions. As this is a virtualized environment, guest TLB provides a mapping from guest virtual address (GVA) to host virtual address (HVA) so that guest can load or store to host actual memory. For doing so, we have to provide a two level of translation i.e., GVA-to-GPA and GPA-to-HVA. In case of invalid address access or any other error, relevant exceptions are generated by the hypervisor and dispatched to the guest.

2.3. Timing infrastructure
Linux kernel keeps track of the time by receiving a periodic timer interrupt that provides the basic framework for the time synchronization in the OS. Correct implementation is crucial for a stable guest system. In native system, simple Count-Compare timing mechanism is implemented in which count register is incremented every clock cycle. Due to the fact that our host system is also Cavium Octeon we use host count register to calculate the current time for guest. When the guest reads its count register, HTTM intercept the call and in action reads the host count register and write to guest count register after converting the value according to the guest clock. In case if the guest system needs a timer interrupt, HTTM registers a timer of required duration with the host. Upon timer expiration, host sends a SIGALRM signal and HTTM notifies the guest by sending an interrupt.

2.4. Interrupt and exceptions
Interrupts originates from an external source and occur asynchronously. Cavium Octeon system has Central Interrupt Unit (CIU) which dispatches the external and inter-core interrupts to the destined core. Implementation of CIU was critical due to the fact that it is involved in interrupt handling and proper working of multiple cores. Exception is any event that can cause a change in the execution flow that can be programmed or un-programmed too. In case of programmed exception, control shifting instructions of guest are replaced with instructions that shift the control to HTTM during DBT. During execution control is shifted back to HTTM which handles the case accordingly. While during un-programmed exceptions, specific guest registers are set with the current status information and control is shifted towards guest’s exception table. At the end of exception service routine eret instruction occurs, on which HTTM directs the guest back to its normal execution by restoring its PC value.

2.5. I/O devices
Implementation of I/O devices is one the most important feature in the virtual guest. We have added the support for UART, disk, and networking in HTTM. Following is brief description of these devices.

2.5.1. UART. Console is a basic necessity of any virtual system and for that purpose UART implementation is required at a very early development stage. UART provides serial communication of characters to and from different cores. UART implemented in HTTM consists of two threads: i) Receiver thread that continuously checks for the availability of input characters from the keyboard and whenever input is available, it generates an interrupt for a particular core through CIU. ii) Transmitter
thread, which helps processor in printing all the messages on the console. After transmitting data, it notifies the processor through CIU that UART is free for transmission.

2.5.2. Network. Communication with the external world is an important part of MIPS based devices. Instead of following the typical inefficient approach of device emulation, HTTM employs Virtio API [9] and Vhost_net module to implement an efficient network virtualization solution. Virtio_net driver acts as the front-end driver in the guest and can easily be configured in the guest Linux kernel as MMIO (Memory Mapped Input Output). Being MMIO device it doesn’t require PCI bus for communication. Vhost_net provides the emulation of network card in the host kernel. Although, Vhost_net does not emulate a complete Virtio PCI adapter, instead it restricts itself to Virt-queues operations only. These two drivers are connected through network device in HTTM. It manages bi-directional control signals between Virtio_net and Vhost_net using device specific registers, e.g., VIRTIO_MMIO_STATUS, which are used by the guest driver for Virt-queues sharing and device controlling.

2.5.3. Disk. The disk is one of the key requirements of any virtual system to enable data persistence. In general, Virtio and Vhost Linux drivers can be used for both character and block devices, (e.g., console and disk respectively). Following the footsteps from network implementation, we have used Virtio_Blk driver for disk. A simple raw file is created on host system through which data can be transferred from host to guest and act as a disk for guest. Virtio_Blk is also a MMIO device similar to Virtio_net and thus do not rely on PCI bus. For making Virtio_Blk front-end driver for virtual disk in the guest, it needs to be explicitly configured. Virtual disk file is passed to Virtio_Blk by the backend driver Vhost_Blk. Note that Vhost_Blk is not the part of the standard Linux kernel. For our purpose, we choose the basic implementation of the Vhost_Blk from [10], and modified it for the Cavium Octeon MIPS64 kernel. Once this module is specifically built, it is loaded in the host kernel. To enable the communication between front-end and backend drivers, Virtual Block Device was implemented which passes the control signals in both directions. It notifies the host kernel when data is provided by the guest or sending interrupt to the guest when Vhost_Blk completes its assigned task.

3. Performance Evaluation

Further we present functional correctness of the single-core guest system followed by the performance evaluation using standard benchmarking tool. Here the native system refers to the real target system and guest is the virtualized system.

