Development of Type-2 Hypervisor for MIPS64 Based Systems

December

2015

[10th Deliverable]

This document is version 10 of first report and includes the implementation details of current deliverable of "Development of Type 2 Hypervisor for MIPS64 based Systems" project, funded by National ICT R & D Fund Pakistan. The 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 current 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

TABLE OF CONTENTS

TABLE OF FIGURES

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 development started on April 1, 2013 and first deliverable was due after 3.5 months i.e. July 15, 2013. In first deliverable, we built the required infrastructure. The infrastructure printed guest kernel banner on console at the end of 1st deliverable. Second deliverable was due after 6.5 months of commencement data i.e. October 15, 2013. The milestone in 2nd deliverable was the dynamic code patching of one sensitive guest instruction with one safer instruction. In 3rd deliverable, dynamic code patching was augmented by implementing cases where one sensitive instruction is replaced by more than one instruction. In 4th deliverable, dynamic code patching was applied on demand. In 5th deliverable, guest kernel booting completes and starts creating user mode processes. In 6th deliverable, SMP support was added to whole infrastructure and many performance related bugs were fixed. In 7th deliverable major units like Timer and UART were added, in code. Exception and interrupt handling mechanism was further developed. Bugs related to memory implementation and instruction execution were fixed. In 8th deliverable, virtual disk implementation was added to the infrastructure. Also Ethernet card detection inside the guest has been done. There were some bug fixes related to timer, TLB and exception handling mechanism. In 9th deliverable, a valid IP was assigned to the guest. Timer infrastructure was changed due to bad performance. Many TLB and UART related bugs were fixed. An error in block fetching mechanism was identified and corrected. Some code structural changes were done for improving timing performance of hypervisor. Ctrl+C implementation was provided by hypervisor for killing guest processes. In 10th deliverable, a complete network infrastructure is implemented. In the next three month extension time period, optimization has been in focus. Numbers of strategies were implemented to achieve better time performance.

2 HIGH LEVEL DESIGN

Type-2 hypervisor behaves like an ordinary Linux process that could be scheduled by host operating system. However, this process has to present a holistic view of virtual hardware for guest operating system(s) to run on it. Virtual hardware consists of software representations of CPU cores, memory and peripheral devices. In real hardware, CPU cores and devices work concurrently and could be considered as processes or threads in software representation. Multiprocessing requires inter-process communication (IPC) whereas multithreading could be implemented using the shared address space. Each one has its own pros and cons. We selected multithreaded design for our hypervisor, as shown in Figure 1. It shows that each core and device is a separate thread. Central interrupt unit (CIU) is another thread that dispatches pending interrupts to the cores using mapped memory.

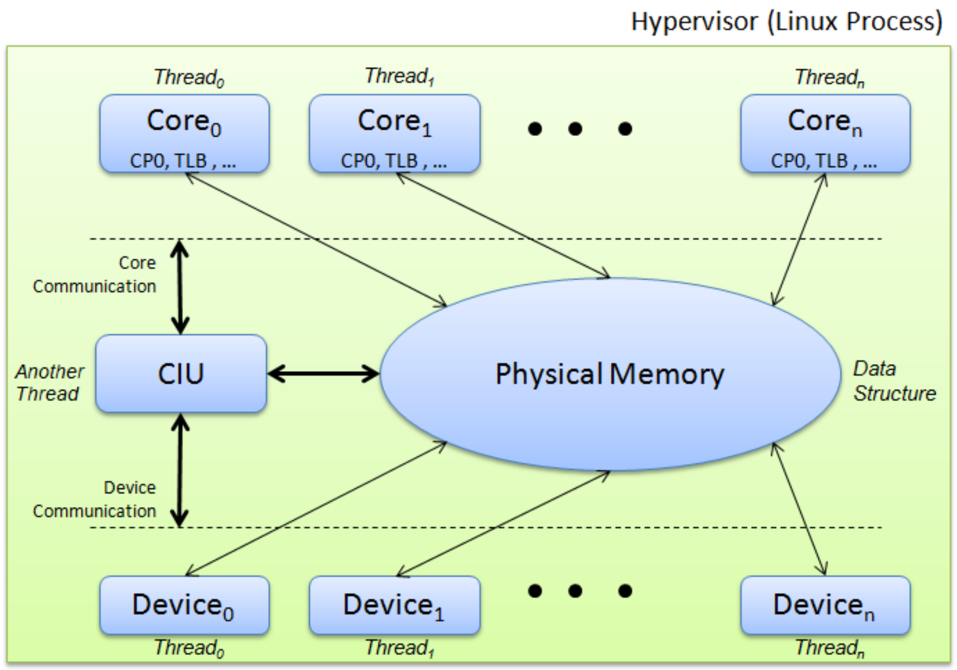


FIGURE 1: Multithreaded design of Type-2 hypervisor

3 IMPLEMENTATION STRATEGIES

Primarily, we have experimentally implemented two different strategies to develop Type-II hypervisor. Firstly, we implemented an instruction level strategy. This strategy is very simple and easier to implement but it greatly reduces time efficiency in order to boot a guest OS because we take trap and then emulate every instruction of the guest OS. It also demands a lot of programming effort because we have to provide almost all MIPS ISA functionality implementation in our hypervisor code. Secondly, on the other hand, we also implemented a block level technique for the execution of guest OS. Because in this strategy, we fetch and translate a set of instructions at a time instead of a single instruction emulation that's why it can be considered a better and faster approach form the previous strategy. We discussed both implementations as follows.

3.1 INSTRUCTION LEVEL EXECUTION MODEL

It is a very simple mechanism to execute a Guest executable binary on the hypervisor. In this strategy, when executable guest OS is loaded then we patch all instructions of the guest OS binary with a trap-call instruction and original instructions are placed into a lookup table (hash-table). Patching means an instruction is replaced with another instruction, which is caused to generate a trap during its execution. By doing so, when the control is shifted on the guest application binary for its execution then we get a signal from the hardware on each instruction because of its patching. This signal is catch by the signalHandler into the hypervisor code, a method which is able to catch signal generated by the hardware. Now the control comes back to our hypervisor code and we can emulate the corresponding instruction into our software based environment also called a virtual environment. In software based environment, we actually have a complete soft image of MIPS's processor.

We have all GP (General Purpose) registers, CP0 registers, TLB, CIU and exception handling mechanism in our software based environment, which is provided by the hypervisor to the guest operating systems for their execution.

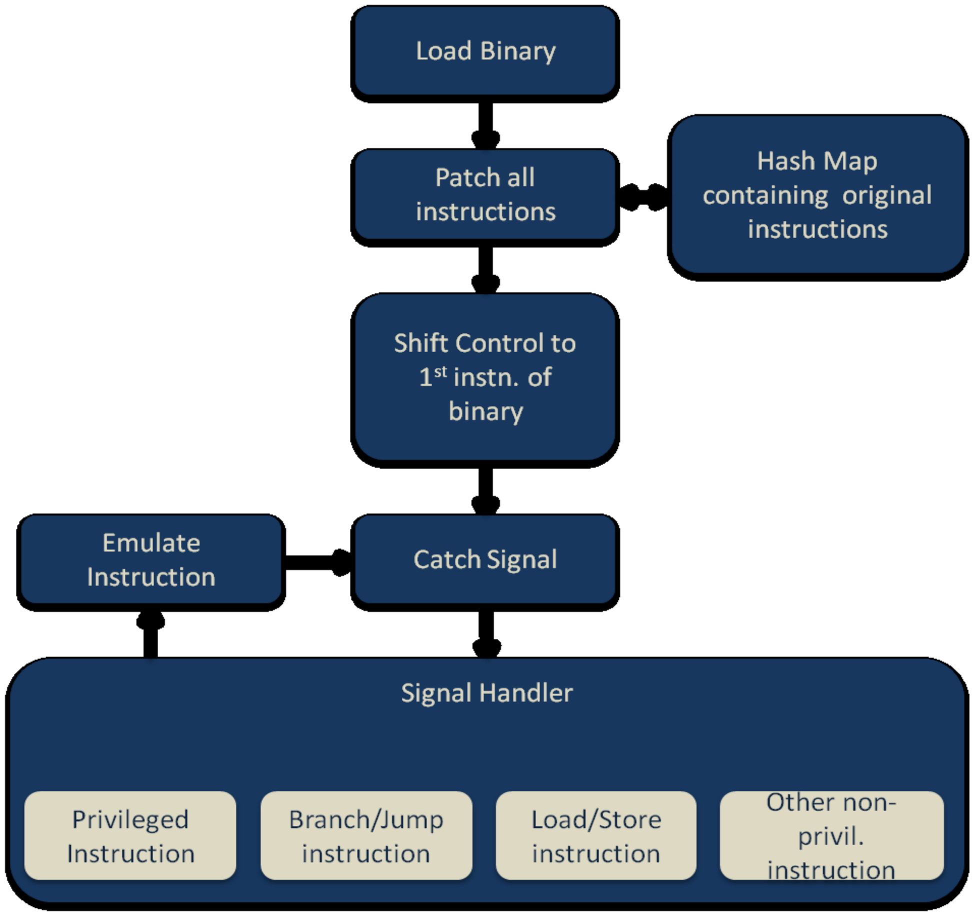


FIGURE 2: Instruction level execution model

3.2 Block Level Execution Model

This strategy is very much different from the above mentioned strategy because it provides block level instruction emulation instead of instruction level trap & emulation mechanism. A block consists of a set of instructions having only one jump/branch instruction as a second last instruction. In this strategy, we actually fetch a block from a corresponding PC address of the guest OS loaded binary and then translate this block. Each instruction in the block fetched from Guest binary is translated into a number of instructions, which are executable on the host machine. The translated instructions are composed in the same order as the original instructions are given in the Guest binary block to preserve the correct behavior of instructions. The translated block is then placed into a Cache-blocks for future reusing purpose so that if the same block is required again then no need to fetch again and retranslate that block if it's available in the Cache-blocks. After block translation we execute it by just placing the starting address of this block into PC register so that this block can be executed.

During execution a block, the control may come into hypervisor code in case of address translation from guest virtual address to host virtual address or any other interrupts or for log files writing purpose. The first block in fetched from the guest virtual reset vector 0xBFC00000 address and then it is translated and after its translation, it is placed into some fixed location memory where control is then shifted at that memory location for the execution of translated block. When whole the execution of translated block is completed, the control comes back into the hypervisor code for fetching the next block. Now, we first check whether the new required block is available into Cache-blocks or not. If it is found in the Cache-blocks then it doesn't need to fetch again and retranslate the required block because the Cache-blocks already have this translated block into it. The control is just placed on the new block, which has been found into the Cache-blocks. Alternatively, if required new block doesn't found into the Cache-blocks then it is fetched, translated and also cached into the Cache-blocks. The Cache-blocks may contain some fixed number of translated blocks into it when it becomes full then one block from it, is replaced by new translated block based on some replacement policy. Figure 3 shows the flow chart of block level execution model of hypervisor.

4 System Development

The whole infrastructure of the hypervisor is divided into modules. The implementation functional description of each unit is given detail below.

4.1 Memory Management Unit (MMU)

It is the most important unit of a computer system. The purpose of memory management unit is to translate virtual addresses to physical addresses. For virtual address translation, some rules are already defined by physical hardware and we implemented these rules in software to provide the virtualization of MMU used by guest operating system(s). In case of hypervisor, it is used to translate GVA to HVA. To translate GVA to GPA, we use same method as used by the hardware. For translation of GPA to HVA, we use hash map to store information of all regions mapped in host virtual address space.

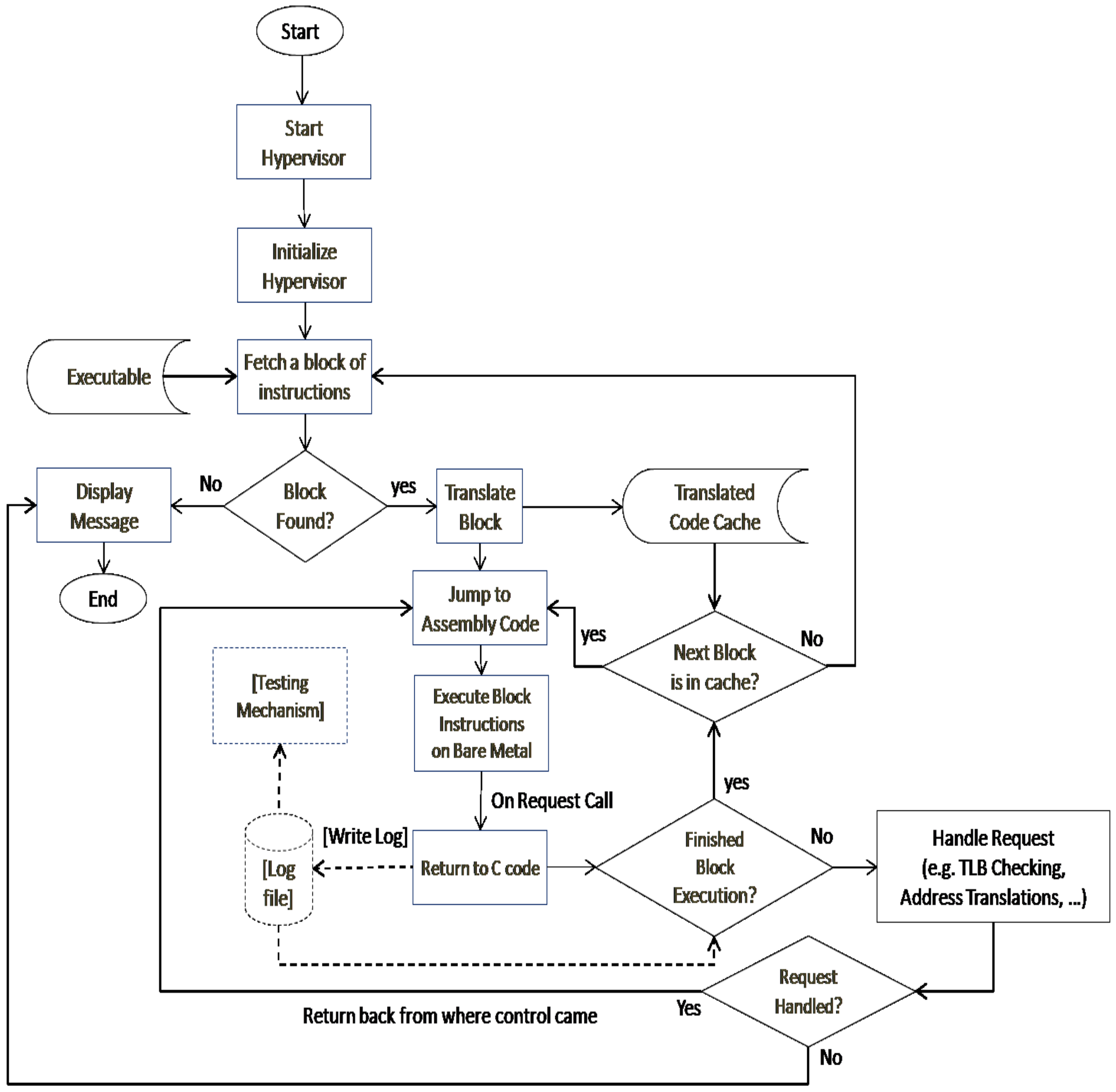


FIGURE 3: Block level execution model

4.1.1 GVA to GPA Translation

MIPS64 architecture supports both 32-bit and 64-bit Addressing modes. In 32-bit addressing mode, address segment is defined by upper 3 bits (i.e. bits 32-29) of virtual address. If these bits are 100 then it is kseg0 region. It is directly mapped to physical memory. If these bits are 101, address is from kseg1 region and this is also directly mapped to physical memory. In both previous cases, lower 20 bits represent physical address. For 110, region is ksseg. This is not directly mapped and we have to search for it in TLB for address translation. For 111, region is kseg3 which is not directly mapped and we have to search TLB for valid entry to translate the address. If these bits are 0xx then it is useg. Translation for useg is slightly different. If ERL bit of status register of CP0 is set then useg is directly mapped to physical memory. If ERL bit is not set then we have to check TLB to get physical address.

In 64-bit addressing mode, address segment is defined by upper 2 bits (i.e. bits 63-62) of virtual address. If these bits are 10, then this is xkphys region which is directly mapped to physical memory or I/O devices. If 49th bit of virtual address is 0 then it is memory access and lower 29 bits represent physical address of memory. If 49th bit is 1 then it is I/0 address and data is load/store from respective device. If these bits are 11 then it is xkseg region which isn't directly mapped and we have to search TLB for valid address translation. For 01, region is xsseg which is also to be searched in TLB for translation. For 00, region is xuseg. If ERL bit of status register of CP0 is set then it is directly mapped otherwise TLB translation would be required.

4.1.2 GPA to HVA Translation

All physical memory regions of a machine are mapped in virtual address space of hypervisor. Once we get the valid translation for GVA, we have to translate that physical address to HVA in order to access valid data. After getting valid physical address, we found the memory region or I/O device to which it belongs. We simply find HVA for required memory region or I/O device using hashmap. Once we get a valid GVA-to-HVA translation, we can simply execute the respective instruction.

4.1.3 Page Table

In MIPS no physical page table is provided by hardware and page table is solely managed by operating system. Hence, there is no need to implement page table.

4.1.4 Translation Look-aside Buffer (TLB)

TLB is a cache used to speedup virtual address to physical address translation. In case of type 2 hypervisor, TLB translates GVA to HVA. There are four basic TLB functions: probe, read, write-random and write-index. TLB probe searches for a TLB entry using the value of EntryHi register of co-processor 0 (CP0). If valid entry is found, it places index of TLB entry in CP0 index register, otherwise it sets probe bit of index register and consult page table. TLB read gets value from CP0 index register and checks the validity of data at this index. If data is valid, the components of entry (i.e. entryHi, entryLo0, entryLo1 and page-mask) are moved to corresponding CP0 registers. Otherwise TLB read raises invalid data exception. TLB write-random gets index of TLB entry from CP0 random register and checks the validity of data at the index. If entry is dirty, it raises dirty data exception, otherwise it writes corresponding values of CP0 registers (i.e. entryHi, entryLo0, entryLo1 and page-mask) to the TLB entry at that index. TLB write-index works same as TLB write-random except that it gets index value from CP0 Index register.

On TLB miss or TLB Mod exception, we jump to the exception handler entry point from where the kernel determines which kind of exception it is and service the exception.

4.1.5 Bugs Fixed in TLB

- **TLB Restructuring:** TLB structure and searching mechanism is modified to accommodate changes related to core mask. At boot time when control passes from u-boot to Linux kernel, core mask is changed which caused TLB exception in previous implementation and kernel crashes but on such thing happens on actual hardware. Some debugging revealed that searching mechanism on hypervisor is slightly different from hardware. In order to search TLB entry, hardware uses pagemask of each entry placed in TLB instead of using pagemask value of CP0 register.
- **Reserved Bits for TLB Registers:** Some bits for EntryHi, EntryLo and Pagemask registers are reserved and shouldn't be changed by the guest operating system. Changing these bits can cause problem in translation of virtual address and sometimes cause kernel crash. Masking of such reserved bits was provided for correct virtual address translation mechanism.

4.1.6 Cavium Segment Implementation

CVMSEG is cavium specific memory segment. CVMSEG resides in KSEG3 region and all memory reference in address range 0xFFFFFFFFFFFF8000 - 0xFFFFFFFFFFFFBFFF are treated specially by MIPS core. Access to this segment is controlled by setting CvmMemCtl[CVMSEGENA*] flags and size of this segment is controlled by CvmMemCtl[LMEMSZ] field. CVMSEG has two portions

1- CVMSEG LM = 0xFFFFFFFFFFFF8000 - 0xFFFFFFFFFFFF9FFF

2- CVMSEG IO = 0xFFFFFFFFFFFFA000 - 0xFFFFFFFFFFFFBFFF

CVMSEG LM is a segment that access portion of DCache as local memory. Larger the size of this segment, smaller the size of DCache. CVMSEG IO has only one legal address 0xFFFFFFFFFFFFA200 and store to this address issues IOBDMA command which returns data from IO bus to CVMSEG.

Operating system normally uses this region as scratch pad memory and register values are stored at these locations during context switching. Implementation of this region was crucial for successful booting.

4.2 MIPS Instruction Set Translation

MIPS instructions are mainly categorized as R, I and J types. “R” category contains those instructions which use on gp registers. I types involves an immediate value plus register and J type has target address field, no registers to manipulate.

The idea is to not completely emulate the instruction but rather change the registers embedded in instruction and execute it on hardware as it is. The registers that are replaced are first loaded with the expected contents. These loading instructions are also written in assembly language. The complete translation of an instruction will have some loading instructions then the actual reconstructed instruction and then some storing instructions. Control shifting and flow control instructions are treated differently.

We have a memory based copy of all registers (i.e. GP, CP and special registers) which belongs to guest OS. After the execution of particular instruction, guest's registers would also be updated accordingly. For translating mips instructions into equivalent set of instructions which will produce same results in the registers kept for guest, we use 3 gp registers. The expected contents (from host's point of view) of the registers are first loaded in these registers and then replaced in instruction to be executed. Results are then saved to our memory based registers.

The categories are based on the type of instruction, privileged or unprivileged, how many registers are used, what is destination register and how the fields are manipulated.

Below are the categories and the instructions included in them are also mentioned.

4.2.1 PRIVILEGED INSTRUCTIONS

Instructions which involve Co-processor 0 registers are privileged instructions and can't be executed in user mode. Privileged Instructions are treated separately.

- **mfc0**: The translated set of instructions will load the contents from particular cp register and store in the place of destination gp register in the memory.
- **mtc0**: The translated set of instructions will load the contents from particular gp register and store in the place of destination cp register in the memory. It also checks whether destination register is $0 or not. If it is then the instruction is replaced with "nop". Certain bits of cp registers are reserved. To avoid over writing them, masking is used.
- **tlbr/ tlbwi/ tlbwr/ tlbp**: In case of tlb instructions, an integer is placed on a particular place in the memory and control is shifted back to handlerRequest(). The control mark indicates the instruction to be handled accordingly.
- **di** (disable interrupts): First the contents of status register is loaded. If any destination register is given then the contents of status register is stored at destination register. First bit of status register is cleared to disable interrupts and the contents are stored in the status register.
- **ei** (enable interrupts): First the contents of status register is loaded. If any destination register is mentioned then the contents of status register is stored at destination register. First bit of status register is set to enable interrupts and the contents are stored in the status register.
- **eret** (exception return): when exception routine end, eret is executed to return to the pc from where we have received exception. In its implementation we first check whether erl bit of status register is set or not. If set then error epc is returned, if not then epc value is returned. The returned value is assigned as next pc to be executed.

