# CASL Hypervisor and its Virtualization Platform

Chien-Te Liu, Kuan-Chung Chen and Chung-Ho Chen
Dept. of Electrical Engineering and Inst. of Computer & Communication Engineering
National Cheng Kung University
Tainan, Taiwan
{ufoderek,edi}@casmail.ee.ncku.edu.tw, chchen@mail.ncku.edu.tw

*Abstract*—**In this paper, we present an ARM-based hardware-assisted hypervisor, named CASL-Hypervisor, and a full system virtualization platform developed in SystemC which enables software/hardware co-simulation of virtual machine systems. CASL-Hypervisor takes advantage of an additional processor mode, extended memory management unit, configurable hardware traps and specialized hardware devices to virtualize unmodified Linux-based guest operating systems. By utilizing hardware extensions, development effort of CASL-Hypervisor can be greatly reduced and the hypervisor has achieved relatively low virtualization overhead. Evaluation is demonstrated on an approximately-timed manner so it is able to do fast software/hardware co-simulation and evaluations. We use the ARM-v7A instruction set simulator as the host processor. The hypervisor overhead can be quantified through instruction count ratio of guest operating system to the hypervisor. The results show that CASL-Hypervisor successfully virtualizes four guest operating systems with about 9.78% overhead.**

***Keywords—instruction set simulator; full system simulation; full virtualization; hypervisor;***

## I. INTRODUCTION

System virtualization has become an important technology in various applications. Most of the state-of-the-art processor architectures have been extended to support virtualization [1]. However, not all processor architectures are designed to be virtualizable and thus several special techniques such as binary translation and paravirtualization are used to implement virtual machine monitor on these processors [2][3][4].

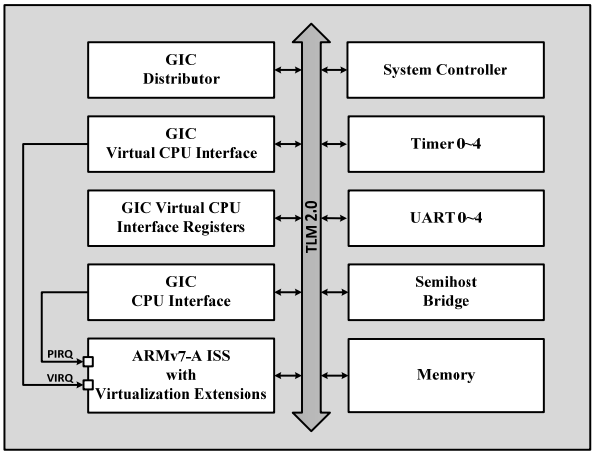


Figure 1. Full system virtualization platform architecture

In binary translation, sequences of source instructions are translated into the target instruction set and the source instruction set does not need to be the same as the target instruction set. The translation process can be achieved statically, dynamically, or in a combined fashion [2]. Paravirtualization is a virtualization technique [5] that the hypervisor presents a set of predefined virtual machine interfaces to the guest operating systems. The guest operating system has to use these interfaces to co-work with the hypervisor to perform the privileged work. Paravirtualization usually needs source-level modifications to the guest operating system, i.e., patching the existing operating systems. While the operating system must be ported to run in a virtual machine, most normal applications can run unmodified.

In this paper, we implement an instruction set simulator based on ARM-v7A [7], supporting hardware virtualization extension, and the full system simulation platform as shown in Fig. 1. We also propose our hypervisor, CASL-Hypervisor, running on the simulation platform and use it to manager four unmodified guest Linux operating systems in a round-robin scheme. The most challenging part of the hypervisor design is to virtualize interrupts and I/O operations. With the help of extended interrupt controller, CASL-Hypervisor can maintain physical and virtual interrupts at the same time with succinct implementation. Moreover, the two-stage memory management unit greatly reduces the I/O virtualization overhead of guest operating systems. Besides, the round-robin scheduler of CASL-Hypervisor is idle-thread-sensitive which means that it can detect the state of guest kernel's scheduler to do appropriate scheduling.