3.1. Linux testing project

To verify that the virtual guest system created by HTTM is functionally correct, we rely on Linux Testing Project (LTP) [11]. We successfully tested various subsystems including memory, file system, scheduling, Interprocess communication (IPC), timer, system calls, and network. Some of the tests include ftest(01, 02), mem(01, 02), trace_schedule, shmem_test_(01, 07), signal_test_(03,04), alarm(01-07), mmap09, vfork(01-02) and pingpong.

3.2. LMbench

Next, we rely on LMbench [12] which is a benchmarking tool for performance evaluation that contains different types of tests. These tests were performed on native hardware and virtual guest system. We define “overhead” as the ratio of native and guest system data transfer rates. It highlights the performance penalty due to the virtualized environment of HTTM. Table 1 shows the test results of different LMbench tests. lat_ops measures the latency of basic CPU operations, such as add, mul, bit (XOR) etc. For memory bandwidth measurements, we executed bw_mem tests for basic operations including read, write, and copy. These tests were performed with 4MB memory size. For network testing, we performed bw_tcp test that measures TCP bandwidth between a server on the guest and a client on an external system. In general, we found that virtualization overhead varies from 5.5 to 24.11
in different tests. We note that the overhead in different design units of HTTM is varying, e.g., ISA translation for guest instructions takes a considerable amount of time.

Table 1. Performance results of LMbench.

|          | int64 bit (nsec) | int64 add (nsec) | int64 mul (nsec) | bw_mem rd (Mb/sec) | bw_mem wr (Mb/sec) | bw_mem cp (Mb/sec) | bw_tcp (Mb/sec) |
|----------|------------------|------------------|------------------|-------------------|-------------------|-------------------|----------------|
| Native   | 1.25             | 1.31             | 0.50             | 821.45            | 2918.56           | 718.97            | 11.60          |
| Guest    | 12.24            | 31.59            | 5.33             | 116.00            | 128.81            | 65.44             | 2.10           |
| Overhead | 9.79             | 24.11            | 10.66            | 7.08              | 22.66             | 10.99             | 5.50           |

For detailed information we refer to working demonstration of HTTM and documentation available at [13]. This research work is funded by National ICT R&D Fund, Pakistan under the project “Development of Type-2 Hypervisor for MIPS64 Based Systems”.

4. Conclusion
In this paper, we present a design and implementation of full-virtualized Type-2 hypervisor for MIPS64 based high-end devices. We present a brief description of its software architecture and brief detail of each unit. Finally we tested the guest system created by hypervisor using LTP. In future we plan to analyse performance of HTTM in detail and optimize it for real-time applications to demonstrate real-time efficacy.

References
[1] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I and Warfield A 2003 Xen and the art of virtualization ACM SIGOPS Operating Systems Review 37 164
[2] Vmware VMware Workstation 2013 [Online] Available: www.vmware.com/products/player/
[3] Cox L P and Chen P M 2007 Pocket Hypervisors: Opportunities and Challenges Proc. of the 8th IEEE Workshop on Mobile Computing Systems and Applications (Tucson, Arizona: IEEE) p46
[4] Aguiar A and Hessel F 2010 Embedded systems virtualization: The next challenge? 21st IEEE Int'l Symp. on Rapid System Prototyping (RSP) (Virginia, USA: IEEE) p1
[5] Heiser G 2011 Virtualizing embedded systems: why bother? Proc. of the 48th Design Automation Conference (NewYork, USA: ACM) p901
[6] Heiser G 2009 The Motorola Evoke QA4-A Case Study in Mobile Virtualization
[7] Barr K, Bungale P, Deasy S, Gyuris V, Hung P, Newell C, Tuch H and Zoppis B 2010 The VMware mobile virtualization platform: is that a hypervisor in your pocket? ACM SIGOPS Operating Systems Review 44 124
[8] Gu Z, Wang Z, Li S and Cai H 2012 Design and implementation of an automotive telematics gateway based on virtualization 15th IEEE Int'l Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (China: IEEE) p53
[9] Jones M T, 2010 Virtio: An I/O virtualization framework for Linux. IBM developerWorks.
[10] Pulavarty B 2010 vhost-blk implementation [Online] Available: http://lwn.net/Articles/379864/
[11] Larson P 2002 Testing Linux® with the Linux Test Project Ottawa Linux Symp. (Ottawa, Canada) p265
[12] Mcvoy L W and Staelin C 1996 Lmbench: Portable Tools for Performance Analysis USENIX Annual technical Conf. (San Diego, USA: useunix) p279
[13] Type-2 Hypervisor for MIPS64 based Systems [Online] Available: http://www.kics.edu.pk/project/downloads/53