4.2.2 UNPRIVILEGED INSTRUCTIONS

Unprivileged Instructions are grouped on the basis of their type and functionality.

- **unprev_R**
  - All those R-type unprivileged instructions, which use 3 gp registers. 2 source gp register and one destination gp register.
  - Includes: baddu, dmul, dpop, pop, or, sllv, dsllv, srlv, dsrlv, rotrv, drotrv, srav, dsrav, movz, movn, add, dadd, addu, daddu, sub, dsub, subu, dsubu, and, xor, nor, slt, sltu, mul, wsbh, seb, seh, dsbh, dshd, clz, clo, dclz, dclo, seq, sne (40 total)
  - First 2 source registers are loaded from memory into register $12 and $13. The register in the instruction to be translated is replaced with these registers and executed as it is. The result is stored on the destination memory based gp register.
- **shift_R**
  - All those R-type unprivileged instructions, which are shift instruction and the no. of times to be shifted is encoded in instruction itself (i.e field from bit 6 to 10). 1 source gp register and 1 destination gp register.
  - Includes: dsrl, srl, dsll, sll, drotr, rotr, dsra, sra, drotr32, dsll32, dsrl32, dsra32 (12 total)
  - First source register is loaded from memory into register 12. The register in the instruction to be translated is replaced with the register and executed as it is. The result is stored in the destination memory based gp register.
- **mulDiv_R**
  - All those R-type unprivileged instructions, which multiple or divide and the destination registers are special register HI and LO (opposite to mul instruction included in uprev_R, whose destination is also a gp register) and 2 source gp registers.
  - Includes: dmult, mult, dmult, multu, ddiv, div, ddivu, divu, madd, maddu, msub, msubu (total 12)
  - First source registers are loaded from memory into registers 12 and 13. The instruction to be translated is replaced with these registers. After the execution of

these instructions the result will be in HI and LO special registers. Mflo and mfhi is executed after these instructions. The result is stored in the guest's Hi and LO.

    - For instruction “madd”, HI and LO are registers of the hardware is also updated first before executing it.

- **moveFromLoHi_R**
    - For moving contents from HI and LO special registers into the gp registers, contents are loaded from HI and LO and saved at the place of destination gp register.
    - Includes : mflo, mfhi (total 2)
- **moveToLoHi_R**
    - For moving contents to HI and LO special registers from gp registers, contents are loaded from particular gp register and saved at the place Hi or Lo register.
    - Includes: mtlo, mthi (total 2)
- **ext_R (extract)**
    - These are R-type instructions, whose fields are used differently than the previous categories. Bit 16-20 are used for destination register and bits 11-15 are used for size. 1 gp source and 1 gp destination register is used.
    - Includes: ext, dextm, dextu, dext, exts, exts32 , (total 6)
    - Source register is first loaded in register 12. Then instruction to be translated in executed with 12 and 13 registers. The result in 13 register is stored in the destination gp register.
- **ins_R (insert)**
    - These are R-type instructions, whose fields are used differently than the previous categories. Bit 16-20 are used for destination register and bits 11-15 are used for size. 1 gp source and 1 gp destination register is used. Similar to extract but the difference is that destination register is also loaded before the execution of instruction.
    - Includes: ins, dinsm, dins, dinsu, cins, cins32 (total 6)
    - Source and destination registers are loaded in register 12 and 13 respectively. Then instruction to be translated in executed with 12 and 13 registers. The result in 13 register is stored in the destination gp register.

- **unprev_I**
    - All those I type instructions which use 1 source and 1 destination register (except lui which have no source register but the translation would not produce any error if translated in this category).
    - Includes: daddi, daddiu, addiu, slti, sltiu, andi, ori, xori, lui, addi, seqi, snei (12 total)
    - Source register is loaded. Instruction to be translated is executed with register 12 and the result is saved in the destination register's place.
- **unprev_I_Load**
    - All I-type load instructions
    - Includes: ldl, ldr, lb, lh, lwl, lw, lbu, lhu, lwr, lwu, ll, lld, ld (total 13)
    - First the address from where the contents would be loaded is translated in terms of hypervisor. For that the address which needs to be translated is saved on a particular location and control is given to the handler. The translated address is loaded in the register and then the load instruction is executed. The loaded contents are saved on the destination register.
- **unprev_I_Store**
    - All I-type store instructions
    - Includes: sdl, sdr, sb, sh, swl, sw, sh, swr, sw, sc, scd, sd (total 12)
    - First the address from where the contents would be stored is translated. For that the address which needs to be translated is saved on a particular location and control is given to the handler. The translated address is loaded in the register and then the store instruction is executed.
- **LL and SC:** Load-Linked and Store Conditional are two instructions which are used to atomically implement read-modify-write using a special LLBit. Assembly instructions are added to translation for correct implementation.

4.2.3 CAVIUM SPECIFIC INSTRUCTIONS

These instructions don't have the standard R, I or J format. Their format is a bit different along with a little difference in their operation from standard instructions.

- **saa**
    - This instruction atomically adds a word to a memory location.
    - This is similar to a store but this instruction directly accesses a memory location contents and adds least significant 32 bits of gp register and save to same memory location. All this operation is done without any interrupt or execution of any other instruction.
    - Other store instructions store the contents of gp register to a particular memory location.
    - The difference in translation is due to the different format of the instruction. Other store instruction has an offset field but this instruction doesn't have any offset field.
- **saad**
    - This instruction is similar to saa but the register's content to be added will be considered 64 bit rather than 32 bit.
- **seqi_snei**
    - This instruction checks whether the value of gp register is equal to the 10 bit constant, specified in the instruction. If equal, then destination register is set otherwise cleared. The translation is provided accordingly.
- **v3mulu**
    - This cavium specific instruction performs 192x64 bit unsigned multiplication. Its execution involves special purpose registers P0, P1, P2, MPL0, MPL1 and MPL2. As hypervisor has its own copy of special purpose registers, so before multiplication we have to move the contents of these registers to hardware and then execute multiplication.
- **mtm0**
    - This instruction is R type (related with v3mulu instruction), which could be categorized in unprev_R. But it moves the contents of gp register to special purpose register (MPL0).
- **mtm1**
    - This instruction is R type (related with v3mulu instruction), which could be

categorized in unprev_R. But it moves the contents of gp register to special purpose register (MPL1).

- **mtm2**
    - This instruction is R type (related with v3mulu instruction), which could be categorized in unprev_R. But it moves the contents of gp register to special purpose register (MPL2).
- **mtp0**
    - This instruction is R type (related with v3mulu instruction), which could be categorized in unprev_R. But it moves the contents of gp register to special purpose register (P0).
- **mtp1**
    - This instruction is R type (related with v3mulu instruction), which could be categorized in unprev_R. But it moves the contents of gp register to special purpose register (P1).
- **mtp2**
    - This instruction is R type (related with v3mulu instruction), which could be categorized in unprev_R. But it moves the contents of gp register to special purpose register (P2).

4.2.4 Branch and Jump Instructions

These instructions include all variants of branches and jumps. One of the reasons to categorize them separately is due to the execution of delay slot. In this case, two instructions are translated collectively.

- **bne_beq (branch if not equal , branch if equal)**
    - These are only two branch instructions which use two source registers.
    - Includes: bne, beq (total 2)
    - First the sources registers are loaded into the temp registers and then the delay slot is executed. Branch's source are first loaded due to the fact that delay slot might change the contents of the registers involved in branch. For correct execution of branch its

source registers are loaded in temporary registers. Then the actual branch is executed but with different offset because the target address needs translation. If the branch is taken than offset is added in branch's pc and if not 1 is added in the branch's pc, then this address is stored on a particular place and the control is shifted to the handler.

- **Branch**
  - Those branch instructions which use one source register.
  - Includes: bltz, blez, bgez, bgtz, bltzal, bgezal, bbit0, bbit032, bbit1, bbit132 (total 10)
  - First the source register is loaded and then the delay slot is executed. Branch's source are first loaded due to the fact that delay slot might change the contents of the register involved in branch. Then the branch is executed but with different offset because the target address needs translation. If the branch is taken than offset is added in branch's pc and if not 1 is added in the branch's pc, then this address is stored on a particular place and the control is shifted to the handler.
- **bne_beq_likely**
  - Both instructions use two registers but different from the previous bne_beq category due to the fact that the execution of delay slot is conditional. If the branch is taken then the delay slot is executed otherwise not.
  - Includes: beql, bnel (total 2)
  - First the sources registers are loaded into the temp registers and the actual branch is executed but with different offset because the target address needs translation. If the branch is taken than delay slot is executed and offset is added in branch's pc and if not 1 is added in the branch's pc, then this address is stored on a particular place and the control is shifted to the handler.
- **branch_likely**
  - These instructions use one register but different from the previous branch category due to the fact that the execution of delay slot is conditional. If the branch is taken then the delay slot is executed otherwise not.
  - Includes: bltzl, blezl, bgezl, bgtzl, bltzall, bgezall (total 6)
  - First the source register is loaded into the temp register and the actual branch is

executed but with different offset because the target address needs translation. If the branch is taken than delay slot is executed and offset is added in branch's pc and if not 1 is added in the branch's pc, then this address is stored on a particular place and the control is shifted to the handler.

- **j (jump)**
    - It doesn't use any source register. It is an “I” type Instruction.
    - Includes: j (total 1)
    - First the delay slot is executed then for executing j the target address needs translation. The address is extracted from instruction encoding and placed at a particular place. Then control is shifted to handler.
- **jr (jump register)**
    - This instruction uses one source register. It is an R type Instruction.
    - Includes: jr (total 1)
    - First the delay slot is executed then for executing jr the target address needs translation. The address is already in the register, it is placed at a particular location in memory. Then control is shifted to handler.
- **jal (jump and link)**
    - It doesn't use any source register. It is an “I” type Instruction and differs from previous “j” due to additional linking operation.
    - Includes: jal (total 1)
    - First the delay slot is executed then for executing jal the target address needs translation. The address is extracted from instruction encoding and placed at a particular place. Then the linking address (i.e. pc+8) is stored in register 31 and control is shifted to handler.
- **jalr (jump and link register)**
    - This instruction uses one source register. It is an R type Instruction and differs from previous “jr” due to additional linking operation.
    - Includes: jalr (total 1)
    - First the delay slot is executed then for executing jalr the target address needs

translation. The address is already in the register, it is placed at a particular location in memory. Then the linking address (i.e. pc+8) is stored in register 31 and control is shifted to handler.

4.2.5 Control Shifting Instructions

These instructions break the normal execution path and shift the control to exception handler. Executing these instructions as it is on hardware will shift the control to host's exception handler and not of the guest's. During translation, this type of instruction is replaced with the instructions, which will shift control to the hypervisor along with a control mark. Hypervisor will perform exception handling accordingly to control mark value.

- **Trap Instructions:** Trap instructions in a system shift the control to exception handler if the condition is true. This instruction can't be executed as it is on the hardware because if true then the control will shift to host's exception handler. So, the condition is checked before and if true then the control is shifted to handler, otherwise next instruction.

- **teq_tne_R**
  - R type trap instructions, which use two source registers.
  - Includes: teq, tne (total 2)
  - First the condition is evaluated, if true the control is shifted back to hypervisor otherwise next instruction is executed.
- **tge_tgeu_tlt_tltu_R**
  - R type trap instructions, which use two source registers but differs in translation.
  - Includes: tge, tgeu, tlt, tltu (total 4)
- **teqi_tnei_I**
  - I type instruction, with one source register and 1 immediate value.
  - Includes: teqi, tnei (total 2)

- **tgei_tgeiu_tlti_tltiu_I**
  - I type instruction, with one source register and 1 immediate value but differ in translation.

   - Includes: tgei, tgeiu, tlti, tltin (total 4)
- **Syscall:** In place of syscall, the control is transferred back to the hypervisor with a specific control mark. Hypervisor service the exception accordingly.
- **Break:** In place of break, the control is transferred back to the hypervisor with a specific control mark. Hypervisor service the exception accordingly.

4.2.6 Special Instructions

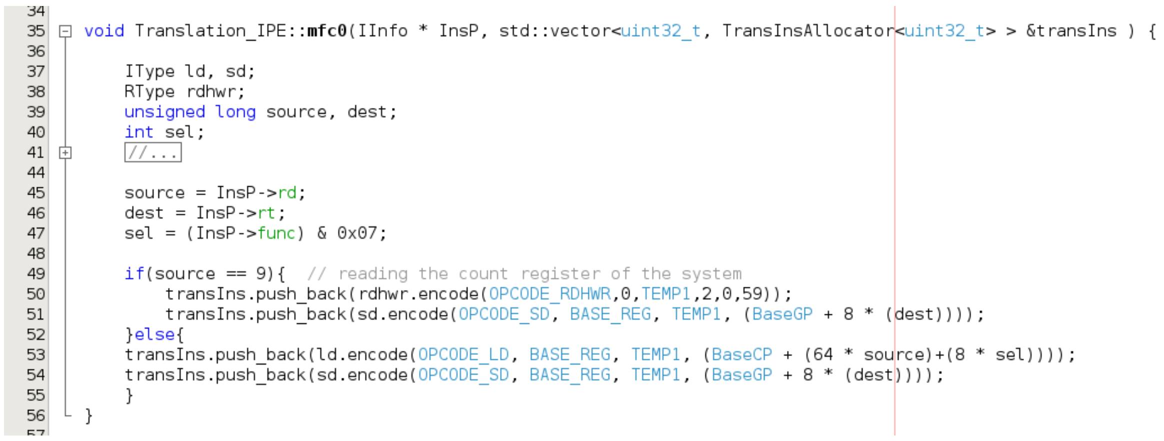
- **rdhwr:** This is a special instruction which allows reading of some hardware registers while in user mode. Due to current translation, only zero is read into the destination register when this instruction is executed. In case of SMP, it is used to get core number.
- **Pref, deret, cache and ssnop:** These instructions are replaced with "nop".
- **Wait:** IP (interrupt pending) bits of "cause" register are monitored continuously. If anyone of them is set, indicating the presence of external interrupt, control is shifted back to hypervisor for interrupt handling.

4.2.7 Examples of Translation

1. mtc0

If we have an instruction: mtc0 v0, c0_status. After executing mtc0(), translated instructions would be:

1. ld t0, offset(a7)
2. sd t0, offset(a7)



```
void Translation_IPE::mfc0(IInfo * InsP, std::vector<uint32_t, TransInsAllocator<uint32_t> > &transIns ) {

    IType ld, sd;
    RType rdhwr;
    unsigned long source, dest;
    int sel;
    //...

    source = InsP->rd;
    dest = InsP->rt;
    sel = (InsP->func) & 0x07;

    if(source == 9){  // reading the count register of the system
        transIns.push_back(rdhwr.encode(OPCODE_RDHWR,0,TEMP1,2,0,59));
        transIns.push_back(sd.encode(OPCODE_SD, BASE_REG, TEMP1, (BaseGP + 8 * (dest))));
    }else{
    transIns.push_back(ld.encode(OPCODE_LD, BASE_REG, TEMP1, (BaseCP + (64 * source)+(8 * sel))));
    transIns.push_back(sd.encode(OPCODE_SD, BASE_REG, TEMP1, (BaseGP + 8 * (dest))));
    }
}
```

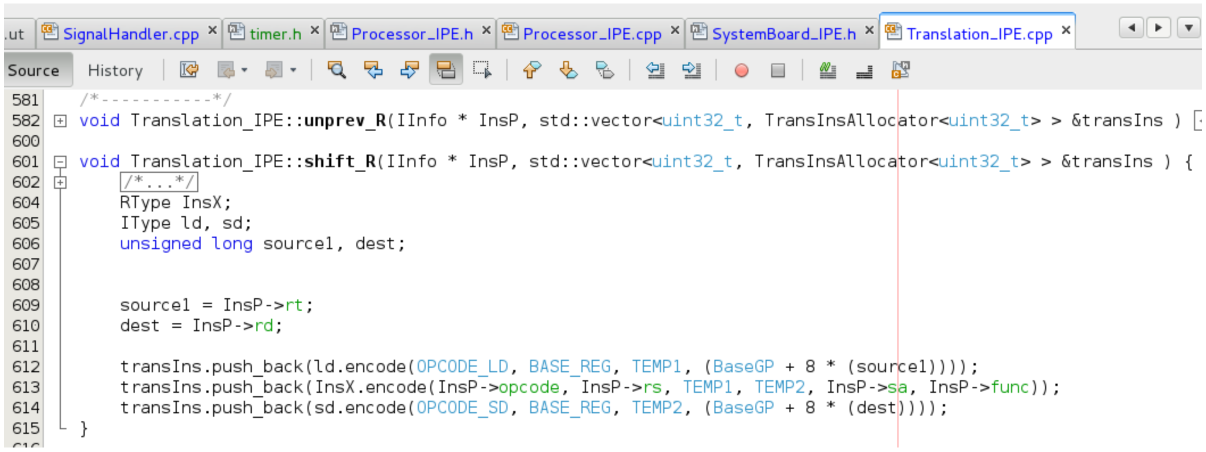
FIGURE 4: Code Snapshot of Mtc0's Translation

The first instruction will load the contents from gp source register and second will store these contents to cp destination register. The offset is created accordingly, as shown in the code. Figure 4 shows the code snapshot of mtc0's translation method.

2. sll

If the instruction is: sll a1,a1,0x2. Translated instruction would be:

1. ld t0,offset(a7)
2. sll t0,t0,0x2
3. sd t0, offset(a7)



```
/*-----------*/
void Translation_IPE::unprev_R(IInfo * InsP, std::vector<uint32_t, TransInsAllocator<uint32_t> > &transIns )

void Translation_IPE::shift_R(IInfo * InsP, std::vector<uint32_t, TransInsAllocator<uint32_t> > &transIns ) {
    /*...*/
    RType InsX;
    IType ld, sd;
    unsigned long source1, dest;


    source1 = InsP->rt;
    dest = InsP->rd;

    transIns.push_back(ld.encode(OPCODE_LD, BASE_REG, TEMP1, (BaseGP + 8 * (source1))));
    transIns.push_back(InsX.encode(InsP->opcode, InsP->rs, TEMP1, TEMP2, InsP->sa, InsP->func));
    transIns.push_back(sd.encode(OPCODE_SD, BASE_REG, TEMP2, (BaseGP + 8 * (dest))));
}
```

FIGURE 5: Code Snapshot Of Sll's Translation

The first instruction brings the contents from guests gp register into temporary register t0, 2nd instruction executes the actual instruction but in terms of temporary register. The third instruction stores the result of target address back to the gp register of guest. Figure 5 shows the code snapshot for sll.

4.3 SOFTWARE CACHE

Guest code passes through a translation layer to make it amenable to run under our hypervisor. Currently this translation is done instruction by instruction and the output is then fused together to make a block. By definition one block ends when control flow has more than one option to move forward (e.g. an unconditional jump, if-else structure etc).

Translation is a fairly involved process and it is desirable to do the translation once and re-use it on subsequent execution. There are many repetitive code structures (e.g. loops) where one block is executed more than once. To seize these performance opportunities, each translated cache is stored in a software cache. Software cache is configurable and initially set to a space for keeping 37 blocks. A class named TranslatedBlockCache is implemented which has rich set of functions to store, retrieve and search a block.

4.3.1 CACHE STORAGE

Hash Map based storage: Due to previous hypervisor architecture, translated blocks are copied at a pre-specified place where epilogue and prologue are already present along with some extra software exception handling code. To copy a block at a new location, software cache generates a new copy and stores it in the cache. Due to optimization needs, we replaced this by new array based storage. (For details see section 7)

Array Based Storage: Currently the translated blocks are placed in array of containers. In this method we don't copy the translated block to any particular place but instead we use the block as it is, when it is created dynamically during translation. We store a pointer to these block in a hash map for reuse.

4.3.2 SEARCHING A BLOCK

Software cache is capable of searching any block in time O(logn) using HashMap that is a C++ Standard Template Library (STL). Hash maps are famous for speedy searching.

Because we have changed our storing mechanism, searching has also changed. Now we find the block in array using a key and return only the pointer to that container. That's why our searching time of block has changed to O(1).

4.3.3 BLOCK RETRIEVAL

Software cache retrieves a block and copies it to a specified location for execution. Retrieval can be based on specific key provided at the time of storage.

Copying the block to new memory location was time consuming operation. So now, we find the presence of block in array and retrieval is done by returning a pointer to that block in array.

4.3.4 Replacement policy

A simple random replacement policy is used to replace a block when the cache is full. A block is randomly selected to replace it with the newly coming block. Replacement policy has also changed. Now, whenever a new block is required, index is generated on base of given key and old block at that index is replaced by new one. Key is converted to index using eq. Index = Key % max capacity.

For Optimization purpose, modulus operation was changed to bit shifting operation.

4.3.5 Bugs fixed in Address cache

- **Address Cache Clearing on ASID Change:** When kernel runs in user mode and starts user mode processes then multiple virtual pages can map to a single physical frame or single virtual page can be mapped to multiple physical frames. For example, for process 1 virtual address 0x120001021 can be mapped to physical address 0x41A36C021. For process 2, same virtual address (i.e. 0x120001021) can also be mapped to 0x4108D9021. To avoid conflict in address translation, address translation cache must be flushed when a process is switched. In case of process switching, kernel writes ID of new process in ASID field of EntryHi register. When ASID is changed we flush address translation cache to provide correct address translation.
- **Read/Write Protection in Address Translation Cache:** Guest kernel can set read/write protection of a page by setting “dirty bit” of entryLo register. When address translation is done using TLB, the state of this bit should be taken under consideration so that data may not be written to a read-only page because this can corrupt the data in RAM. Without this bit implementation memory corruption causes the guest kernel to crash during loading of dynamic libraries. Dirty bit was already implemented in TLB but was not being checked during address translation cache implementation. This implementation was provided in address translation cache for proper working of dynamic user binaries.