The rest of this paper is organized as follows: In Section II, we introduce our full system simulation platform and the hardware extension for virtualization. Section III discusses our hypervisor architecture. Section IV presents the evaluation results and we have brief conclusions in Section V.

## II. FULL SYSTEM VIRTUALIZATION PLATFORM ARCHITECTURE

To develop and verify CASL-Hypervisor, we implemented a full system virtualization platform using SystemC library. As shown in Fig. 1, the full system virtualization platform consists of an instruction set simulator, a system controller, several timers and UARTs, all of which are necessary to successfully boot operating system and all modules are connected via a transaction level modeling bus.

### A. ARMv7A Instruction Set Simulator

An instruction set simulator (ISS) is a computer program which mimics the behavior of a target processor. The instruction set simulator implemented in this paper is based on ARMv7A architecture with virtualization extensions and its correctness has been verified by successfully booting a Linux kernel with an initial ram disk. Following are the ARMv7A virtualization extension functions implemented in our instruction set simulator.

#### 1) Privilege Levels and Processor Modes:

The ARM architecture has three privilege levels (PL) from privilege level 0 to privilege level 2. The User mode is the least privilege mode while Supervisor mode, Abort mode, IRQ mode, FIQ mode, and Undefined mode have higher privilege level than the User mode. These modes exist in prior ARM architecture. The ARM virtualization extension introduces the HYP mode which has the highest privilege and is designed for the use of hypervisor. Operating system usually executes under Supervisor mode while applications run under User mode. When ARM processor gets into the privilege operations, such as WFI (wait for interrupt), the processor changes its mode to the HYP mode and executes the hypervisor code.

#### 2) Exceptions:

There are eight exceptions defined in ARMv7A architecture as shown in Table I. The new exception type, HYP trap, has its own interrupt vector address. The operating system or the hypervisor must provide interrupt service routines in advance in order to resolve processor exceptions.

TABLE I. ARMV7A EXCEPTIONS

| Exception Type | Mode | High Vector Address | Low Vector Address |
|---|---|---|---|
| Undefined | Undefined | 0x00000004 | 0xFFFF0004 |
| Software Interrupt | Supervisor | 0x00000008 | 0xFFFF0008 |
| Prefetch Abort | Abort | 0x0000000C | 0xFFFF000C |
| HYP Trap | HYP | 0x00000010 | 0xFFFF0010 |
| Data Abort | Abort | 0x00000014 | 0xFFFF0014 |
| IRQ | IRQ | 0x00000018 | 0xFFFF0018 |
| FIQ | FIQ | 0x0000001C | 0xFFFF001C |

#### 3) Hyperviosr Traps:

For better efficiency on processor virtualization, the ARM processor with virtualization extensions supports trapping certain privileged operations into the HYP mode. For instance, coprocessor access and WFI instruction can be configured to be trapped to the HYP mode. The error code is saved to the HYP syndrome register (HSR) so that when a hardware trap is generated, the hypervisor can read and decode the error code inside the HSR to perform the corresponding operations. For the exceptions which are configured to be routed to the HYP mode, the return address is saved to the exception return register (ELR) and then hypervisor can use the exception return instruction (ERET) for fast exception return.

Same as a hypervisor trap, the corresponding error code is saved to the HSR when exceptions are routed to the HYP mode so that the hypervisor can read and decode the HSR to do the corresponding service. The error codes are listed in Table II.

TABLE II. ERROR CODES OF HSR

| Error Code | Exception Class |
|---|---|
| 0x01 | Trapped WFI or WFE instruction |
| 0x03 | Trapped MCR or MRC access to CP15 |
| 0x04 | MCRR or MRRC access to CP15 |
| 0x05 | Trapped MCR or MRC access to CP14 |
| 0x07 | Trapped access to CP0-CP13 |
| 0x08 | Trapped access to CP0-CP13 |
| 0x11 | Supervisor call exception routed to HYP mode |
| 0x12 | Hypervisor call |
| 0x20 | Prefetch abort routed to HYP mode |
| 0x21 | Prefetch abort taken from HYP mode |
| 0x24 | Data abort routed to HYP mode |
| 0x25 | Data abort taken from HYP mode |

### B. Virtual Memory System Architecture

In ARMv7A architecture, the address translation, accessing permission, attribution determination and checking are controlled by the memory management unit. With ARM virtualization extensions, the memory management unit (MMU) can support two stages of virtual address translation and the page table descriptors can be in either 32-bit short format or 64-bit long format.