4.4 Timer Unit

On actual hardware, Operating system keeps track of time by receiving a timer tick after a configured time. This timer interrupt gives the timing framework to the OS above it. For providing timer tick to guest OS, we have tried two timer infrastructures. Description of both along with their drawbacks and benefits are given.

4.4.1 CONTINUOUS TICK TIMER

To provide the timer interrupt to the system, we have to provide a continuous tick to the guest. Timer unit is initialized in a separate thread as the hypervisor is started. The timer Unit thread registers a timer with the host OS (i.e. 5nsec interval). After every timer expiration time, the thread directly receives the SIGALRM from the host OS. On receiving SIGALRM signal, the cause register (IP7 bit) of every core is set. IP7 bit of cause indicates the presence of timer interrupt. If interrupt mask in status register is set then the interrupt would be taken after setting exception code in cause. The interrupt mask is configured by the guest itself. The cause is checked for timer interrupt every time the control is shifted to handleRequest() of hypervisor. So, timer interrupt can also be taken between the block but not necessarily at the exact time it occurred.

The timer bit (i.e. IP7) of cause is cleared, when there is a write operation on compare register. Note that this interrupt isn't handled by the CIU. CIU can only change IP2, IP3 and IP4 bit of cause register.

Implementation of Count and Cavium Count Register: Only setting IP7 in cause register, at timer expiration wasn't the correct implementation that was needed by the guest OS. Count, Compare and Cavium count registers are hardware register that are used by OS to schedule processes. Although the kernel keep the timing information by incrementing jiffies at every timer interrupt but guest also reads these hardware registers for setting timer with the hardware or getting timing information. These registers are incremented on each clock cycle by hardware and whenever value of Count register become equal to value of compare register an interrupt is generated and notifies guest about timer event which then schedules the processes and again sets time for next timer event.

In implementation, a timer is set with the host for 2ns. At timer expiration, Count and Cavium count registers are incremented and if Count becomes equal to compare register an interrupt is generated by setting the IP7 bit of cause register. Count and Compare are hardware registers and are not readable as GP or CP registers. To read hardware registers, MIPS provide some special instruction (i.e. rdhwr). Same instruction is used to read count register. Implementation of this instruction is given to facilitate guest reading these registers and use them accordingly. Figure 6 shows the flow chart of continuous tick timer implementation in hypervisor.

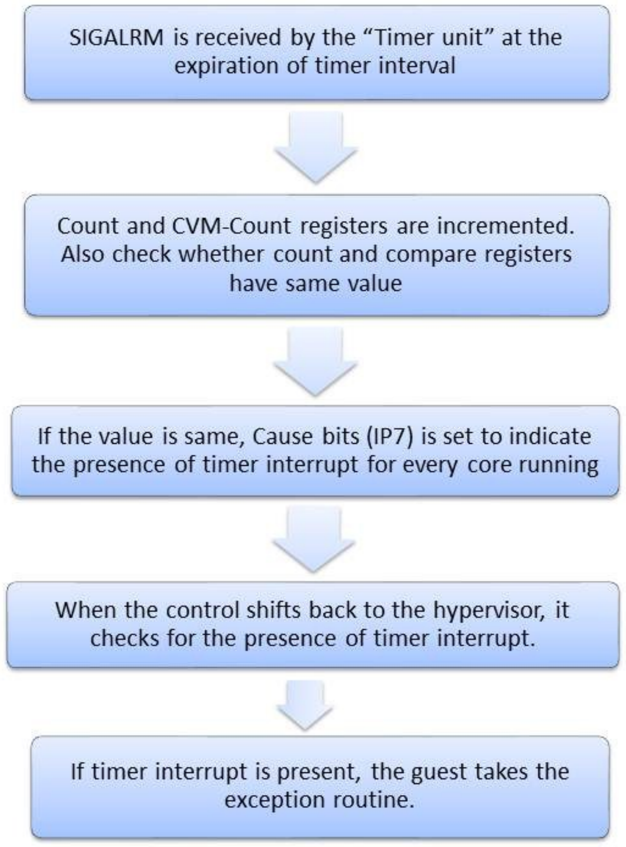

SIGALRM is received by the "Timer unit" at the expiration of timer interval
Count and CVM-Count registers are incremented. Also check whether count and compare registers have same value
If the value is same, Cause bits (IP7) is set to indicate the presence of timer interrupt for every core running
When the control shifts back to the hypervisor, it checks for the presence of timer interrupt.
If timer interrupt is present, the guest takes the exception routine.

FIGURE 6: Continuous Tick Timer Implementation

Drawbacks: Although this design strategy works correctly but it has two major drawbacks. First, it is computationally intense. As a periodic timer is being registered for such a short time period (i.e. 5 ns sec) and on timer expiry we have to increment the count by one. Also check the conditions to whether generate a guest interrupt or not. Second, even incrementing count after such short duration, the increment of count was very slow. At the command prompt, we see delayed response of entering the command and its execution due to or slower timer mechanism.

4.4.2 ON-DEMAND TICK TIMER

This strategy is completely different from the Continuous tick timer. In this strategy rather than creating a separate thread, timer unit is embedded in each core thread. Now the timer works serially with the core. Figure 7 shows the design difference in Continuous tick timer and On-demand timer.

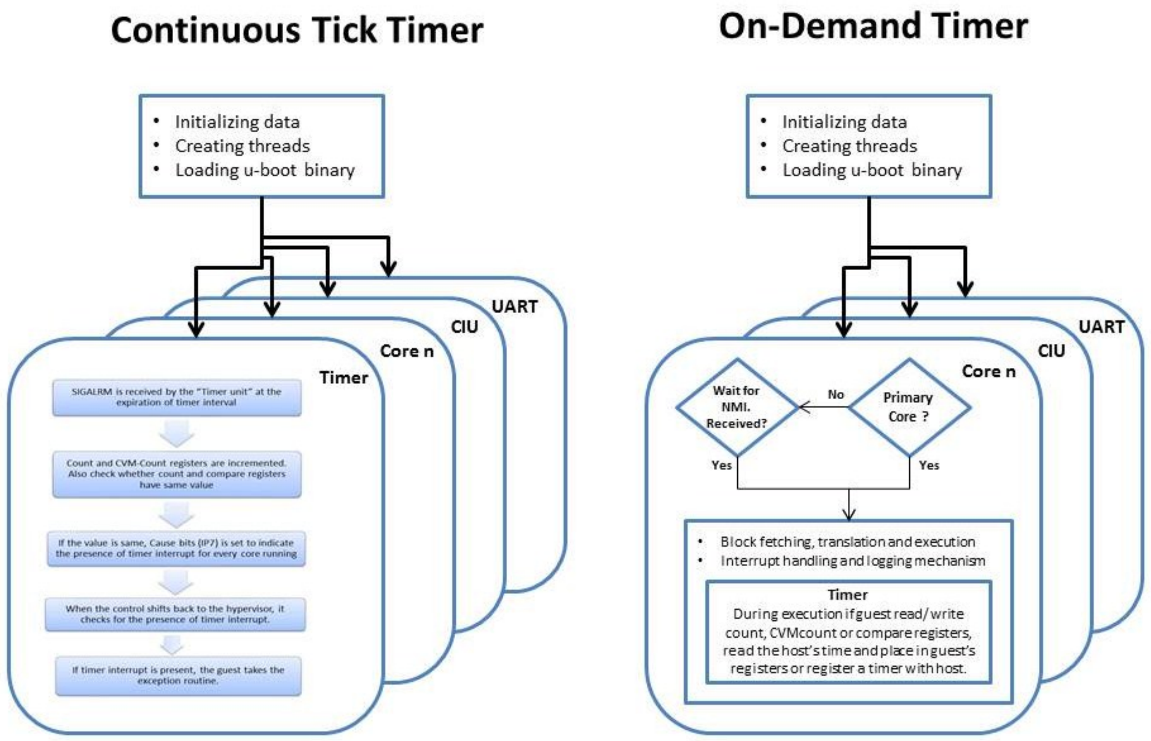

Continuous Tick Timer
Initializing data
Creating threads
Loading u-boot binary
UART
CIU
Core n
Timer
SIGALRM is received by the "Timer unit" at the expiration of timer interval
Count and CVM-Count registers are incremented. Also check whether count and compare registers have same value
If the value is same, Cause bits (IP7) is set to indicate the presence of timer interrupt for every core running
When the control shifts back to the hypervisor, it checks for the presence of timer interrupt.
If timer interrupt is present, the guest takes the exception routine.
On-Demand Timer
Initializing data
Creating threads
Loading u-boot binary
UART
CIU
Core n
Wait for NMI. Received?
No
Primary Core ?
Yes
Yes
Block fetching, translation and execution
Interrupt handling and logging mechanism
Timer
During execution if guest read/write count, CVMcount or compare registers, read the host's time and place in guest's registers or register a timer with host.

FIGURE 7: Design Diagram of Two Timer Strategies

In Continuous tick timer, a complete timer device was created. But On-Demand timer works on different strategy. Previously, the incrementing count register mechanism was way too slow. For fixing this drawback, now the host time is directly given to guest by reading host time and setting the guest's registers. When the guest needs to get time or register a timer, it read/write the count, compare and CVMcount registers.

Whenever the guest needs to get time from hardware it reads the count or CVMcount register. We intercept this read and read the host's time in nanosecond resolution and update guest's count and CVMcount registers. When guest wants to register timer with the hardware, it writes on the compare register. The write operation is also intercepted by hypervisor and it registers a timer with the host. The duration of registered timer is kernel's desired value multiplied by a multiplying factor. This multiplying factor was needed to reduce the increased timer interrupts. Otherwise kernel get stuck in servicing the timer interrupts and actual code is not given time to be executed. After the implementation of this strategy, the prompt is showing less latency when the command is entered.

Advantages: This strategy is not computationally intensive as we only update the count register when the guest reads it (i.e. on-demand from guest). Now we only register the timer when guest want a timer registered with the hardware.

4.5 INTERRUPT AND EXCEPTION HANDLING

Exceptions cause change in normal execution flow and control is transferred to some exception handling routines, if implemented, or crash the application otherwise. During block execution by hypervisor, two possible exceptions could occur:

- An instruction like trap or syscall, itself shifts control to an exception routine. Exceptions like these are called programmed exceptions.
- An exception like overflow, address error and tlb related exceptions are generated during the execution of instruction. This type of exceptions is unpredictable because they are not programmed.

The challenge is to emulate exception handling mechanism in user mode. On an exception, control may go to host kernel and may not return back if not emulated properly. In case of programmed exceptions, the possible emulation is to replace exception-causing instruction with innocuous instructions that explicitly transfer control back to a hypervisor provided handler. The handler could identify actual (exception-causing) instruction from control mask and handle it accordingly. In second case, a signal is raised that could be caught to handle the exception. Once the control is available in hypervisor, exception handling routine could be called to do the rest.

In our implementations, Perform_Exception() is called to set various exception related registers. Exception code is set in cause register. EI, EXL and/or ERL bits of status register are set to indicate the presence of an exception. EPC register is set with the program counter (pc) of exception-causing instruction. According to the exception type, exception entry point is assigned to current pc so that new block could be fetched from there.

When the exception routine is completely executed, eret instruction is called. eret is privileged instruction and cannot be executed on hardware as it is (from user mode). To emulate it, we check the status register and then accordingly set pc back to the address from where exception has actually occurred. Figure 8 shows the overall flow and Figure 9 shows a snippet of hypervisor code, dealing with exception handling.

Entry point for all exceptions is generic except for tlb. For example, invalid tlb entry encountered while executing load/store instruction lead to tlb refill exception. The entry point for tlb refill exception is different from that of others. In case of nested exception (e.g. exception raised in an exception routine), general exception entry point is used and corresponding instruction pc is placed in EPC register. Interrupts are caused by external devices in order to rather communicate or in response to a request. Timer unit creates continuous interrupts in a running system for providing timing information. UART also communicate with the cores through generating interrupt.

4.5.1 SIGFPE: FLOATING POINT EXCEPTION HANDLING

This exception is thrown if the result of an operation is invalid or cause divide-by-zero, underflow or overflow. On production of such results during guest code execution, underlying hardware generates SIGFPE signal. Our hypervisor provide a handler to catch this signal. When control comes to this handler, we redirect it to the exception routine of guest operating system. After executing exception routine, control comes back to the handler form where it is jumped back to the immediate next instruction of exception-causing instruction.

4.5.2 SYSCALL: SYSTEM CALL HANDLING

The system call is the fundamental interface between user mode programs and Linux kernel. syscall() is a small library function that invokes the system call whose assembly language interface has specified number and type of arguments. Whenever the syscall instruction comes in guest code, control is transferred to hypervisor code and then redirected to corresponding exception handling routine of guest operating system. The remaining mechanism remains same as above.

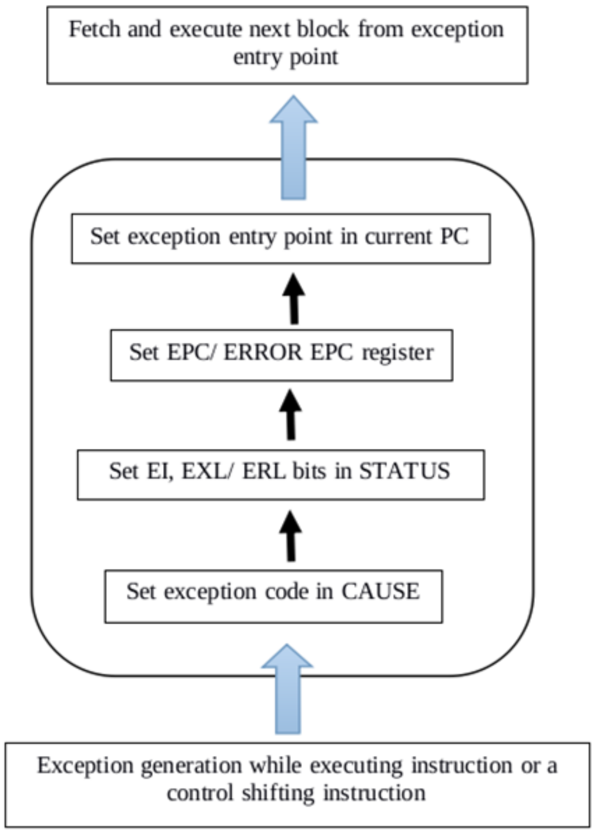
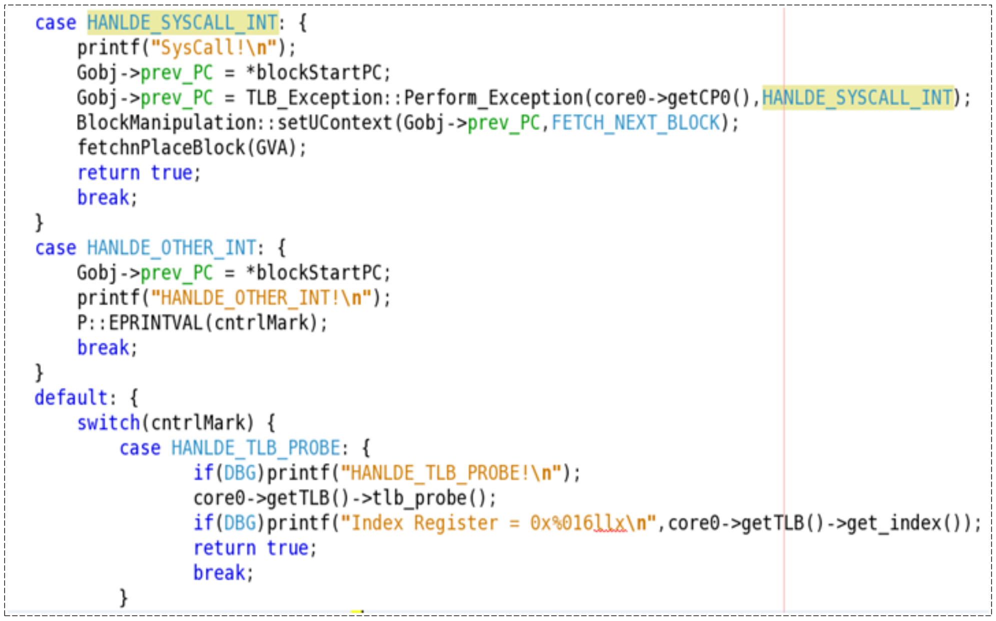

Fetch and execute next block from exception entry point
Set exception entry point in current PC
Set EPC/ ERROR EPC register
Set EI, EXL/ ERL bits in STATUS
Set exception code in CAUSE
Exception generation while executing instruction or a control shifting instruction

FIGURE 8: Exception handling in user mode



```
case HANLDE_SYSCALL_INT: {
    printf("SysCall!\n");
    Gobj->prev_PC = *blockStartPC;
    Gobj->prev_PC = TLB_Exception::Perform_Exception(core0->getCP0(),HANLDE_SYSCALL_INT);
    BlockManipulation::setUContext(Gobj->prev_PC,FETCH_NEXT_BLOCK);
    fetchnPlaceBlock(GVA);
    return true;
    break;
}
case HANLDE_OTHER_INT: {
    Gobj->prev_PC = *blockStartPC;
    printf("HANLDE_OTHER_INT!\n");
    P::EPRINTVAL(cntrlMark);
    break;
}
default: {
    switch(cntrlMark) {
        case HANLDE_TLB_PROBE: {
                if(DBG)printf("HANLDE_TLB_PROBE!\n");
                core0->getTLB()->tlb_probe();
                if(DBG)printf("Index Register = 0x%016llx\n",core0->getTLB()->get_index());
                return true;
                break;
        }
```

FIGURE 9: Code snippet showing the emulation of exception handling

4.5.3 TLB AND ADDRESS ERROR EXCEPTION HANDLING

When the load/store instruction has to be performed in hypervisor first the address on which the load or store has to be performed, is translated into hypervisor address. During this translation, privileges are checked, whether this address is allowed to be accessed or not. If not then address error exception is generated and the next block fetched would be from the exception entry point. But if we have an address which is not violating any privileges, then the contents are looked up in TLB. If the invalid bit or dirty bit is set or no entry is present in the TLB then corresponding exception Mod, TLBL or TLBS is generated.

4.5.4 MODIFICATION IN EXCEPTION HANDLING

Context register was not set before in case of exception because this register is normally used in 32-bit mode but guest was using this register in user mode. Now context register is also been set so guest can read it and perform exception handling correctly.

4.5.5 EXTERNAL INTERRUPTS

Interrupts are caused by the external devices like timer and UART. When an interrupt occurs it set the "pendingInterrupt" variable, which indicates that external interrupt is present. Before fetching the next block, it is checked whether there is any pending interrupts or not. If they are present then some particular bits of status are checked to determine this interrupt should be passed or not. The exception code set for the interrupt is zero and routed to general exception entry point. It is the responsibility of the kernel handler to figure out what kind of interrupt has occurred and dispatch it to proper handler.

4.5.1 CTRL+C SIGNAL FOR GUEST

CTRL + C signal is used to terminate a process in OS. When we press CTRL+C the host OS terminates hypervisor instead of terminating process in guest. For implementing process termination in guest, hypervisor captures this termination signal and sends CTRL+C ASCII character to guest through UART. When guest kernel receives CTRL+C through UART, it terminates a guest process. CTRL+A should be pressed for terminating hypervisor.

4.6 SMP SUPPORT

As mentioned in our high level design, every core will be running in a separate thread that will make our hypervisor a multithreaded process. For providing SMP support, some code level structural changes were needed (e.g. removing all global variables and creating separate objects for each core). Figure 10 shows the multithreaded view of hypervisor, with cores and CIU as separate threads. First hypervisor initialize the necessary data structures and objects. Then it loads uboot binary and dork child threads according to the number of cores initialized and other parallel units. Initially only Core 0 is running and other cores are is sleep mode. After some booting process core 0 enables all other cores. This enabling and controlling mechanism is carried out through CIU (Central Interrupt Unit). The other mechanisms like fetch, translate and execution of blocks remains the same for all cores (section 4.2). Figure 11 shows the modified flow chart of hypervisor.

4.6.1 INTER-CORE COMMUNICATION THROUGH CIU

For Cavium mips64, inter-core communication is performed through CIU. Specific CIU registers (like CIU_Fuse, CIU_NMI , CIU_PP_RST and mailbox registers) are used during interrupt dispatching and identification. CIU_Fuse register contains the information about the number of processors in the hardware. Operating system can have have this information by reading CIU_Fuse register. After some initial booting process core 0 signals other cores to initialize themselves. To do so, primary core (i.e. core 0) sets a bit corresponding to the particular core in CIU_PP_RST register and that core initializes itself on low power mode.

CIU unit and other cores are all in separate threads. The threads running cores (else than primary core) will initially be in sleep mode. Core 0 sends NMI pulse to each core by setting corresponding bit in CIU_NMI register, secondary cores goes out of low power modes and start initializing core.

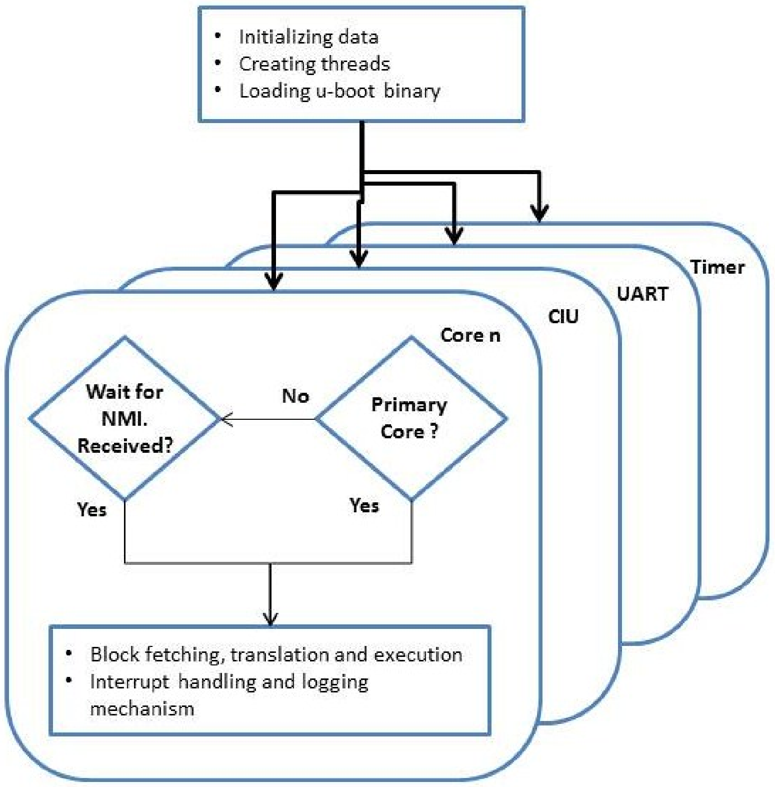


FIGURE 10: Multithreaded view of hypervisor and external devices

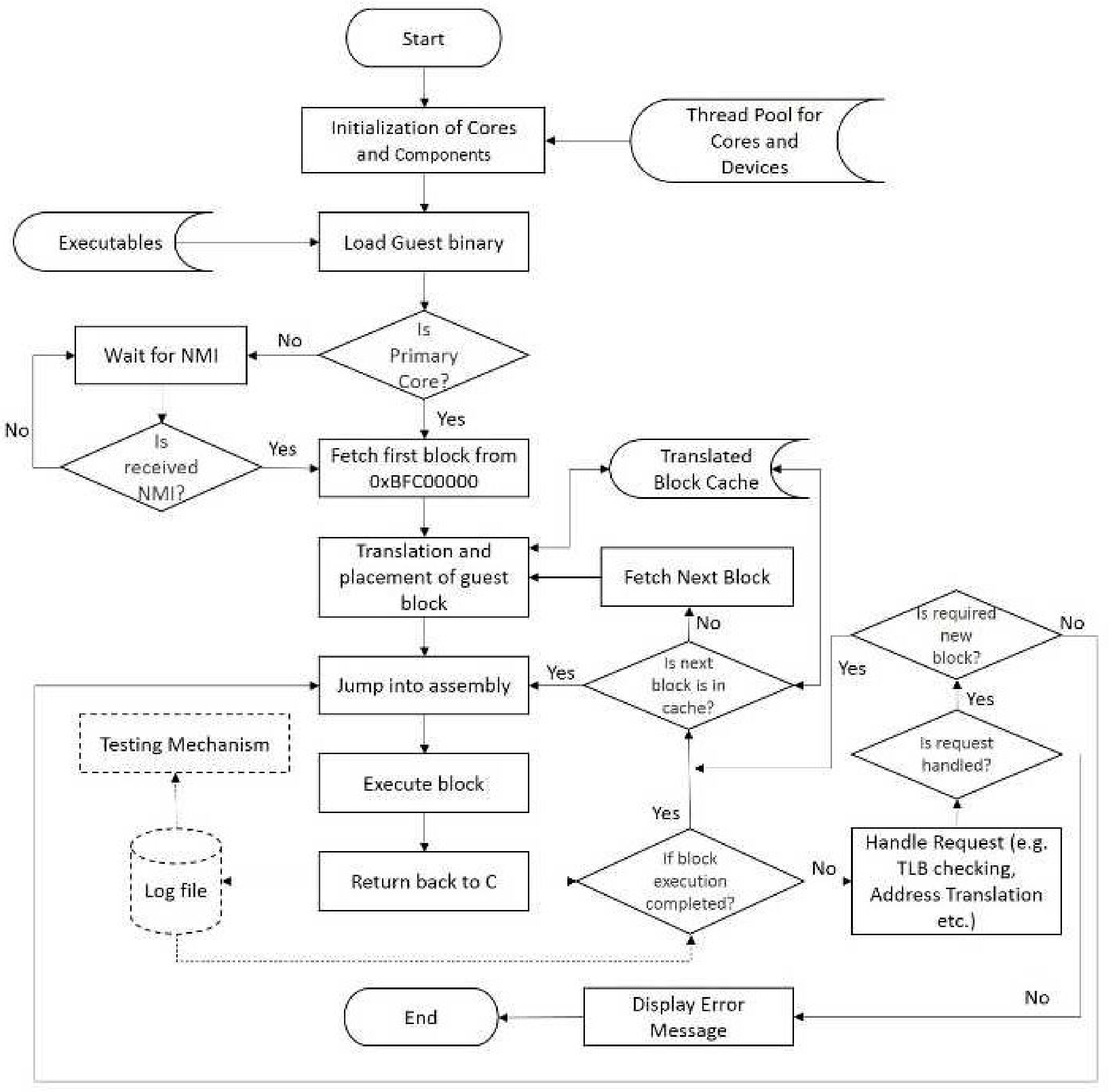


FIGURE 11: Execution flow of hypervisor with SMP

4.7 IO Device Management

In hypervisor, each core and IO device is emulated in separate thread. When a core has to communicate with any device it either reads or writes IO device register. Corresponding IO device is notified and device updates its flags according to the operation. Implementing each device in a separate thread enables maximum parallelization.

To notify IO device thread, a separate class is defined named DeviceMessageBox. It contains address which is being accessed, data which is being written to the register at specified address and whether it is read/write operation. Some posix variables are also part of DeviceMessageBox which are required for thread communication.

At time being, only 2 IO devices are implemented

1. UART (Universal Asynchronous Receiver Transmitter)
2. CIU (Central Interrupt Unit)

4.7.1 UART

The UART is typically used for serial communication with a peripheral, modem (data carrier equipment, DCE), or data set. Either a core or a remote host can use the UART. The cores transfer bytes to and receive characters from the UART core via 64-bit CSR accesses. The UART core transfers and receives the characters serially. Either polling (during booting/ in kernel mode) or interrupts (after booting/ in user space) can be used to transfer the bytes. Processor communicates with console and keyboard using UART device. So, its implementation was inevitable for a complete booting system.

There are basically 12 register in UART.

- **RBR (Receiver Buffer Register):** Receiver buffer register contains data received from input device. Whenever data is received, “Data Available” flag is set in LSR and an interrupt is generated by the UART so that processor can get received data.
- **THR (Transmitter Holding Register):** Transmitter holding register contains data which is being transmitted to the output device. Whenever this register is written, “THR empty” flag is cleared in LSR and UART starts transmitting data. After data is

being transferred successfully, “THR empty” flag is set in LSR and interrupt is generated to tell the processor that UART is idle now and ready to send new data.

- **IER (Interrupt Enable Register):** Interrupts are not generated unless UART is told to do so. Processor enables UART interrupts by setting corresponding flags in Interrupt enable register.
- **IIR (Interrupt Identification Register):** Whenever an interrupt is occurred, processor jumps to its interrupt routine. The interrupt routine must know which kind of event caused that interrupt so that it can service it properly. IIR register tells the processor about the cause of interrupt.
- **FCR (FIFO Control Register):** FIFO is used in UART for both receiver buffer and transmitter buffer. Whenever data is received, it is placed in RBR. But if RBR is not empty then data is moved to receiver FIFO. When UART is transmitting data and new data is provided by processor then it is placed in transmitter FIFO. FCR is used to control the behavior of the FIFOs.
- **LCR (Line Control Register):** LCR is set at initialization time and controls the parameters of line. Parity and number of data bits can be changed using LCR. DLD and DLH can also be accessed by setting “DLAB” flag in this register.
- **MCR (Modem Control Register):** MCR register is used to perform handshaking actions with attached devices. Setting and resetting of control registers is done using this register.
- **LSR (Line Status Register):** LSR shows the current state of communication. Errors are reflected in this register. The state of receiver and transmitter buffers is also available.
- **MSR (Modem Status Register):** MSR contains information about the four incoming modem control lines on the device. The information is split in two nibbles. The four most significant bits contain information about the current state of the inputs where the least significant bits are used to indicate state changes. The four LSB's are reset, each time the register is read.
- **SCR (Scratch Register):** There is no use of this register in UART communication. Sometimes it may be used by processor to store a single byte.

- **DLL (Divisor Latch LSB) and DLM (Divisor Latch MSB):** For generating its timing information, UART uses an oscillator. Oscillator frequency is divided by 16 and obtained value is further divided by value placed in Divisor Latch registers. In this way, baud-rate of UART is adjusted. For error free communication, both receiver and transmitter UART have same timing base i.e. have same baud-rate.

Implementation of UART: There are basically two functions of UART. Transmit data provided by the processor and receive data from input devices. For both these functionalities we have separate threads called Receiver Thread and Transmitter thread.

- **Receiver Thread:** The purpose of receiver thread is to handover data to processor which we input using keyboard. This thread continuously checks for availability of input from keyboard. Whenever input is available, it sets "Data Available" flag in LSR and generates an interrupt. When processor reads received data, "Data Available" flag is cleared from LSR.
- **Transmitter Thread:** The purpose of transmitter thread is to transmit data which is being provided by the processor. Transmitter thread helps the processor in printing all the messages on the console. After transmitting data, it sets "THR is empty" flag in status register and generates an interrupt to tell processor that UART is free now for further transmission.
- **Interrupt Generation:** When UART performs an operation, it checks it IER. If interrupt for corresponding action is enabled, it sets appropriate flags in IIR and notifies the CIU thread about interrupt generation. CIU reads enable registers of all the cores to check if any core wants to receive UART interrupt. If it finds the core with enabled UART interrupt, it sets summary register for that core and generates interrupt. Core jump to its interrupt routine and service the interrupt.

Modification in UART implementation:

In previous version, some characters of guests console output were sometimes missed due to two problems. First, "printf()" was used to print characters which is in fact formatted output and takes much time so we implemented this mechanism with "write" system call which is unformatted and is faster than “printf()”. Second, wait mechanism was introduced in core implementation. Core waits for the UART to print previous characters before sending new ones.

4.7.2 CENTRAL INTERRUPT UNIT (CIU)

CIU is responsible for dispatching interrupt requests (coming) from external devices to a particular core. CIU is discussed here in context of our test bed i.e. Cavium Networks OCTEON Plus CN57XX evaluation board [1]. CIU distributes a total of 37 interrupts i.e. 3 per core plus 1 for PCIe. Three interrupts for each core set/unset bit 10, 11, 12 of Cause register of the core. Using these cause register bits, interrupt handler of a core could prioritize different interrupts. Interrupt requests from external devices are accumulated in a 72-bit summary vectors with naming convention CIU_INT<core#>_SUM<0|1|4>. Summarized interrupts reach to their ultimate destination by using corresponding 72 bits interrupt enable vector with naming convention CIU_INT<core#>_EN<0|1> and CIU_INT<core#>_EN4_<0|1>.

Interaction of CIU, external devices and cores is shown in figure 12 (a). CIU reads memory mapped registers of the external devices to know about pending interrupt requests and sets corresponding bits of cause register of target core. Figure 12 (b) shows a simplest description of the internal working of CIU, where interrupt identification/handling is done in software.

We have implemented a simplest abstraction of CIU. It has been integrated in a copy of main hypervisor code and works as a separate thread. CIU is only reading CP0's cause register. As UART is not fully developed yet, UART's memory mapped registers are artificial (for the time being). UART writing and other devices would be implemented in future. CIU itself has set of summary and enable registers for every core. An interrupt request goes to only those cores that had enabled the interrupt by configuring its enable register. In current code, CIU reads UART's Interrupt Identification Register (IIR), extracts identity bits and set/clear the corresponding summary registers bits. These summary registers for every core are than “AND” with their enable registers to set or clear cause register's bit 10, 11 and 12.

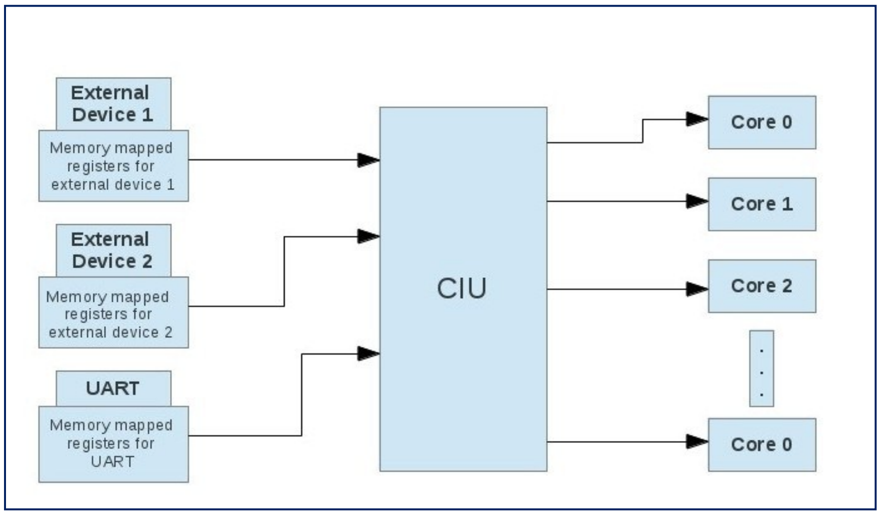

External Device 1
Memory mapped registers for external device 1
External Device 2
Memory mapped registers for external device 2
UART
Memory mapped registers for UART
CIU
Core 0
Core 1
Core 2
.
.
.
Core 0

FIGURE 12(A): (CIU) Interrupt distribution from external devices to core

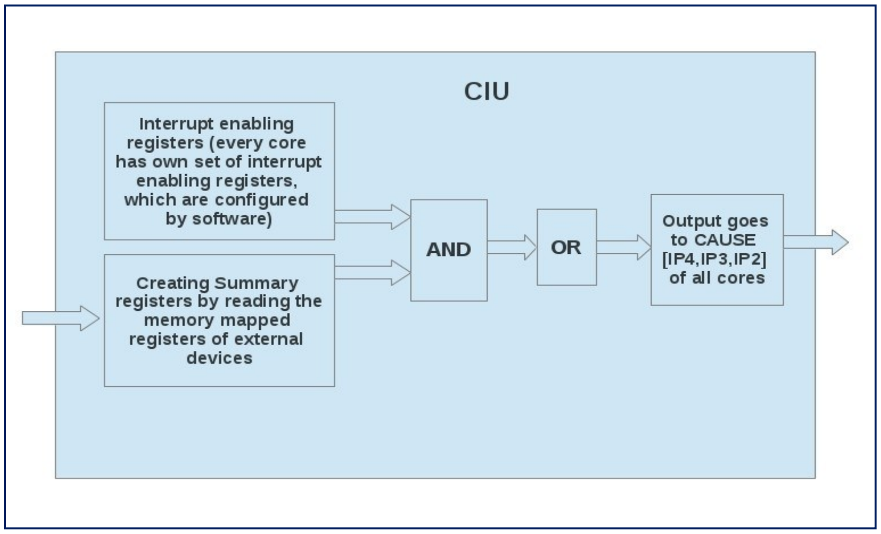

CIU
Interrupt enabling registers (every core has own set of interrupt enabling registers, which are configured by software)
Creating Summary registers by reading the memory mapped registers of external devices
AND
OR
Output goes to CAUSE [IP4,IP3,IP2] of all cores

FIGURE 12(B): Internal working of CIU, inwards arrow comes from external devices and outward arrow goes to all cores

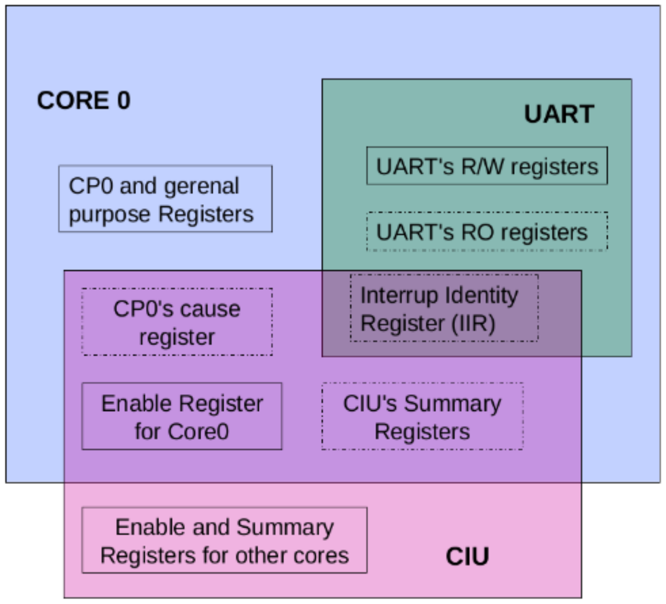

CORE 0
UART
UART's R/W registers
UART's RO registers
CP0 and gerenal purpose Registers
CP0's cause register
Interrup Identity Register (IIR)
Enable Register for Core0
CIU's Summary Registers
Enable and Summary Registers for other cores
CIU

FIGURE 13: Memory mapping between core and external devices

In integrated code, shared memory regions are defined for CIU to work with other components of virtual board (see figure 1). Figure 13 shows these shared memory regions for core0, CIU and a single device i.e. UART. Region overlapping and dotted lines represent the accessibility and access mode of registers, respectively. For example, CP0 Cause register belongs to core0, CIU can access it but UART cannot. As Cause register belongs to core0, it can be read-written by core0 but it is read-only for CIU. IIR register of UART is read-only for CIU and Core0, hence it is at the intersection of three regions and have dotted boundary. CIU's summary registers are read-only for core0, hence dotted and at the intersection of two regions. As CIU's enable register is readable and writeable for core0 and CIU, it has solid boundary and lies in overlapped region.

5 Virtual Disk

A virtual disk (also known as a virtual drive or a RAM drive) is a file that represents as a physical disk drive to a guest operating system. The main idea of providing disk to guest was to create persistence of data across boot. The guest should be able to create and store files on the drive.

Virtio and Vhost can be configured for block devices such as Disk. Virtio para-virtualized driver for emulation of disk was used. Virtio driver directly interacts with Vhost client in host kernel and hypervisor only works on control path i.e. notifying host kernel when data is provided by guest or sending interrupt to guest when vhost completes its assigned task. Due to some limitations, we cannot use Virtio_Blk or Vhost_Blk directly for this purpose and some changes have to be made.

5.1 VIRTIO BLOCK CONFIGURATION

In guest kernel, virtio block is already present and can be enabled from “menuconfig” of kernel. But virtio devices are implemented as PCI devices in kernel. As PCI bus hasn’t been implemented in hypervisor so some changes are required in these drivers to configure them as MMIO based devices. Figure 14 shows the code that needed to be added in virtio_blk.c file.

```
static struct platform_device *vblk_virtio_device;
static void register_mmio_device(void)
{
       int ret;
       struct resource vblk_resources[] = {
              {
              .flags = IORESOURCE_MEM,
              }, {
              .flags = IORESOURCE_IRQ,
              }
       };

       vblk_virtio_device = platform_device_alloc("virtio-mmio", 0);

       if(!vblk_virtio_device)
              printk("***%s device struct initialization failed\n",__func__);

       vblk_resources[0].start = 0x1180070000200ull;
       vblk_resources[0].end = vblk_resources[0].start + 0x120;
       vblk_resources[1].start = OCTEON_IRQ_GPIO0;
       vblk_resources[1].end = OCTEON_IRQ_GPIO0;

       ret = platform_device_add_resources(vblk_virtio_device, vblk_resources,
                                           ARRAY_SIZE(vblk_resources));

       if (ret)
              printk("***%s: device resource allocation failed\n",__func__);

       ret = platform_device_add(vblk_virtio_device);

       if (ret)
              printk("***%s: device add failed\n",__func__);
}
```

FIGURE 14: Code Snippet from virtio_blk.c file

Call this function in “init” of driver to register this device as MMIO device. “vblk_resources” array represents resources of this device. Entry at index zero represents start and end address of this MMIO device and entry at index 1 represents interrupt line for this device. In our case, GPIO0 interrupt is used for this purpose.

We also need to create file for this device in “/dev” folder. There are two options for it. First is to create file by ourselves and second by adding the entry in Makefile for embedded rootfs. To create device file, we use command “mknod /dev/vda b 253 0” where ‘b’ represents block device, ‘253’ represents major of device, and ‘0’ represents minor of device. To automate the process, add the code in “linux/embedded_rootfs/pkg_makefiles/device_file.mk” as shown in Figure 15.

```
sudo mknod ${ROOT}/dev/vda b 253 0
sudo mknod ${ROOT}/dev/vda1 b 253 1
sudo mknod ${ROOT}/dev/vda2 b 253 2
sudo mknod ${ROOT}/dev/vda3 b 253 3
sudo mknod ${ROOT}/dev/vda4 b 253 4
```

Figure 15: Commands for Device Creation

For mounting disk partition in guest, simply make a directory using “mkdir” and mount it using mount command. “/dev/vdaX” are partition file just like “/dev/sdaX” in normal systems.

5.2 VHOST BLOCK CONFIGURATION

We are using Vhost block in host kernel as client interface for Virtio_Blk driver. Vhost block is not part of linux kernel and only some test codes are available on internet. Some of the builds are using some structures and method which are in-compatible for our kernel as well as our Vhost implementation. Provided Vhost_blk client is treating each sector as of 256 bytes but our Virtio driver uses 512 bytes sectors so make this correction in "do_handle_io” function. Once we build the module and load it in our host kernel, we need to create file for this module so that we can use it in our hypervisor. Following command is used to create vhost-blk file, where ‘10’ is major for “misc devices” and ‘243’ is minor for “vhost block”.

```
mknod /dev/vhost-blk c 10 234
```

Now all the major configurations for Virtio Block and Vhost Block are complete in both guest and host end.

5.3 CREATION OF VIRTUAL DISK FOR HYPERVISOR

A simple raw file is used as virtual disk. Some methods are used to create partitions in raw file and to mount these partitions in host. Data can be transferred from host to guest by mounting these partitions in host. Following steps must be taken to create and mount disk file.