#### 1) Two stage virtual address translation

The ARMv7A architecture with virtualization extensions provides multiples stages of memory system control. Memory system control of each stage is provided by MMU and each MMU has its own independent set of controls. Our ISS has implemented the following MMUs:

- Non-secure PL2 stage 1 MMU
- Non-secure PL1&0 stage 1 MMU
- Non-secure PL1&0 stage 2 MMU

## III. CASL-HYPERVISOR ARCHITECTURE

CASL-Hypervisor is a virtual machine monitor designed for ARM architecture; it can virtualize ARM Linux without any source-level modifications. CASL-Hypervisor exploits virtualization extensions supported in ARMv7A architecture to run multiple guest operating systems efficiently at the same time.

### A. CPU Virtualization

To manage processes, the operating system (PL1 supervisor mode) has to transfer the control of the processor to a user process (PL0 user mode) and reclaim the control later. The CASL-Hypervisor uses a similar approach as used in a normal operating system. To support full virtualization, CASL-Hypervisor runs under PL2 HYP mode with the guest operating system running under the PL1 supervisor mode. When performing virtual machine switches, the hypervisor transfers control to the guest operating system using ERET instruction to change the processor mode and program counter simultaneously. Since the CASL-Hypervisor uses full virtualization, only hypervisor traps, physical interrupts, and routed data aborts can reclaim the control from the guest operuating systems.

This work was sponsored and supported in part by the National Science Council of Taiwan under grand NSC 101-2220-E-006-002-

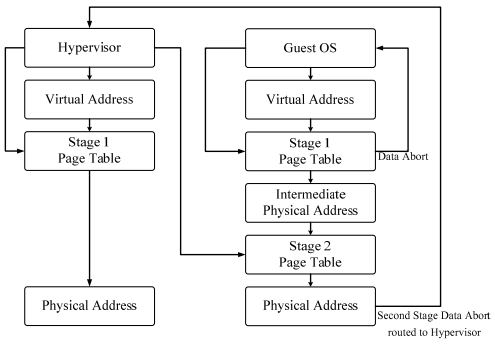


Figure 2. Two stage page table walks

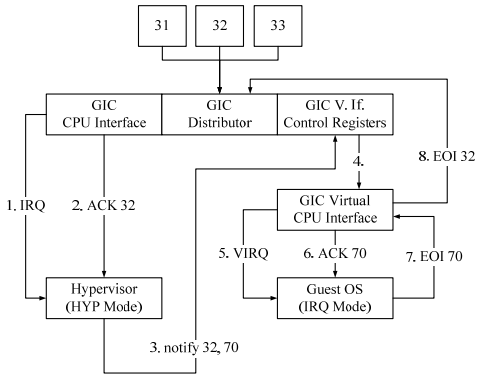


Figure 3. Interrupt in virtulized environment

### B. *Memory Virtualization*

There are three page tables used in the translation process under the virtualization environment as shown in Fig. 2. The fully virtualized guest operating system retains the control of its own stage-1 page table. This table translates a virtual address to the intermediate physical address (IPA). The IPA is then translated via the stage-2 page table controlled by the hypervisor. The translation process is achieved using hardware only so the performance overhead is minimal. It is noted that the guest operating systems are not aware of the existence of the stage-2 page table.

In order to allocate memory to each guest operating system, CASL-Hypervisor divides the memory into aligned and consecutive 4 KB pages. Since ARM Linux also uses aligned 4 KB pages as a basic block in its virtual memory system, it is reasonable for the hypervisor to use the same page size to do memory management and achieve better space efficiency with low performance overhead. We also implement an on-demand memory allocation scheme to achieve better memory space usage. The access to the unallocated memory region by a guest operating system is routed to the hypervisor via a data abort exception. The hypervisor decodes the destination address and if the address is in the range of a virtual machine's main memory mapping, the hypervisor allocates an unused page and allows the virtual machine to finish the operation. In addition, subsequent accesses to that memory region can be performed directly without interferences from the hypervisor. With on-demand memory allocation, CASL-Hypervisor can have fine-grained memory management while not losing performance. Fig. 2 illustrates the table walk of hypervisor and guest operating system.