- **Create Disk File:** To create disk file, execute the following command. This will create null file of 512 * 262144 = 128MB in current directory.

  ```
  "dd if=/dev/zero of=./disk.img bs=512 count=262144"
  ```

- **Attach Loopback Device to File:** To attach file to loopback device, execute the following command. It will attach file to loopback device. You can confirm it using losetup /dev/loop0 command.

  ```
  "losetup /dev/loop0 ./disk.img"
  ```

- **Create Partitions:** Now simply create partitions using fdisk command.

  ```
  "fdisk /dev/loop0"
  ```

  It is just a raw file and not a device that's why you need to provide cylinder count manually. In fdisk, go to extended menu by typing 'x' and then set cylinder count by typing 'c'. Each cylinder represents 16065 sector or 8 MB. So set cylinders according to size of disk. Verify partitions by typing 'v' before writing back partition table. Now detach file from loopback device by using command.

  ```
  "losetup –d /dev/loop0"
  ```

- **Attach Loopback Device to Partitions:** To attach loopback device to a specific partition in file, offset of that partition must be known. This can be done using fdisk –lu ./disk.img. It output is shown below.

  ```
  Device Boot         Start          End          Blocks        Id      System

  ./disk.img1         63             257039       128488+       83      Linux
  ```

"Start" shows sector offset of partition. Byte offset can be calculated by multiplying it with 512. In this case byte offset of partition is 63 * 512 = 32256. To attach this partition to loopback device, execute the following command.

```
"losetup -o 32256 /dev/loop0 ./disk.img"
```

- **Create Filesystem:** After attaching partition to loopback device, create filesystem using "mkfs" command like mkfs.ext2 /dev/loop0
- **Mount Partitions:** To mount partition, first create a directory for mounting if it is not present already. Now use mount command to mount this partition.

```
"mount -t ext2 /dev/loop0 /mnt/vfs"
```

After mounting, the disk is ready to be used. Data can be copied to/from disk. Un-mount partition after using it and detach file from loopback device.

```
"umount /mnt/vfs"
```

```
"losetup -d /dev/loop0"
```

If you already have file system and you only want to mount it then follow only attach and mount the already created disk.

6 NETWORK INFRASTRUCTURE

For enabling networking in type-II hypervisor, implementation of a network device was inevitable. The foremost method to provide networking is emulation, which will end up us emulating not only network device but also PCI bus. Emulating the behavior of complete device requires more time, effort and also more likely to be slower and inefficient. To implement an efficient solution, type-II hypervisor makes use of a para-virtualized approach. A para-virtualized device driver in guest (i.e. Virtio) as front-end device driver and vhost-net in host as backend driver are used. Virtio creates the networking device inside the guest. Vhost-net is a kernel module in host which connects directly with the networking interface of host system. Both of them are linked through hypervisor's network device implementation. Hypervisor implements the network device, which acts as a bridge between guest's para-virtual device driver and vhost-net on host OS. Figure 16 shows the complete networking infrastructure of type-II hypervisor.

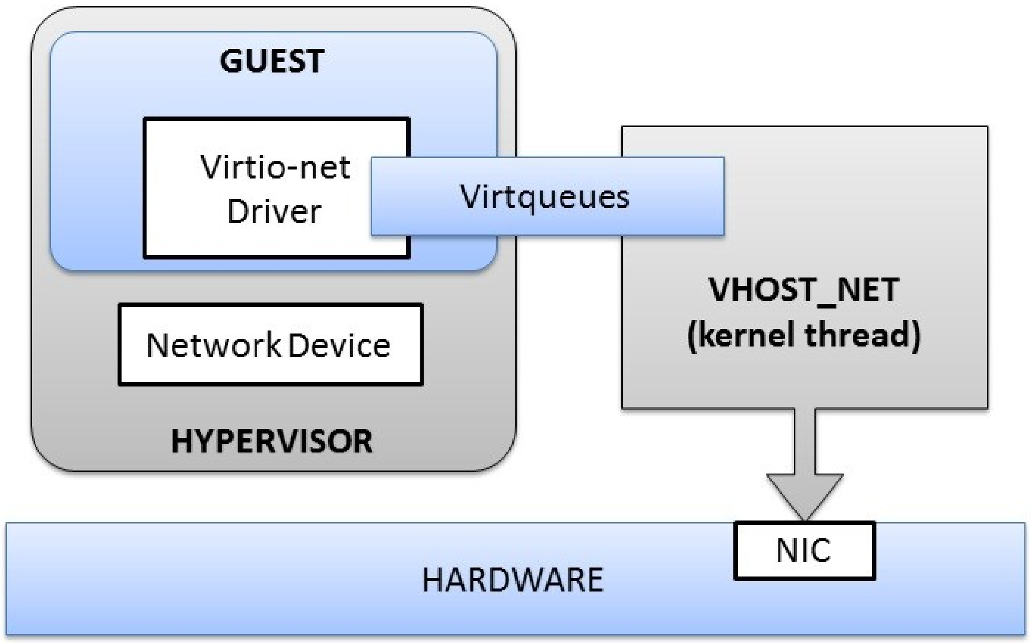


FIGURE 16: Networking Infrastructure of Type-II Hypervisor

The implementation comprises of 3 units; Virtio-net, vhost-net and network device implemented in hypervisor. Providing complete network implementation requires

a) The guest is able to detect and communicate with a networking device.
b) A networking interface with the host system which will be communicating with a real interface on host
c) A link between the two, which routes the network traffic from the guest device to host's interface.

To successfully implement above requirements, three units were designed which are described below.

6.1 VIRTIO-NET

Virtio is a series of efficient, well-maintained Linux drivers which can be adapted for different hypervisor implementations. Virtio is Linux internal abstraction API. It is a standardize virtualization solution for network and disk device drivers. The guest's device driver is only aware that it is running in a virtual environment, and communicates with the hypervisor. This enables guests to get high performance network and disk operations, and gives most of the performance benefits of para-virtualization. Virtio can be used to implement number of types of devices. It provides virtio-block, virtio-net, virtio-pci, virtio-console, etc.

Virtio devices communicate through virtual device using Virtqueues. Each device may have zero or more queues. In case of network device, two queues are used these are called transfer queue and receiver queue. Each queue has size parameter which implies number of entries and size of queue. Each queue has three parts

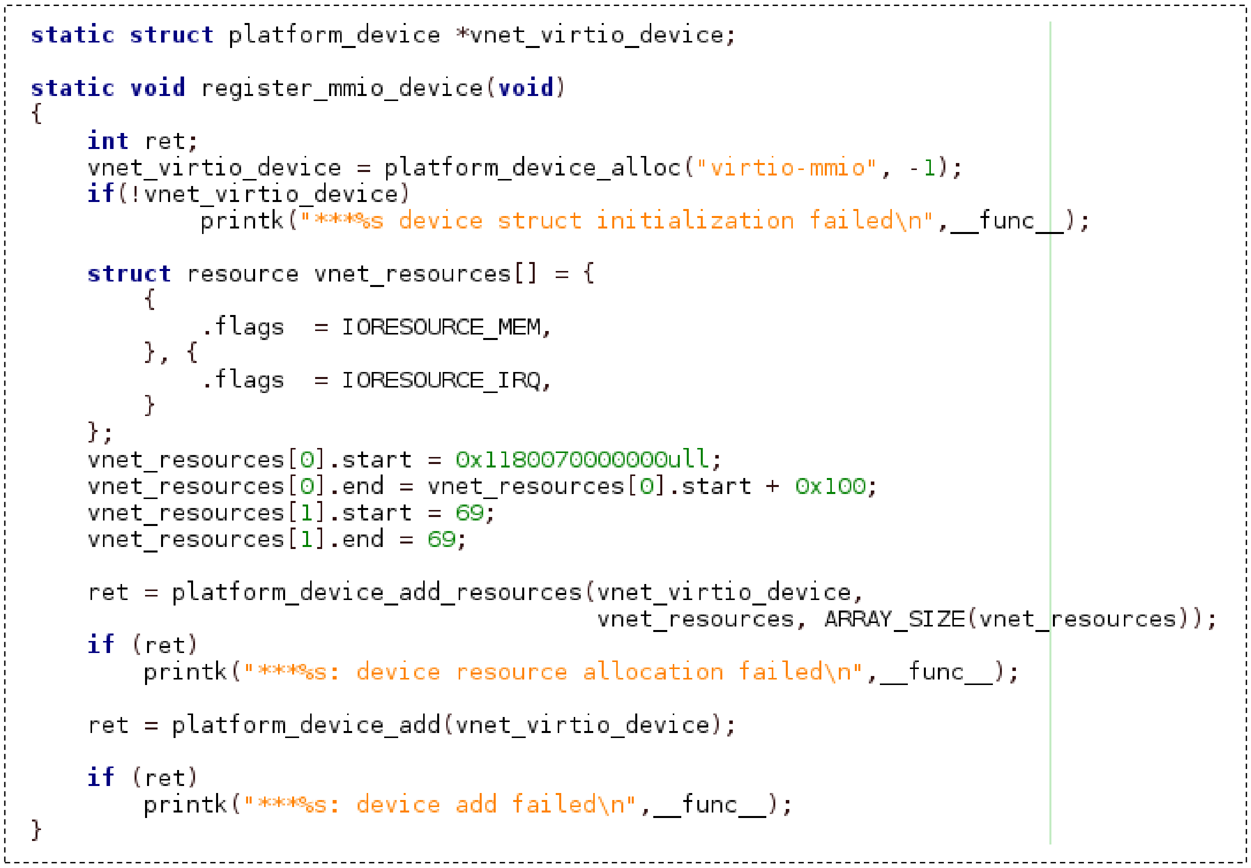
- Descriptor Table
- Available Ring
- Used Ring

These are contiguous in memory. To send buffer to the device, drivers fills a slot in descriptor table and write its index in to the available ring. After consuming the buffer, device writes its index to used ring and generates an interrupt.

These rings are created when driver probes the device. Device specifies the maximum number of buffers. After reading this value from the device, the driver create queue buffers and then share memory address of these buffers with the device using specific registers.

We used virtio-net for our network device implementation. In native kernels, virtio devices are implemented as standard PCI devices but emulation of PCI bus would be less efficient and more time consuming. We implemented MMIO(Memory Mapped Input Output) device which is faster and relatively easy to implement. For this purpose virtio-mmio driver is used which is a wrapper driver for MMIO based virtio devices. It performs all the functionality using MMIO instead of PCI bus.

To implement virtio-net as MMIO device, we need to register this device as platform device and specify its base address and interrupt line. Figure 17 shows the code snippet of function used to register network device as MMIO device.



```
static struct platform_device *vnet_virtio_device;

static void register_mmio_device(void)
{
    int ret;
    vnet_virtio_device = platform_device_alloc("virtio-mmio", -1);
    if(!vnet_virtio_device)
            printk("***%s device struct initialization failed\n",__func__);

    struct resource vnet_resources[] = {
        {
            .flags  = IORESOURCE_MEM,
        }, {
            .flags  = IORESOURCE_IRQ,
        }
    };
    vnet_resources[0].start = 0x1180070000000ull;
    vnet_resources[0].end = vnet_resources[0].start + 0x100;
    vnet_resources[1].start = 69;
    vnet_resources[1].end = 69;

    ret = platform_device_add_resources(vnet_virtio_device,
                                        vnet_resources, ARRAY_SIZE(vnet_resources));
    if (ret)
        printk("***%s: device resource allocation failed\n",__func__);

    ret = platform_device_add(vnet_virtio_device);

    if (ret)
        printk("***%s: device add failed\n",__func__);
}
```

FIGURE 17: Code Snippet Used for Registering Network Device as MMIO Device

6.2 VHOST-NET

For the backend driver implementation, vhost-net is used. Vhost net is a character device that can be used to reduce the number of system calls involved in virtio networking. User-space hypervisors are supported as well as kvm. The vhost drivers in Linux provide in-kernel virtio device emulation. The vhost-net driver emulates the virtio-net network card in the host kernel to reduce the number of system calls required for data communication. Vhost-net is the oldest vhost device and the only one which is available in mainline Linux. Experimental vhost-blk and vhost-scsi devices have also been developed.

When the hypervisor starts, it initializes vhost-net instance with several ioctl calls. A kernel thread is created called "vhost worker thread". The job of the worker thread is to handle

I/O events and perform the device emulation. Vhost does not emulate a complete virtio PCI adapter. Instead it restricts itself to virt-queue operations only.

The vhost worker thread waits for virtqueue kicks and then handles buffers that have been placed on the virtqueue. vhost-net takes packets from the tx virtqueue and transmitting them over the tap file descriptor. File descriptor polling is also done by the vhost worker thread. In vhost-net the worker thread wakes up when packets come in over the tap file descriptor and it places them into the rx virtqueue and calls the hypervisor using irqfd. Hypervisor then generates an interrupt to notify guest about packet availability.

When a guest notifies device after placing buffers onto a virtqueue, there needs to be a way to signal the vhost worker thread that there is work to do. To notify vhost about packet availability, hypervisor uses eventfd file descriptor which the vhost worker thread watches for activity.

On the return trip from the vhost worker thread to interrupting the guest a similar approach is used. Vhost takes a "call" file descriptor which it will write to in order to kick the guest. The vhost instance only knows about the guest memory mapping, a kick eventfd, and a call eventfd.

6.3 NETWORK DEVICE IN HYPERVISOR

For enabling the communication between the virtio-net and vhost-net, control signals needed to be sent and received. A network device is implemented in the hypervisor which passes the control signals in both directions.

Network device in hypervisor implements some device specific registers which are being used by guest driver for virtqueues sharing and device controlling. Device provides the maximum limit of buffers which is read by the virtio driver. After reading this value, virtio driver creates virtqueues and share them with the device. Once these queues are available, device share these queues with the vhost along with the eventfd for kick and call mechanism for each queue.

After creating queues, driver write VIRTIO_CONFIG_S_DRIVER_OK in status register of device which means driver acknowledges the device as valid device. After receiving acknowledgement from driver, device opens tap interface and share it descriptor with vhost to complete set-up of back-end driver. Once the setup is complete, vhost worker thread is created.

When virtio driver needs to transfer data, it fills one of the descriptors and notifies the device. Device in return kicks vhost using eventfd. After receiving kick from network device of hypervisor, vhost transfer data over network using Tap descriptor.

When vhost receives data, it place in one of the descriptor of receive queue and notifies the network device of hypervisor using irqfd. After receiving call from vhost, network device of hypervisor generates an interrupt and notifies guest about available data.

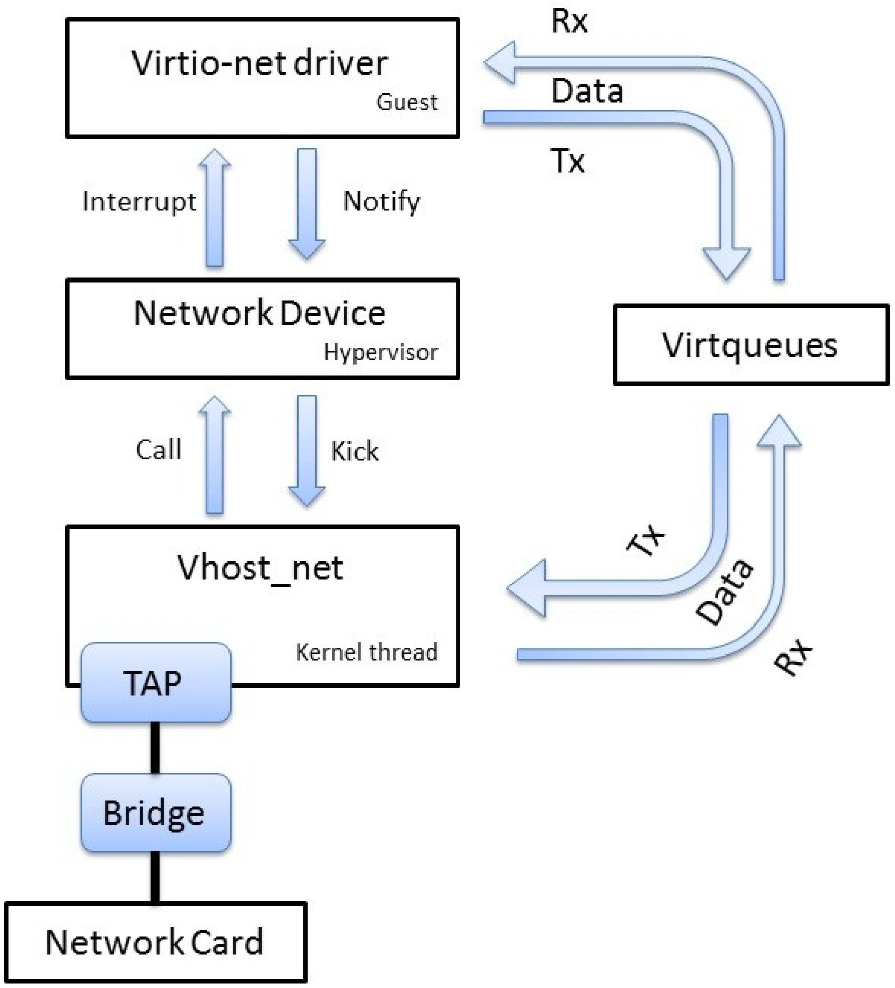


Figure 18: Communication between Virtio-Net, Vhost-Net and Network Device

6.4 EXECUTION FLOW OF NETWORKING

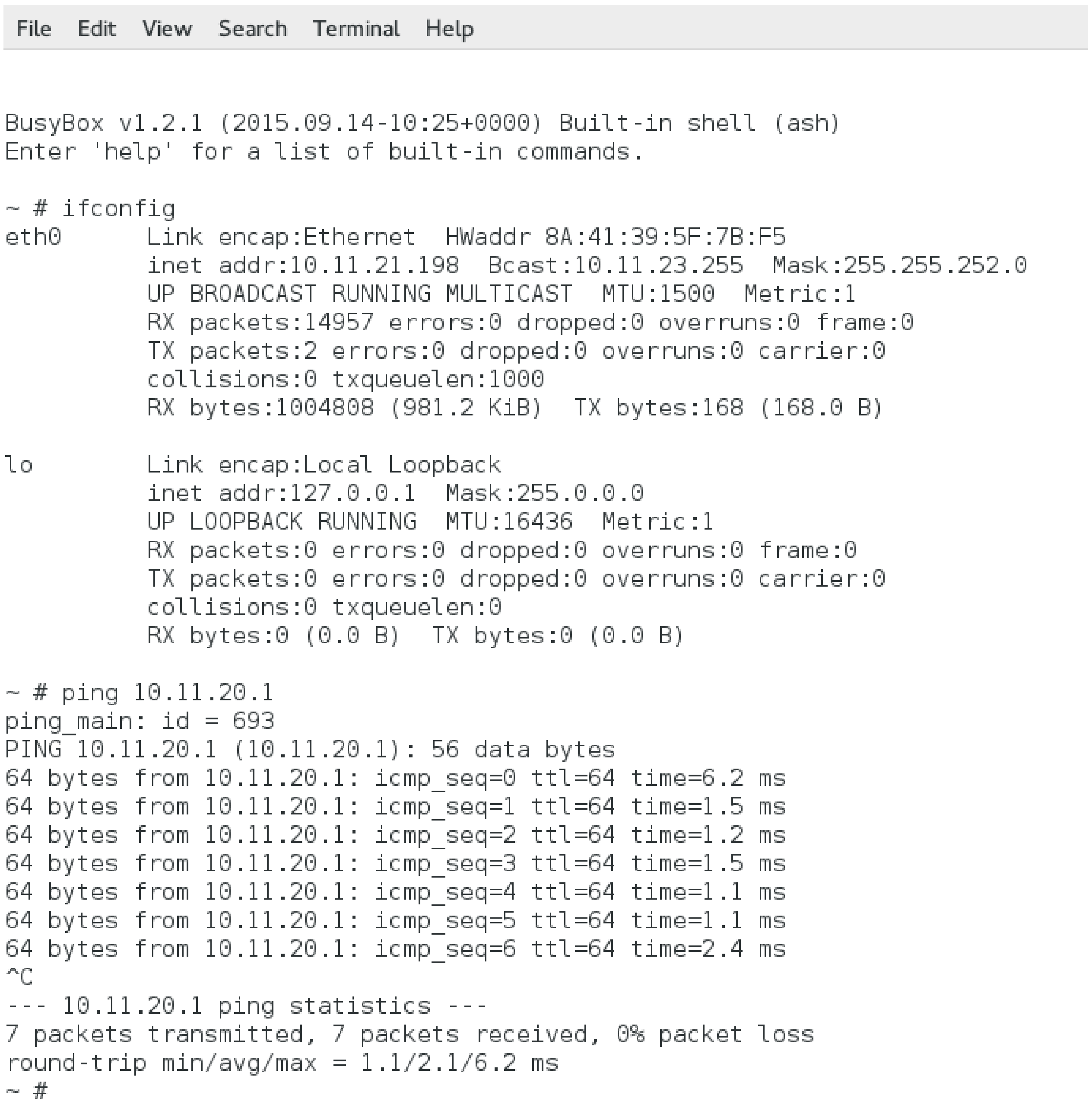
Execution flow of networking follows the steps described below.

1. Hypervisor opens vhost and make some initial settings.
2. The guest is informed that the virtio device is enabled.
3. Virtio provides the virt-queues to hypervisor's network device.
4. The hypervisor's network device shares it with the vhost.
5. Hypervisor's network device shares kick and call event fd for each queue.
6. Hypervisor's network device shares tap fd with vhost.
7. After receiving tap fd, vhost creates worker thread and starts polling event fd and tap fd.
8. After receiving packet, vhost places in descriptor of receive queue and notifies hypervisor's network device using irqfd.
9. Hypervisor' network device generates interrupt to notify guest.
10. To send packet, virtio driver fills one of descriptor of transfer queue and notify device.
11. Hypervisor's network device kicks vhost using eventfd.
12. Vhost transmits packet over tap fd.

Figure 18 shows the communication between the units involved in network implementation.

6.5 TEST CASE

Figure 19 shows command prompt of guest. The figure shows that there is an active network device which is detected in the guest. That device is configured using "`ifconfig`" command. The available devices are also shown as "eth0" and "lo" by executing "`ifconfig`". The "eth0" interface is assigned a valid IP. Ping to an in-net address is also successful, showing that the system can receive and transmit data.



```
BusyBox v1.2.1 (2015.09.14-10:25+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 8A:41:39:5F:7B:F5
          inet addr:10.11.21.198  Bcast:10.11.23.255  Mask:255.255.252.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14957 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1004808 (981.2 KiB)  TX bytes:168 (168.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

~ # ping 10.11.20.1
ping_main: id = 693
PING 10.11.20.1 (10.11.20.1): 56 data bytes
64 bytes from 10.11.20.1: icmp_seq=0 ttl=64 time=6.2 ms
64 bytes from 10.11.20.1: icmp_seq=1 ttl=64 time=1.5 ms
64 bytes from 10.11.20.1: icmp_seq=2 ttl=64 time=1.2 ms
64 bytes from 10.11.20.1: icmp_seq=3 ttl=64 time=1.5 ms
64 bytes from 10.11.20.1: icmp_seq=4 ttl=64 time=1.1 ms
64 bytes from 10.11.20.1: icmp_seq=5 ttl=64 time=1.1 ms
64 bytes from 10.11.20.1: icmp_seq=6 ttl=64 time=2.4 ms
^C
--- 10.11.20.1 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 1.1/2.1/6.2 ms
~ #
```

FIGURE 19: Active Network Device in Guest

7 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.

7.1 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. Up to current deliverable, our focus is on the test cases discussed in following subsections.

7.1.1 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.

7.1.2 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".

7.1.3 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.

7.1.4 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 blocks 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.

7.2 Testing with SMP Support

Testing and Debugging with single core was much easier. But for multi-core the testing and debugging has become a difficult task. As every core executes its piece of code, the corresponding log file is written separately for each of them. Each log file for every processor contains the original instruction block (guest instructions), its corresponding translated block (host instructions) as well as a state of all GP registers and CP0 registers. This information is enough to check whether the corresponding block executed correctly or not. The original and translated blocks in the log verify the correctness of translation. Remaining information is used to determine whether corresponding instruction is correctly emulated or not.

7.3 BOOTING WITH CUSTOMIZED MINIMAL INITRAMFS FILE

The file system upon which the root directory can be mounted and which contains the files necessary to bring the system to a state where other file systems can be mounted and user space daemons and applications are started, is called rootfs. The kernel boot process concludes with the init code (see init/main.c) whose primary purpose is to create and populate an initial root file system with a set of directories and files. It then tries to launch the first user mode process to run an executable file found on this initial file system. This first process ("init") is always given process ID 1. Once the init process is started it typically begins to launch other user space programs. On a desktop or server system this is known as the sysvinit process and includes the set of scripts found (typically) under /etc/rc.d.

In default rootfs file, most of the scripts are used to initialize and communicate with external devices. As we have not yet implemented these devices, the default roofts cannot be used. So, we have created our minimal initramfs, which only have console as mounted device and contains no initializing scripts. This file performs the basic I/O operations like creating, reading, writing a file and process creation. The file doesn't involve any external communication with the devices.

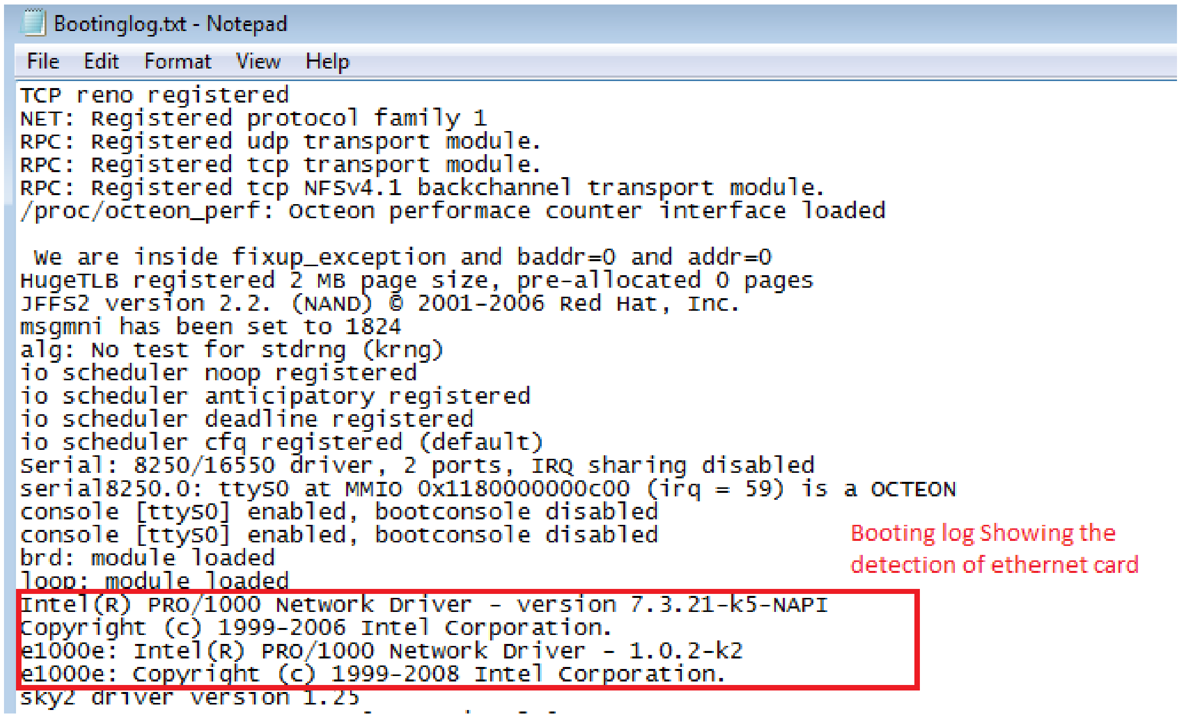
We have also executed the same binary containing our customized minimal initramfs on CN57XX Cavium MIPS64 board. It shows the same output on the console as our hypervisor.

7.4 VIRTUAL ETHERNET CARD DETECTION

For Ethernet card detection, we configured the kernel with networking and executed the guest. The hypervisor successfully detects the virtual Ethernet card, as shown in the Figure 20. It detects the standard e1000 network interface.

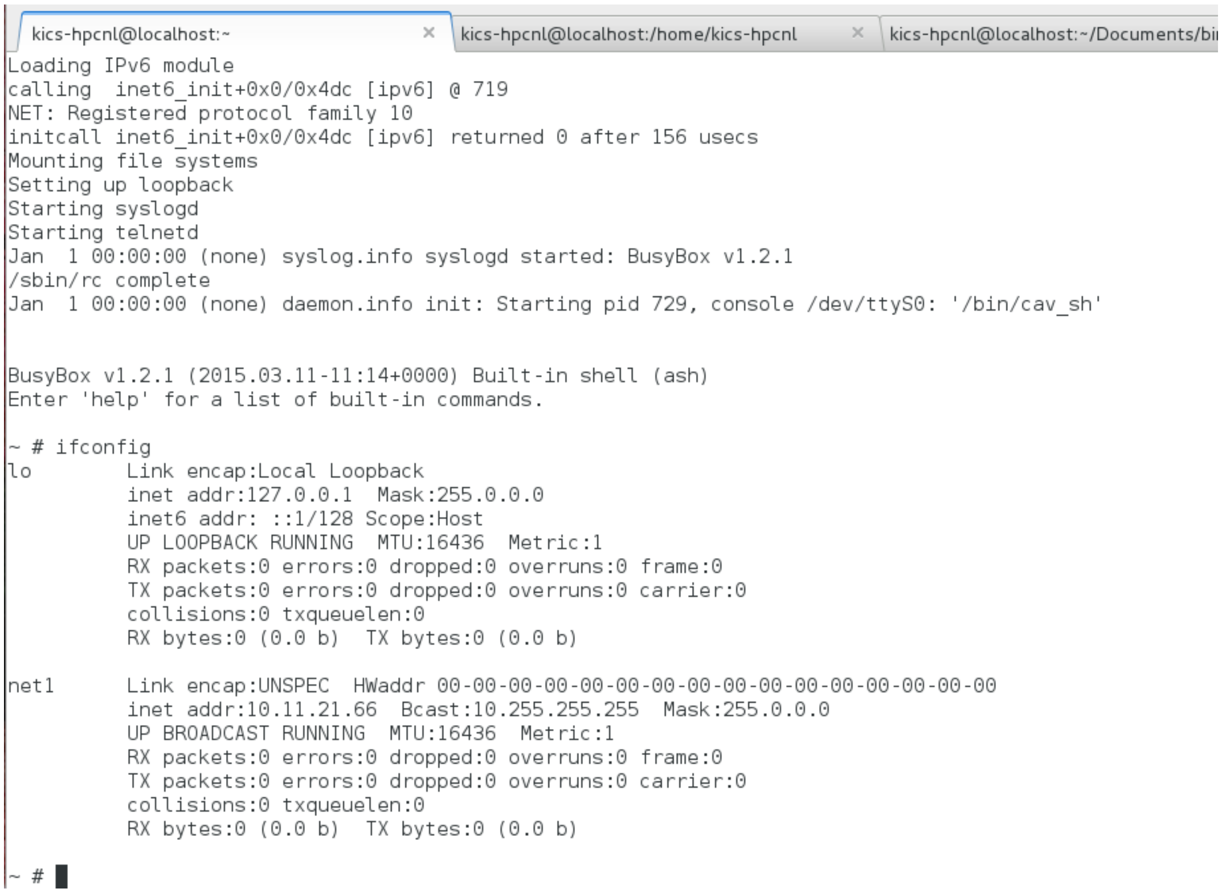
7.5 ASSIGNING A VALID IP TO GUEST

A separate guest network module is created for enabling networking in the kernel. This module will initialize a network interface for the guest. It is initialized during the kernel booting and the networked is configured (i.e. assigned an IP) in the kernel's initializing script. So that when the kernel is booted and command prompt appears, we can verify the network interface with "`ifconfig`" command. This command lists all the available network interfaces. The Figure 21 shows the result of "`ifconfig`" command after the kernel booting. The guest's console output shows two network interfaces, one is loopback "lo" and other is "net1" which is guest's network interface with a valid inet IP.


Bootinglog.txt - Notepad
File Edit Format View Help
Booting log Showing the detection of ethernet card

```
TCP reno registered
NET: Registered protocol family 1
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
/proc/octeon_perf: Octeon performace counter interface loaded

 We are inside fixup_exception and baddr=0 and addr=0
HugeTLB registered 2 MB page size, pre-allocated 0 pages
JFFS2 version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
msgmni has been set to 1824
alg: No test for stdrng (krng)
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: 8250/16550 driver, 2 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x1180000000c00 (irq = 59) is a OCTEON
console [ttyS0] enabled, bootconsole disabled
console [ttyS0] enabled, bootconsole disabled
brd: module loaded
loop: module loaded
Intel(R) PRO/1000 Network Driver - version 7.3.21-k5-NAPI
Copyright (c) 1999-2006 Intel Corporation.
e1000e: Intel(R) PRO/1000 Network Driver - 1.0.2-k2
e1000e: Copyright (c) 1999-2008 Intel Corporation.
sky2 driver version 1.25
```

FIGURE 20: Booting Log of Hypervisor, Showing the Detection of Ethernet


kics-hpcnl@localhost:~
kics-hpcnl@localhost:/home/kics-hpcnl
kics-hpcnl@localhost:~/Documents/bi

```
Loading IPv6 module
calling  inet6_init+0x0/0x4dc [ipv6] @ 719
NET: Registered protocol family 10
initcall inet6_init+0x0/0x4dc [ipv6] returned 0 after 156 usecs
Mounting file systems
Setting up loopback
Starting syslogd
Starting telnetd
Jan  1 00:00:00 (none) syslog.info syslogd started: BusyBox v1.2.1
/sbin/rc complete
Jan  1 00:00:00 (none) daemon.info init: Starting pid 729, console /dev/ttyS0: '/bin/cav_sh'


BusyBox v1.2.1 (2015.03.11-11:14+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

net1      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:10.11.21.66  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

~ #
```

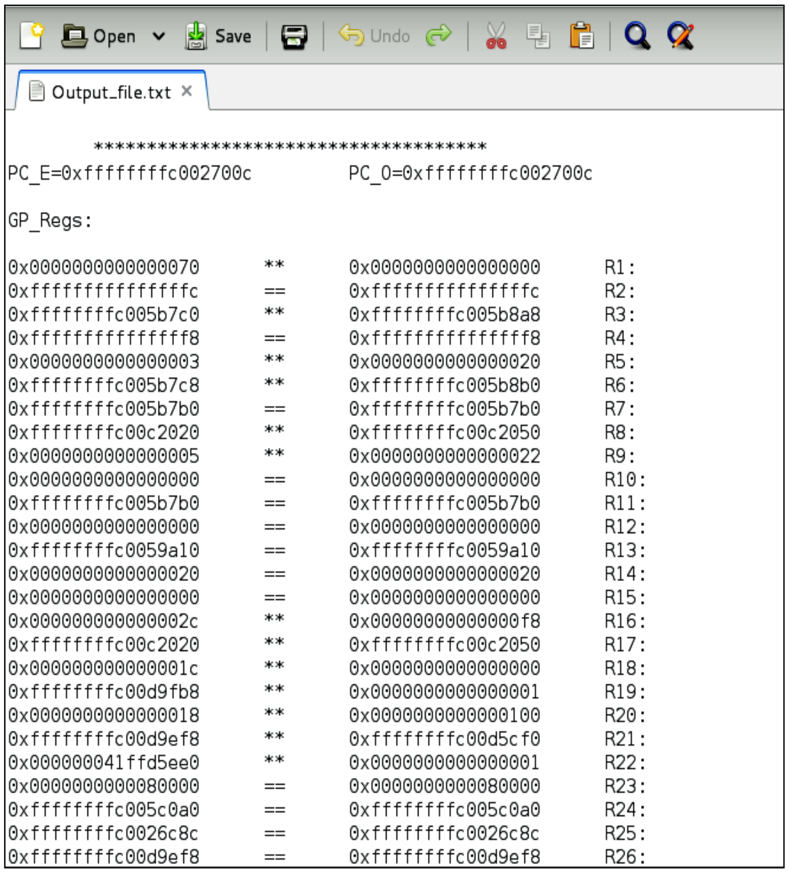
Figure 21: Guest's Network Interfaces

8 TEST RESULTS

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

8.1 OUTPUT OF SYSTEM STATE MATCHING TEST

We trap at every instruction to create a state-file. This state-file is matched with QEMU log state-file to see if any register contains different contents. Mismatches are written in other file as shown in Figure 22.

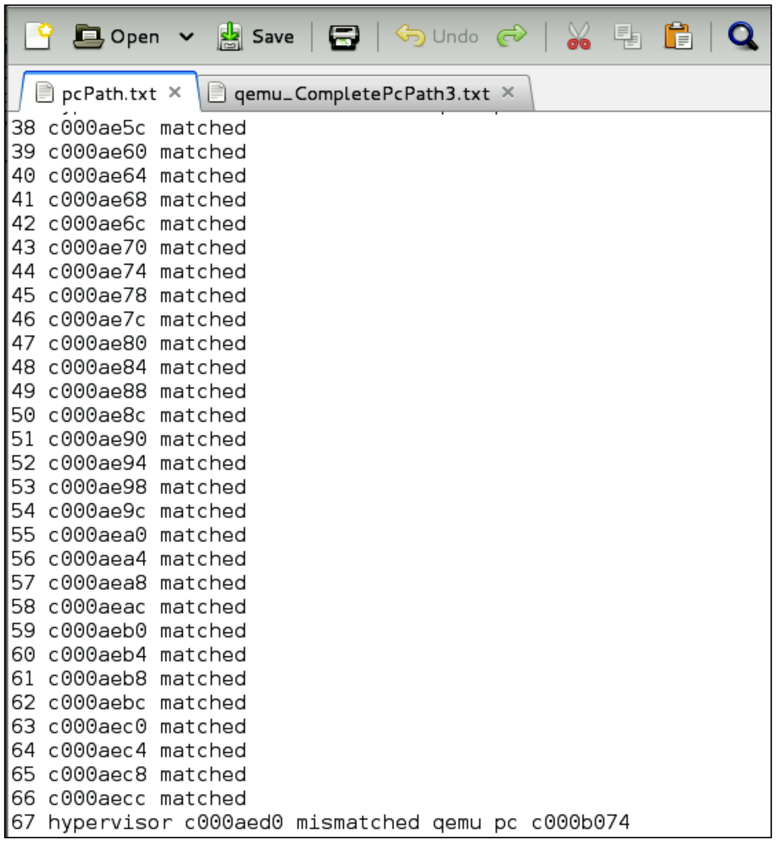


```
       ***********************************
PC_E=0xffffffffc002700c         PC_O=0xffffffffc002700c

GP_Regs:

0x0000000000000070      **      0x0000000000000000      R1:
0xfffffffffffffffc      ==      0xfffffffffffffffc      R2:
0xffffffffc005b7c0      **      0xffffffffc005b8a8      R3:
0xfffffffffffffff8      ==      0xfffffffffffffff8      R4:
0x0000000000000003      **      0x0000000000000020      R5:
0xffffffffc005b7c8      **      0xffffffffc005b8b0      R6:
0xffffffffc005b7b0      ==      0xffffffffc005b7b0      R7:
0xffffffffc00c2020      **      0xffffffffc00c2050      R8:
0x0000000000000005      **      0x0000000000000022      R9:
0x0000000000000000      ==      0x0000000000000000      R10:
0xffffffffc005b7b0      ==      0xffffffffc005b7b0      R11:
0x0000000000000000      ==      0x0000000000000000      R12:
0xffffffffc0059a10      ==      0xffffffffc0059a10      R13:
0x0000000000000020      ==      0x0000000000000020      R14:
0x0000000000000000      ==      0x0000000000000000      R15:
0x000000000000002c      **      0x00000000000000f8      R16:
0xffffffffc00c2020      **      0xffffffffc00c2050      R17:
0x000000000000001c      **      0x0000000000000000      R18:
0xffffffffc00d9fb8      **      0x0000000000000001      R19:
0x0000000000000018      **      0x0000000000000100      R20:
0xffffffffc00d9ef8      **      0xffffffffc00d5cf0      R21:
0x000000041ffd5ee0      **      0x0000000000000001      R22:
0x0000000000080000      ==      0x0000000000080000      R23:
0xffffffffc005c0a0      ==      0xffffffffc005c0a0      R24:
0xffffffffc0026c8c      ==      0xffffffffc0026c8c      R25:
0xffffffffc00d9ef8      ==      0xffffffffc00d9ef8      R26:
```

FIGURE 22: Output of system state matching test

8.2 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 23.

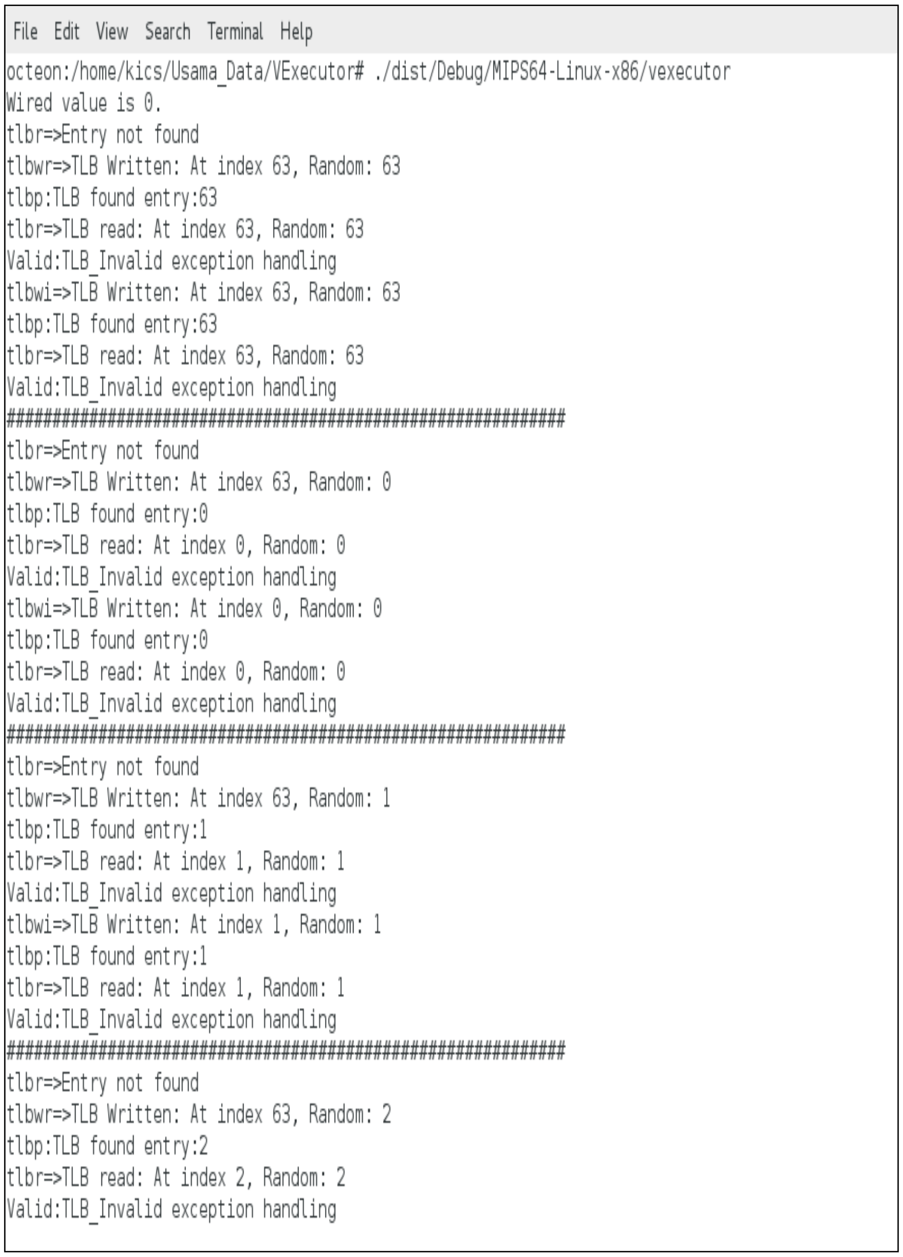


```
38 c000ae5c matched
39 c000ae60 matched
40 c000ae64 matched
41 c000ae68 matched
42 c000ae6c matched
43 c000ae70 matched
44 c000ae74 matched
45 c000ae78 matched
46 c000ae7c matched
47 c000ae80 matched
48 c000ae84 matched
49 c000ae88 matched
50 c000ae8c matched
51 c000ae90 matched
52 c000ae94 matched
53 c000ae98 matched
54 c000ae9c matched
55 c000aea0 matched
56 c000aea4 matched
57 c000aea8 matched
58 c000aeac matched
59 c000aeb0 matched
60 c000aeb4 matched
61 c000aeb8 matched
62 c000aebc matched
63 c000aec0 matched
64 c000aec4 matched
65 c000aec8 matched
66 c000aecc matched
67 hypervisor c000aed0 mismatched qemu pc c000b074
```