### C. *Interrupt Virtualization*

In a virtualization environment, the hypervisor has to trap the physical interrupt and then distribute the interrupt signal to a guest operating system. With the virtualization extension supported by the generic interrupt controller (GIC), CASL-Hypervisor traps the physical interrupt signal sent from peripheral devices and then passes it to the guest operating systems through the GIC virtual interface. Fig. 3 has the detailed interrupt processing sequence.

In Fig. 3, assuming interrupt (ID32) is forwarded to the processor and a physical IRQ exception is routed to the hypervisor. The hypervisor reads interrupt acknowledgement (ACK32) from the GIC CPU interface. After getting the interrupt information, the hypervisor transfers the interrupt to the GIC virtual interface register (notify 32, 70). The notification has pairing information of the physical interrupt ID and the virtual interrupt ID. Based on the information, the GIC virtual interface register, GIC virtual CPU Interface forwards the interrupts to the processor. The processor generates a virtual IRQ exception to trigger the guest operating system that enters its interrupt service routine (ISR). After executing the ISR, the guest operating system writes end of interrupt (EOI 70) to the GIC virtual CPU interface. Finally, the GIC distributor changes the state of the finished physical interrupt (EOI 32). Through this manner, CASL-Hypervisor can also simulate virtual devices and generate the virtual interrupts sent to the guest operating systems.

### D. *Virutal Machine Scheduling*

The virtual machine switching process involves of storing and restoring virtual machine context to and from the memory. Virtual machine context contains essential execution states including registers values, coprocessor states, page tables, interrupt states, virtual devices' states and scheduling information.

The scheduler of CASL-Hypervisor uses a round-robin mechanism to switch between virtual machines. Each virtual machine has the same priority and fixed execution time. When a virtual machine is resumed from the inactive queue, the scheduler resets the system timer and restores the virtual machine's context so that the guest operating system can continue its execution. The system timer generates an interrupt after a fixed period of time to let the hypervisor regain the control; current configuration uses 30 ms as the default virtual machine switch interval. In addition, Linux has a special kernel thread called idle thread which runs only when no other runnable processes are available. The idle thread of ARM Linux contains the WFI instruction. If a virtual machine issues a WFI instruction, the scheduler removes the virtual machine from the scheduling queue. The virtual machine will be rescheduled only when an interrupt for that virtual machine comes.

## IV. EVALUATION RESULT

This section describes the performance overhead of the CASL-Hypervisor when booting Linux OS and running MiBench test suite applications [8]. The processor utilization rate of the hypervisor is calculated based on the percentage of the total number of instructions used by the hypervisor. The

target architecture for simulation is ARMv7A ISS with the detailed parameters listed in Table III. Furthermore, we exploit Newlib as the C standard library for developing CASL-Hypervisor.

TABLE III. HYPERVISOR SYSTEM

| Platform Component | Configuration |
|---|---|
| Processor Model | ARMv7-A ISS |
| Processor Frequency | 1000 MHz |
| Device Frequency | 400 MHz |
| Memory | 1024 GB |
| Memory per Guest OS | 128 MB |
| Virtual Machine Switch interval | 30 ms |
| Guest OS Context Switch Interval | 2.5 ms |
| Guest OS version | Linux 2.6.38 |
| Busybox Version | 1.19.4 |
| Newlib Version | 1.19.0 |

### *A. Booting Linux OS*

Fig. 4 shows the processor utilization rate of hypervisor during four guest operating systems booting. Beginning of the graph, the hypervisor is preparing Linux kernels and initializes the ram-disks, which is built using Busybox, for booting. Center of the graph have different utilization rates due to access to the virtual devices, page allocations, and trapped coprocessor operations. Finally, four guest operating systems have successfully booted and enter the idle thread. When all guest operating systems are idle, the hypervisor uses 93% of processor time only to switch in between them. The result shows that CASL-Hypervisor spends about 9.78% overhead in exception trapping and virtual machine scheduling. Because of the round-robin scheduling policy, we can find out that the guest operating system occupied the processor regularly.