FIGURE 23: Output of execution path test

8.3 Output of TLB Testing

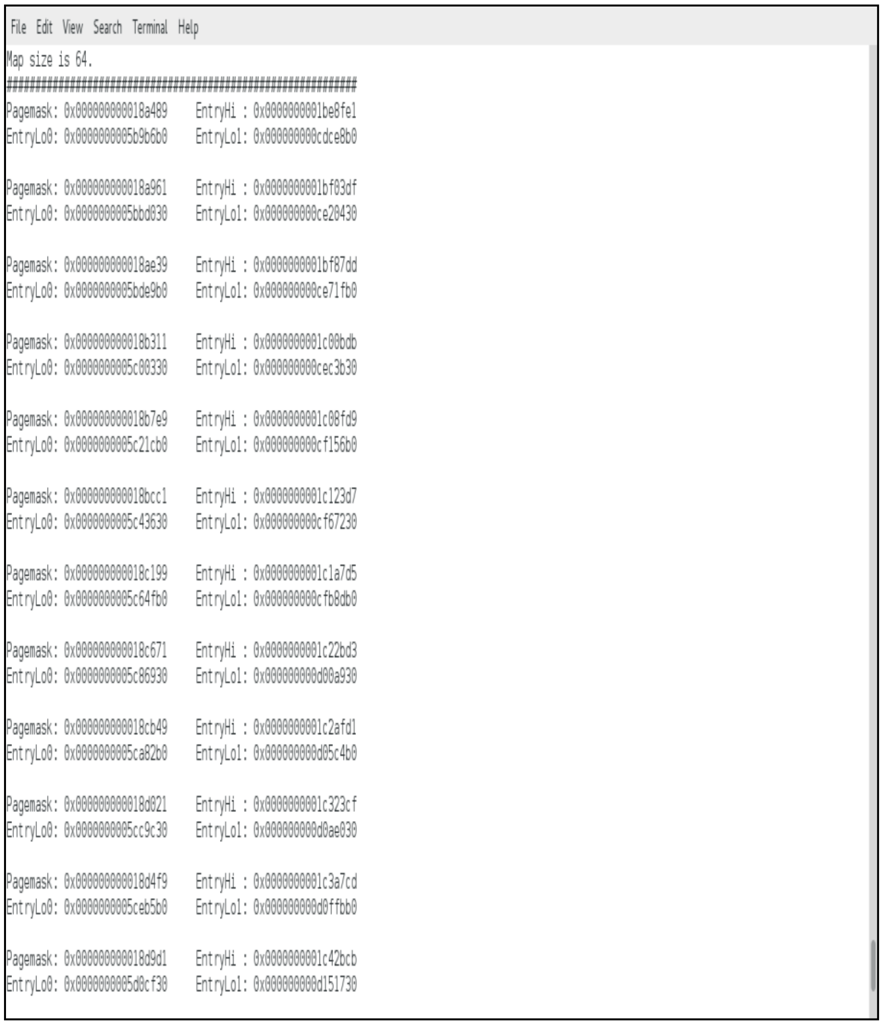
To test TLB mechanism, random TLB entries are generated and searched in TLB. A TLB miss is obvious because the entry is newly generated. Hence, probe bit is set and TLB write-random function is called to place this entry at the index present in CP0 random register. Random register is incremented and entry is searched again. On TLB hit, we call TLB read to fetch the entry from the index set by TLB probe, as shown in Figure 24.


File Edit View Search Terminal Help

```
octeon:/home/kics/Usama_Data/VExecutor# ./dist/Debug/MIPS64-Linux-x86/vexecutor
Wired value is 0.
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 63
tlbp:TLB found entry:63
tlbr=>TLB read: At index 63, Random: 63
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 63, Random: 63
tlbp:TLB found entry:63
tlbr=>TLB read: At index 63, Random: 63
Valid:TLB_Invalid exception handling
#############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 0
tlbp:TLB found entry:0
tlbr=>TLB read: At index 0, Random: 0
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 0, Random: 0
tlbp:TLB found entry:0
tlbr=>TLB read: At index 0, Random: 0
Valid:TLB_Invalid exception handling
#############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 1
tlbp:TLB found entry:1
tlbr=>TLB read: At index 1, Random: 1
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 1, Random: 1
tlbp:TLB found entry:1
tlbr=>TLB read: At index 1, Random: 1
Valid:TLB_Invalid exception handling
#############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 2
tlbp:TLB found entry:2
tlbr=>TLB read: At index 2, Random: 2
Valid:TLB_Invalid exception handling
```

FIGURE 24: Searching for random TLB

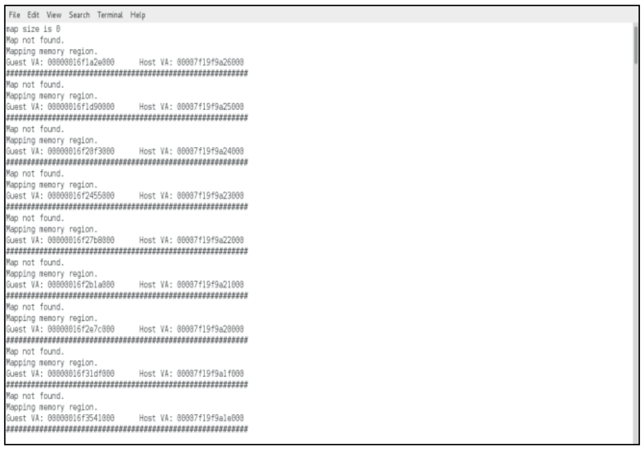
Then TLB write-index function is called that writes TLB entry at the index present in index register. As index register was set by TLB probe, it writes the entry at same index that was previously written by TLB write-random. TLB probe and TLB read are called again and then a new random entry is generated. This process is repeated 640 times.



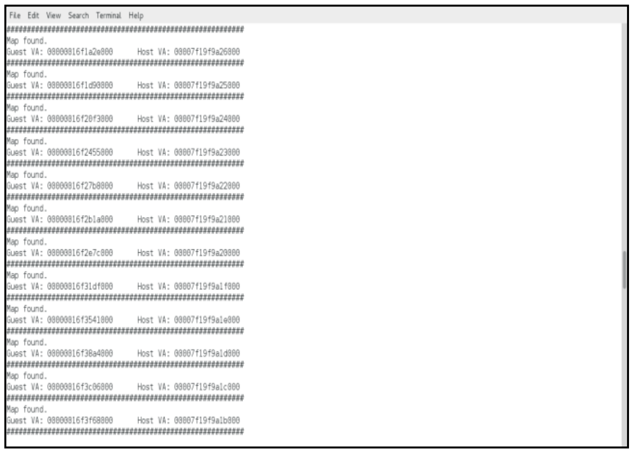
```
Map size is 64.
########################################################
Pagemask: 0x000000000018a489    EntryHi : 0x000000001be8fe1
EntryLo0: 0x0000000005b9b6b0    EntryLo1: 0x000000000cdce8b0

Pagemask: 0x000000000018a961    EntryHi : 0x000000001bf03df
EntryLo0: 0x0000000005bbd030    EntryLo1: 0x000000000ce20430

Pagemask: 0x000000000018ae39    EntryHi : 0x000000001bf87dd
EntryLo0: 0x0000000005bde9b0    EntryLo1: 0x000000000ce71fb0

Pagemask: 0x000000000018b311    EntryHi : 0x000000001c00bdb
EntryLo0: 0x0000000005c00330    EntryLo1: 0x000000000cec3b30

Pagemask: 0x000000000018b7e9    EntryHi : 0x000000001c08fd9
EntryLo0: 0x0000000005c21cb0    EntryLo1: 0x000000000cf156b0

Pagemask: 0x000000000018bcc1    EntryHi : 0x000000001c123d7
EntryLo0: 0x0000000005c43630    EntryLo1: 0x000000000cf67230

Pagemask: 0x000000000018c199    EntryHi : 0x000000001c1a7d5
EntryLo0: 0x0000000005c64fb0    EntryLo1: 0x000000000cfb8db0

Pagemask: 0x000000000018c671    EntryHi : 0x000000001c22bd3
EntryLo0: 0x0000000005c86930    EntryLo1: 0x000000000d00a930

Pagemask: 0x000000000018cb49    EntryHi : 0x000000001c2afd1
EntryLo0: 0x0000000005ca82b0    EntryLo1: 0x000000000d05c4b0

Pagemask: 0x000000000018d021    EntryHi : 0x000000001c323cf
EntryLo0: 0x0000000005cc9c30    EntryLo1: 0x000000000d0ae030

Pagemask: 0x000000000018d4f9    EntryHi : 0x000000001c3a7cd
EntryLo0: 0x0000000005ceb5b0    EntryLo1: 0x000000000d0ffbb0

Pagemask: 0x000000000018d9d1    EntryHi : 0x000000001c42bcb
EntryLo0: 0x0000000005d0cf30    EntryLo1: 0x000000000d151730
```

FIGURE 25: TLB entries in TLB table

As TLB could have 64 entries at max, additional entries require a replacement policy. After setting all entries, TLB entries are printed, as shown in figure 25. To test page table, a random GVA is generated and searched in the page table. Obviously, there is no matching entry in page table because this is the newly generated address. Hence, it maps a new memory region and returns its address. This process is repeated several times. Each time it maps a new region, places translation in page table and returns translated address. The output is shown in figure 26(a). After creating appropriate entries in page table, same process is repeated again for all the generated addresses and we get valid translation now, as shown in figure 26(b). Then whole page table is printed in figure 26(c) and reverse page table, shown in figure 26(d), is also managed to use for future testing of hypervisor.



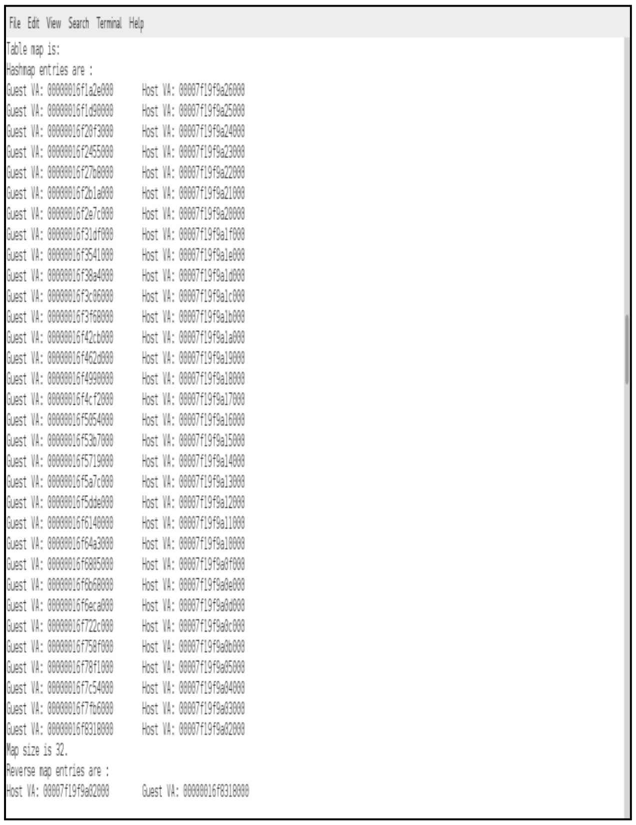
```
map size is 0
Map not found.
Mapping memory region.
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f2e7c000      Host VA: 00007f19f9a20000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f31df000      Host VA: 00007f19f9a1f000
############################################################
Map not found.
Mapping memory region.
Guest VA: 00000016f3541000      Host VA: 00007f19f9a1e000
############################################################
```



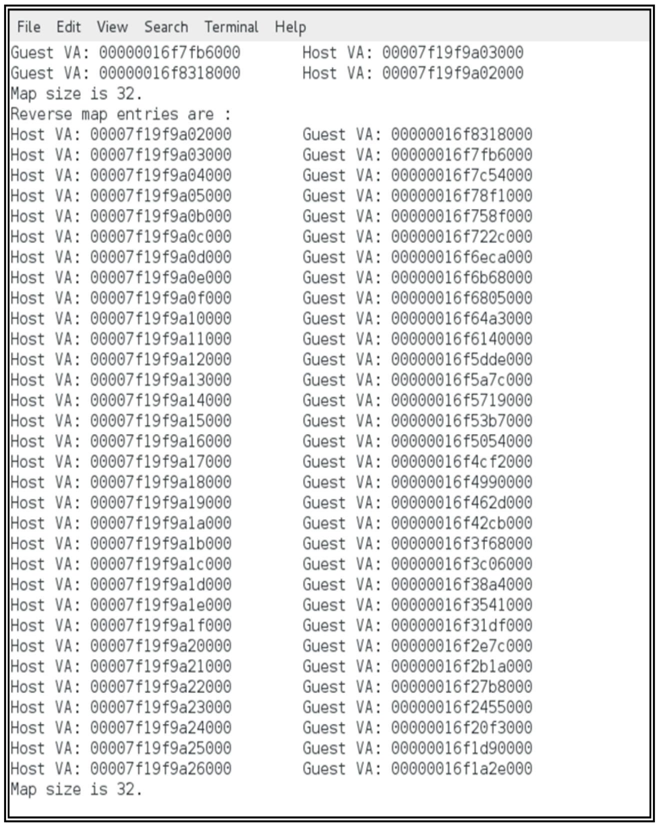
```
############################################################
Map found.
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
############################################################
Map found.
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
############################################################
Map found.
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
############################################################
Map found.
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
############################################################
Map found.
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
############################################################
Map found.
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
############################################################
Map found.
Guest VA: 00000016f2e7c000      Host VA: 00007f19f9a20000
############################################################
Map found.
Guest VA: 00000016f31df000      Host VA: 00007f19f9a1f000
############################################################
Map found.
Guest VA: 00000016f3541000      Host VA: 00007f19f9a1e000
############################################################
Map found.
Guest VA: 00000016f38a4000      Host VA: 00007f19f9a1d000
############################################################
Map found.
Guest VA: 00000016f3c06000      Host VA: 00007f19f9a1c000
############################################################
Map found.
Guest VA: 00000016f3f68000      Host VA: 00007f19f9a1b000
############################################################
```

(a)

(b)



```
Table map is:
Hashmap entries are :
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
Guest VA: 00000016f2e7c000      Host VA: 00007f19f9a20000
Guest VA: 00000016f31df000      Host VA: 00007f19f9a1f000
Guest VA: 00000016f3541000      Host VA: 00007f19f9a1e000
Guest VA: 00000016f38a4000      Host VA: 00007f19f9a1d000
Guest VA: 00000016f3c06000      Host VA: 00007f19f9a1c000
Guest VA: 00000016f3f68000      Host VA: 00007f19f9a1b000
Guest VA: 00000016f42cb000      Host VA: 00007f19f9a1a000
Guest VA: 00000016f462d000      Host VA: 00007f19f9a19000
Guest VA: 00000016f4990000      Host VA: 00007f19f9a18000
Guest VA: 00000016f4cf2000      Host VA: 00007f19f9a17000
Guest VA: 00000016f5054000      Host VA: 00007f19f9a16000
Guest VA: 00000016f53b7000      Host VA: 00007f19f9a15000
Guest VA: 00000016f5719000      Host VA: 00007f19f9a14000
Guest VA: 00000016f5a7c000      Host VA: 00007f19f9a13000
Guest VA: 00000016f5dde000      Host VA: 00007f19f9a12000
Guest VA: 00000016f6140000      Host VA: 00007f19f9a11000
Guest VA: 00000016f64a3000      Host VA: 00007f19f9a10000
Guest VA: 00000016f6805000      Host VA: 00007f19f9a0f000
Guest VA: 00000016f6b68000      Host VA: 00007f19f9a0e000
Guest VA: 00000016f6eca000      Host VA: 00007f19f9a0d000
Guest VA: 00000016f722c000      Host VA: 00007f19f9a0c000
Guest VA: 00000016f758f000      Host VA: 00007f19f9a0b000
Guest VA: 00000016f78f1000      Host VA: 00007f19f9a05000
Guest VA: 00000016f7c54000      Host VA: 00007f19f9a04000
Guest VA: 00000016f7fb6000      Host VA: 00007f19f9a03000
Guest VA: 00000016f8318000      Host VA: 00007f19f9a02000
Map size is 32.
Reverse map entries are :
Host VA: 00007f19f9a02000       Guest VA: 00000016f8318000
```



```
Guest VA: 00000016f7fb6000      Host VA: 00007f19f9a03000
Guest VA: 00000016f8318000      Host VA: 00007f19f9a02000
Map size is 32.
Reverse map entries are :
Host VA: 00007f19f9a02000       Guest VA: 00000016f8318000
Host VA: 00007f19f9a03000       Guest VA: 00000016f7fb6000
Host VA: 00007f19f9a04000       Guest VA: 00000016f7c54000
Host VA: 00007f19f9a05000       Guest VA: 00000016f78f1000
Host VA: 00007f19f9a0b000       Guest VA: 00000016f758f000
Host VA: 00007f19f9a0c000       Guest VA: 00000016f722c000
Host VA: 00007f19f9a0d000       Guest VA: 00000016f6eca000
Host VA: 00007f19f9a0e000       Guest VA: 00000016f6b68000
Host VA: 00007f19f9a0f000       Guest VA: 00000016f6805000
Host VA: 00007f19f9a10000       Guest VA: 00000016f64a3000
Host VA: 00007f19f9a11000       Guest VA: 00000016f6140000
Host VA: 00007f19f9a12000       Guest VA: 00000016f5dde000
Host VA: 00007f19f9a13000       Guest VA: 00000016f5a7c000
Host VA: 00007f19f9a14000       Guest VA: 00000016f5719000
Host VA: 00007f19f9a15000       Guest VA: 00000016f53b7000
Host VA: 00007f19f9a16000       Guest VA: 00000016f5054000
Host VA: 00007f19f9a17000       Guest VA: 00000016f4cf2000
Host VA: 00007f19f9a18000       Guest VA: 00000016f4990000
Host VA: 00007f19f9a19000       Guest VA: 00000016f462d000
Host VA: 00007f19f9a1a000       Guest VA: 00000016f42cb000
Host VA: 00007f19f9a1b000       Guest VA: 00000016f3f68000
Host VA: 00007f19f9a1c000       Guest VA: 00000016f3c06000
Host VA: 00007f19f9a1d000       Guest VA: 00000016f38a4000
Host VA: 00007f19f9a1e000       Guest VA: 00000016f3541000
Host VA: 00007f19f9a1f000       Guest VA: 00000016f31df000
Host VA: 00007f19f9a20000       Guest VA: 00000016f2e7c000
Host VA: 00007f19f9a21000       Guest VA: 00000016f2b1a000
Host VA: 00007f19f9a22000       Guest VA: 00000016f27b8000
Host VA: 00007f19f9a23000       Guest VA: 00000016f2455000
Host VA: 00007f19f9a24000       Guest VA: 00000016f20f3000
Host VA: 00007f19f9a25000       Guest VA: 00000016f1d90000
Host VA: 00007f19f9a26000       Guest VA: 00000016f1a2e000
Map size is 32.
```

(c)

(d)

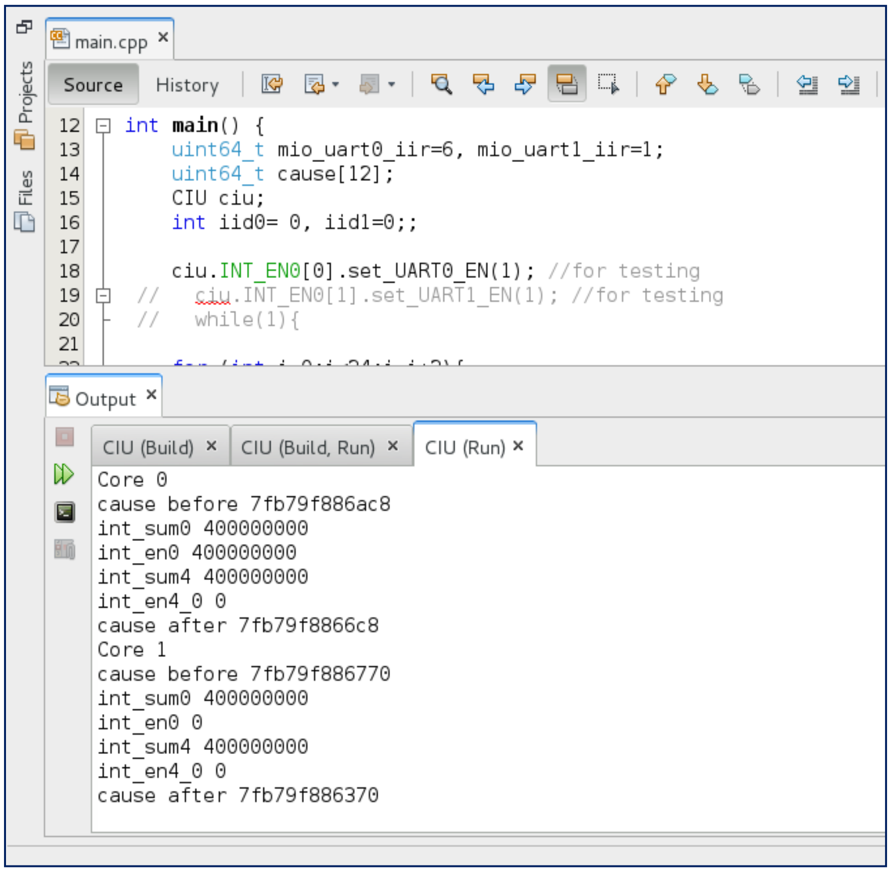
FIGURE 26: Output of TLB and page table testing.

Searching entries in (a) empty page table, and (b) page table having valid entries. (c) whole page table with valid translation. (d) : reverse mapping of page table.