### *B. Executing MiBench*

After booting Linux operating system, we execute the MiBench applications. In Fig. 5, we use the same emulation environment to run the applications on the native Linux and serve as the comparison basis. It is obviously to find out that the execution time is almost linear increasing from native execution to 4 guest operating systems scheduled by the CASL-hypervisor because each operating system runs the same applications on a single core platform. This also demonstrates that our hypervisor has little overhead when scheduling four guest operating systems by exploiting the virtualization extension supported in ARMv7A processor.

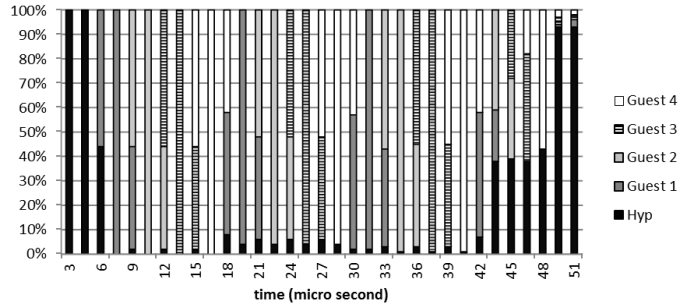


Figure 4. CPU Utilization Rate of CASL-Hypervisor

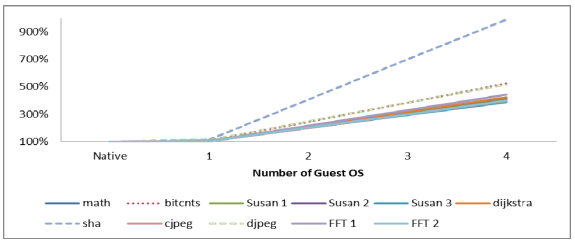


Figure 5. MiBench execution result

CASL-Hypervisor is developed with virtualization extension; the guest operating systems can directly access their own memory pages through the MMU without hypervisor intervention after the hypervisor has assigned the page to them. In this situation, the main overhead of our hypervisor is from trapping the physical interrupt signal whenever the physical interrupt occurs

## V. CONCLUSIONS

We have presented and demonstrated an ARM-based CASL-Hypervisor and its full system simulation platform. With the help of an additional processor mode, extended memory management unit, configurable hardware traps and specialized hardware devices, CASL-Hypervisor is able to virtualize multiple guest operating systems using full virtualization method and at the same time keep the virtualization overhead low.

In addition, a hardware simulation model cannot only be built fast but also easily because the full system virtualization platform is based on SystemC.

## REFERENCES


[1] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uuhlig, R. “Intel virtualization technology: Hardware support for efficient virtualization.” In IntelTechonolgy Journal 10, 3 August 2006, pp. 167-177.

[2] M. Reshadi, P. Mishara, and N. Dutt, “Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation,” ACM Transactions on Embedded Computer Systems, Vol. 8, No. 3, pp. 20-27, April 2009.

[3] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, “Binary Translation,” Communications of the ACM, Vol. 36, No. 2, pp. 68-81, February 1993.

[4] M. Reshadi, P. Mishara, and N. Dutt, “Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation,” Proceedings of the 40th ACM/IEEE Design Automation Conference (DAC‟03), Vol. 8, No. 3, pp.758-763, Anaheim, CA, USA, June 2003.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. “Xen and the art of virtualization.” In 19th ACM Symposium on Operating Systems Principles, Oct 2003.

[6] Whitaker A, Shaw M, Gribble S D. “Denali: Lightweight Virtual Machines for Distributed and Networked Applications.” University of Washington Technical Report 02-02-012002.

[7] “ARMv7A Technical Reference Manual DDI-0406C,” ARM Co. Ltd. November 2011.

[8] M. R. Guthaus, et al., “MiBench: a Free, Commercially Representative Embedded Benchmark Suite,” Proceedings of the 2008 IEEE International Workshop on Workload Characterization (WWC‟01), Austin, TX, USA, December 2001.