8.4 OUTPUT OF CIU TESTING

Artificial UART registers are read to test the code. UART registers were set to see the effect on the 10, 11 and 12 bits of cause register. If Interrupt ID (IID) field of IIR is 1 than there is no pending interrupt request. Otherwise, it represents the ID of pending interrupt. In actual system enable register is set by the system but here we are setting it explicitly. The cause register is initialized with garbage value every time because CIU will only change the 9, 10 and 11 bits of cause register.

In source code of figure 27, mio_uart0 IIR register is set to 6 to show that "Receiver line status" interrupt is present. Similarly, mio_uart1 IIR register is set to 1 to represent no interrupt. Only core0's enable register is set. And all the other cores have disabled the hardware interrupts. Output for core 0 in figure 27 shows that initially cause register is initialized by a garbage value.



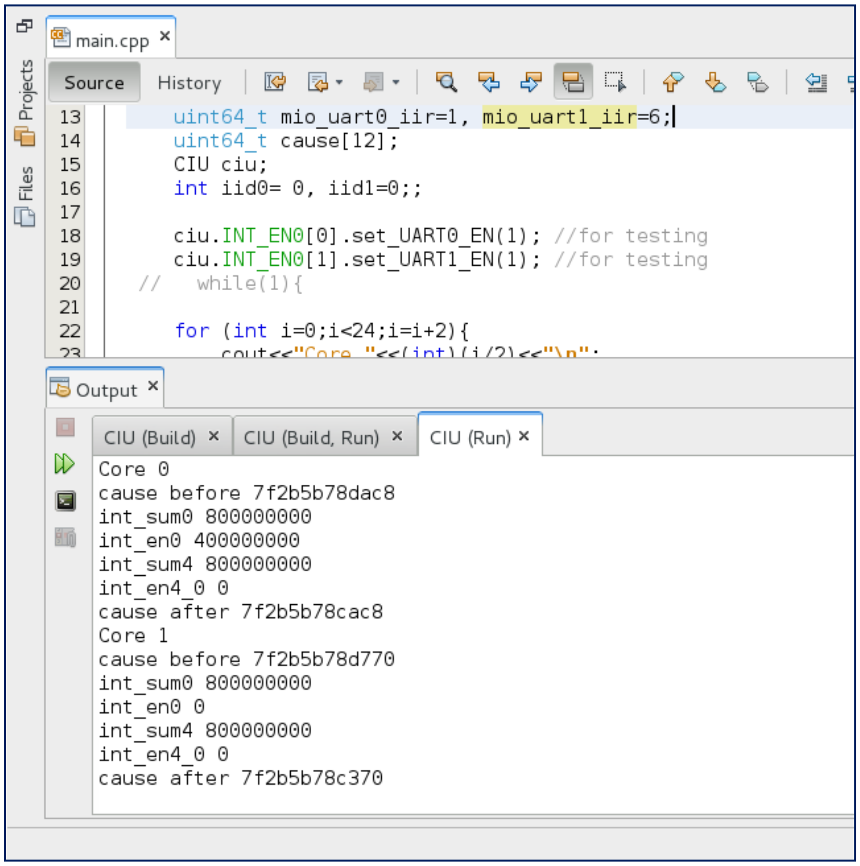
```
int main() {
    uint64_t mio_uart0_iir=6, mio_uart1_iir=1;
    uint64_t cause[12];
    CIU ciu;
    int iid0= 0, iid1=0;;

    ciu.INT_EN0[0].set_UART0_EN(1); //for testing
//   ciu.INT_EN0[1].set_UART1_EN(1); //for testing
//   while(1){
```

```
Core 0
cause before 7fb79f886ac8
int_sum0 400000000
int_en0 400000000
int_sum4 400000000
int_en4_0 0
cause after 7fb79f8866c8
Core 1
cause before 7fb79f886770
int_sum0 400000000
int_en0 0
int_sum4 400000000
int_en4_0 0
cause after 7fb79f886370
```

FIGURE 27: Output of CIU. No pending interrupt on core 1

The summary register's 34th bit (uart 0) is set, making it 400000000. Corresponding 34th bit in enable register is also set, which means that the 9th bit of cause will be set. The enable register for 10th and 11th bit are zero, so cause bits would be cleared. Initially the xxxxxxxx6ac8 is changed to xxxxxxxx66c8 by setting 9th bit and clearing 10th and 11th bit. For core 1, as none of the enable registers are configured so three bits would be cleared i.e. xxxxxxxx6770 changes to xxxxxxxx6370.

In figure 28, uart0 has no interrupt and uart1 is receiving an interrupt with id 6. For core0, bit 9 and 10 of enable register is set and cause register is get initialized with garbage value. As summary register shows the presence of an interrupt and bit 35 is set, it means that uart1 interrupt is present. Its enable register should also be set for uart1, in order to pass on the pending interrupt. Hence, bit 9 and 11 will be cleared and bit 10 will be set for core 0 i.e. xxxxxxxxxdac8 changes to xxxxxxxxxcac8 in the output. For core1, nothing is enabled so all three bits would be cleared i.e. xxxxxxxxd770 changes to xxxxxxxxc370.



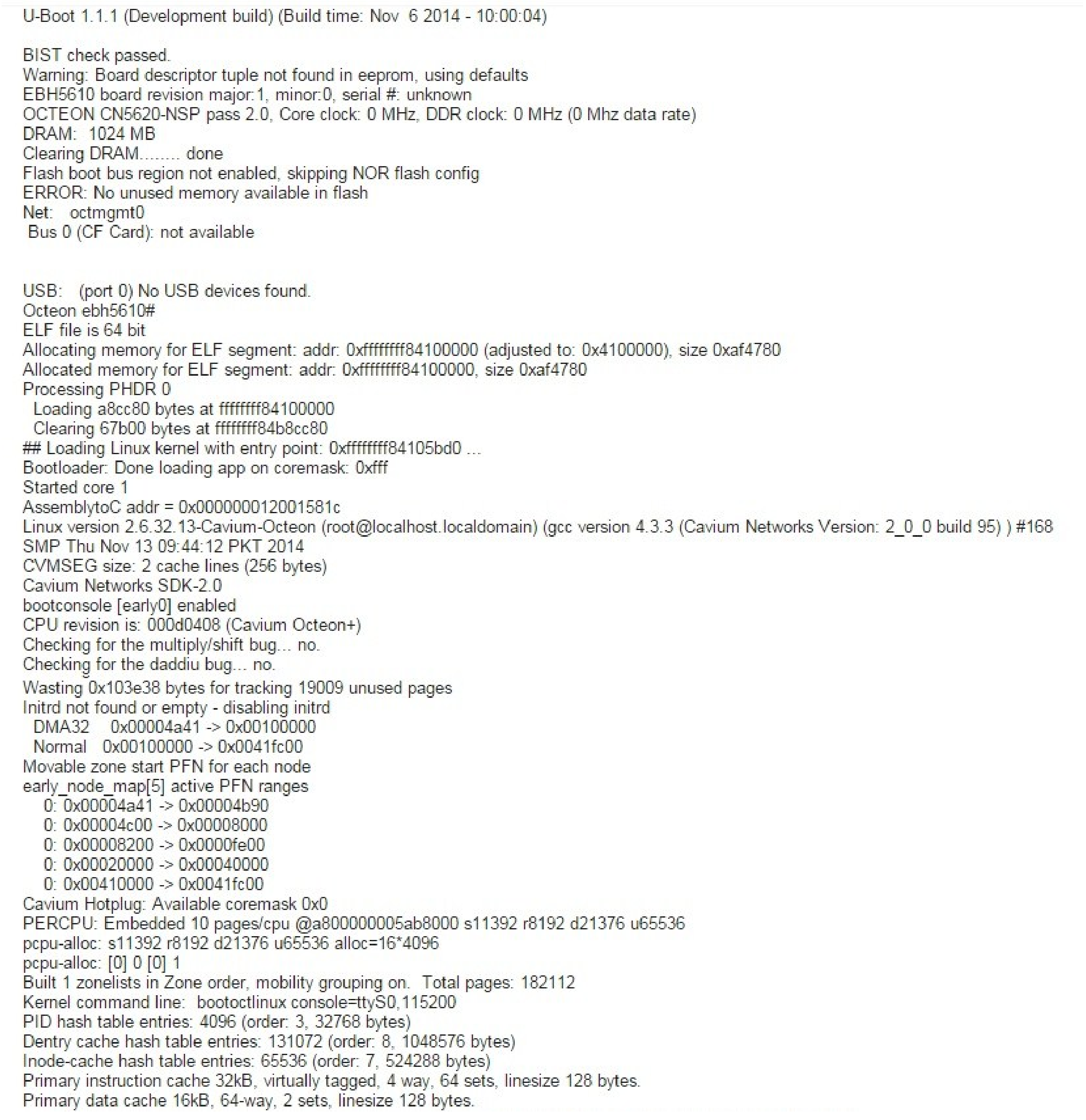
```
    uint64_t mio_uart0_iir=1, mio_uart1_iir=6;
    uint64_t cause[12];
    CIU ciu;
    int iid0= 0, iid1=0;;

    ciu.INT_EN0[0].set_UART0_EN(1); //for testing
    ciu.INT_EN0[1].set_UART1_EN(1); //for testing
//    while(1){

    for (int i=0;i<24;i=i+2){
        cout<<"Core "<<(int)(i/2)<<"\n";
```

```
Core 0
cause before 7f2b5b78dac8
int_sum0 800000000
int_en0 400000000
int_sum4 800000000
int_en4_0 0
cause after 7f2b5b78cac8
Core 1
cause before 7f2b5b78d770
int_sum0 800000000
int_en0 0
int_sum4 800000000
int_en4_0 0
cause after 7f2b5b78c370
```

FIGURE 28: Output of CIU. No pending interrupt on core 0

8.5 OUTPUT OF HYPERVISOR CONSOLE

During execution, if UART's memory mapped registers are written by the core, UART display it in on the console. To validate virtual execution of binaries, hypervisor console output (e.g. shown in Figure 29) was compared with that of real host system console. (For complete log please look at the VM booting log file attached)



```
U-Boot 1.1.1 (Development build) (Build time: Nov  6 2014 - 10:00:04)

BIST check passed.
Warning: Board descriptor tuple not found in eeprom, using defaults
EBH5610 board revision major:1, minor:0, serial #: unknown
OCTEON CN5620-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
ERROR: No unused memory available in flash
Net:   octmgmt0
 Bus 0 (CF Card): not available


USB:   (port 0) No USB devices found.
Octeon ebh5610#
ELF file is 64 bit
Allocating memory for ELF segment: addr: 0xffffffff84100000 (adjusted to: 0x4100000), size 0xaf4780
Allocated memory for ELF segment: addr: 0xffffffff84100000, size 0xaf4780
Processing PHDR 0
  Loading a8cc80 bytes at ffffffff84100000
  Clearing 67b00 bytes at ffffffff84b8cc80
## Loading Linux kernel with entry point: 0xffffffff84105bd0 ...
Bootloader: Done loading app on coremask: 0xfff
Started core 1
AssemblytoC addr = 0x000000012001581c
Linux version 2.6.32.13-Cavium-Octeon (root@localhost.localdomain) (gcc version 4.3.3 (Cavium Networks Version: 2_0_0 build 95) ) #168
SMP Thu Nov 13 09:44:12 PKT 2014
CVMSEG size: 2 cache lines (256 bytes)
Cavium Networks SDK-2.0
bootconsole [early0] enabled
CPU revision is: 000d0408 (Cavium Octeon+)
Checking for the multiply/shift bug... no.
Checking for the daddiu bug... no.
Wasting 0x103e38 bytes for tracking 19009 unused pages
Initrd not found or empty - disabling initrd
  DMA32    0x00004a41 -> 0x00100000
  Normal   0x00100000 -> 0x0041fc00
Movable zone start PFN for each node
early_node_map[5] active PFN ranges
    0: 0x00004a41 -> 0x00004b90
    0: 0x00004c00 -> 0x00008000
    0: 0x00008200 -> 0x0000fe00
    0: 0x00020000 -> 0x00040000
    0: 0x00410000 -> 0x0041fc00
Cavium Hotplug: Available coremask 0x0
PERCPU: Embedded 10 pages/cpu @a800000005ab8000 s11392 r8192 d21376 u65536
pcpu-alloc: s11392 r8192 d21376 u65536 alloc=16*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 182112
Kernel command line:  bootoctlinux console=ttyS0,115200
PID hash table entries: 4096 (order: 3, 32768 bytes)
Dentry cache hash table entries: 131072 (order: 8, 1048576 bytes)
Inode-cache hash table entries: 65536 (order: 7, 524288 bytes)
Primary instruction cache 32kB, virtually tagged, 4 way, 64 sets, linesize 128 bytes.
Primary data cache 16kB, 64-way, 2 sets, linesize 128 bytes.
```

FIGURE 29: Booting log of hypervisor

9 PERFORMANCE OPTIMIZATION

Virtualization solutions are notorious for performance bottlenecks. To optimize performance of hypervisor and guest code, it is necessary to find these bottlenecks to tune code for performance improvement.

9.1 PERFORMANCE TUNING

First step in performance tuning is to identify the most time consuming functions of hypervisor code that are called during execution of guest code. We collected running time of all called functions to identify the hot spots in code. Each function is optionally instrumented to introduce time keeping code at the start and end of each function code. It gives us total time consumed by the function. Total time of a function also includes the time consumed by the functions called by this function. Net (or self) time is then calculated by subtracting the total time consumed by all callees, from the total time of caller function. Another important performance metric is the call count of a function i.e. how many times a function is called. Sample output of sorted flat profile of hypervisor is shown in **Error! Reference source not found.**. The data shows that address ranslation is taking much more time as compared to other functions and it should be optimized.

TABLE 1: Sorted Flat Profile Before Optimization (Showing More Time Consuming Functions Only)

Count	Function Name	Net Time (sec)	Total Time (sec)
49107203	MMUTranslator::GVAtoGPA	2326.3283	4572.8963
29513958	BlockExecController::fetchnPlaceBlock	2283.8921	3057.8145
50000001	BlockExecController::handleRequest	1836.5265	5972.9834
49107203	GPAtoHVATranslator::GPA_to_HVA	1221.9575	1511.028
49107203	MMUTranslator::verify_priviliges	1063.2158	1378.4329
49107203	GVAtoHVATranslator::GVAtoHVA	852.3648	5487.4371
47218428	MMUTranslator::look_staic_translation_32bit	789.6234	896.4091

Software Cache Implementation: To optimize address translation, we implemented a translation cache. Once we translate an address, we place it's translation in translation cache and when next time translation for that address is required we don't need to repeat all steps to get translation. We can directly convert Guest Virtual Address (GVA) to Host Virtual Address (HVA) using this cache. Whenever we need translation for address, we call GVAtoHVATranslator::GVAtoHVA. First it checks whether translation is present in cache. If present, it will directly get translation from cache otherwise GVA is converted to Guest Physical Address (GPA) and then GPA is converted to HVA.

After this optimization, we generated sorted flat profile again. It shows significant performance improvement in MMUTranslator::GVAtoGPA, as shown in **Error! Reference ource not found.**. Due to this optimization, we do not need GVA-to-GPA and GPA-to-HVA translation frequently and lead to reduction in call count. Reduction in call count of MMUTranslator::GVAtoGPA significantly reduces its total time.

In-Place Block Execution: Although using cache has shown its advantage but it was not quite enough. Changing some code level implementations and applying In-place block execution had proved more beneficial. Given below is current improved timing profile of the hypervisor. As we have eliminated the block copying step, fetchnplaceBlock has reduced the time dramatically.

TABLE 2: Sorted Flat Profile after Software Cache Implementation

Count	Function Name	Net Time (sec)	Total Time (sec)
29513958	BlockExecController::fetchnPlaceBlock	2283.8921	2780.8145
50000001	BlockExecController::handleRequest	1804.6144	4351.9108
49107203	GVAtoHVATranslator::GVAtoHVA	742.5720	1468.1078
49106050	GVAtoHVATranslator::Check_cache	449.7157	449.7157
35715	MMUTranslator::GVAtoGPA	1.306426	2.650513
35716	GPAtoHVATranslator::translate_GPA_to_HVA	0.270219	0.270219
26628	MMUTranslator::look_staic_translation_32bit	0.178782	0.310867

TABLE 3: Sorted Flat Profile After In-Place Block Execution Implementation

Count	Function Name	Net Time (sec)
215982611	Processor_IPE::handleRequest()	14.03
124936958	Processor_IPE::fetchNextBlock()	12.65
124936958	TranslatedBlockCache_IPE::doesExists()	9.77
39934396	TransInsAllocator<unsigned int>::construct()	7.64
86515038	Processor_IPE::handleLDST()	6.82
215982610	Processor_IPE::C2Assembly2C()	6.77
87261169	Processor_IPE::incrementTrapCount()	6.22

9.2 PREVIOUS PERFORMANCE IMPROVEMENTS

For improving performance we have made some changes on code implementation level and also in the hardware used. The improvements are mentioned in detail below.

9.2.1 CODE STRUCTURAL ENHANCEMENT

Previously, we were using some C++ standard STL containers for performing many operations. But they were time consuming. A considerable numbers of calls to these STL containers were deteriorating our performance. Using non-standard implementation has improved the overall timing of system.

IN-Place Execution: Initially, we were using C++ standard vector to store translated block (host executable instructions). Before execution we have to relocate this translated block onto a predefined memory place. This memory to memory copying takes some time, which increases the execution time almost double. To avoid this memory-to-memory unnecessary coping, we need to execute the translated block in vector. But we cannot do so because of unavailability of execution rights on vector. Alternatively, we implemented an allocator for vector so that execution rights of vector can be changed and block can be executed directly without copying.

Using Non-Standard C++ Hashmap: We were using C++ standard hashmap to store the TCBs (Translated Cache Block) and GVAtoHVA (Guest-Virtual-Address to Host-Virtual-Address) translated Addresses. A TCB is a set of host executable instructions got after translation of guest executable instructions. It is stored into a key-value pair hashmap. The "key" to search into hashmap is a 64 bit virtual address, which is a starting address of TCB in terms of guest virtual address and "value" against that key is a TCB. This hashmap is used for reusability of TCBs. whenever a translated cached block is required again for execution, it can be directly used after obtaining from hashmap container if the corresponding "key" of TCB is matched. Thus, there is no need to re-fetch and retranslate the required guest's instruction block again. Theoretically, by doing so there should be an improvement in performance but in practice it significantly reduced the timing efficiency. Most of the time is wasted during searching the required TCB in the hashmap container. To resolve this problem, we used an array and implemented an efficient hashing function to find the index of required TCB quickly into the hashmap container. This made the cached block searching timing efficient. Similar mechanism is applied for GVAtoHVA address translation.

Custom Vector for Translated Block: Previously, C++ standard vector was used. Standard vector uses STL containers and insertion and retrieval for entries in it are very costly due to the fact that standard classes do lot of book-keeping for operations and these book-keepings are of no use for us. To avoid this, we implemented a container to hold translated instructions. It is in fact a memory mapped block of 4 Kilobytes with some insertion and retrieval operations. If translated instructions become greater than 4 kilobytes we double the size of block and so on. This reduced the implementation time by 3 minutes.

Macros: For encoding different type of translated instructions, each type of instruction class creates an object. This object calls its own encode function to make an instruction. After encoding translated instruction the object is destroyed, as it is a local object. This object creation and destruction is performed for each instructions translation and consumes time. These encoding functions were replaced with macros, performing the same functionality. So object creation was avoided by macros. This has reduced the execution time by roughly 15 min.

9.2.2 HARDWARE PLATFORM

A significant improvement in time was observed when evaluation board was changed.

The older evaluation board has the following specifications:

TABLE 4: Specification of Old Evaluation Board

Number of Processors	16
RAM	2GB
TLB_entries	32
CPU Model	Cavium Octeon II V0.3

The new evaluation board has the following specifications:

TABLE 5: Specification of New Evaluation Board

Number of Processors	12
RAM	8GB
TLB_entries	128
CPU Model	Cavium Octeon II V0.10

9.3 CURRENT PERFORMANCE OPTIMIZATIONS

We have implemented few techniques and their effects are briefly explained below.

9.3.1 MODIFICATION IN CACHE REPLACEMENT POLICES

Caches have played a very important role in previous optimization effort. We have currently two caches. One for storing translated blocks, this avoids the block translation of the same block. Other is for storing translated addresses, this avoid the address translation procedure if the translation is found in the cache.

Any particular replacement policy weren't implemented before. Replacement of the blocks was random. If the replacement is done carefully so that the more translated blocks are found in cache. This will give us performance improvement.

We have implemented replace-least-used strategy to see whether if this gives us any improvement or not. In Replace-least-used, the blocks which is used the least is replaced with new block which is currently translated. For implementing this, a "count" variable should be incremented every time a block is obtained and used from the cache. This will tell us how frequently a block is in use. The higher the number, that block is more likely to be used. The block with the least frequency number should be replaced when a new translated block is to be placed in the cache. This adds to operations one is just addition but that would be done all the time when cache is checked (i.e. it would be millions of times during booting) and second is searching operation, which will find the block with least frequency number.

Although these operations doesn't seem time consuming at first sight but they will be called millions of times and also in serial with the guest code execution. This incurs more time cost than it will give us benefit from maintaining the cache more efficiently. Also searching becomes more time consuming as we increase the size of a 2D cache for storing more blocks. These added operations should at least give that much optimization to cancel out its own cost. But these operations were more time consuming and actually degraded the timing performance.

9.3.2 MODIFICATION IN CACHE SIZE AND HASH FUNCTION

Previously, the hash function that was used in placing cache block was modulus, which is actually division. "Objdump" shows that it was generating 15 instructions for that operation. This hash function is replaced with simple bit-shifting operation, which only generates couple of instructions. As this operation is performed millions of times, changing some instruction shows improvement. Also now the size of cache was increased in power of 2. These changes proved to be effective in optimization.

9.3.3 PARALLELIZATION OF BLOCK TRANSLATION

When guest code is running most of time is consumed in block translation. Normally the block translation is done in serial with the guest code flow. A block is fetched, translated and executed. Until the block is executed the next block is not translated.

The main idea was to parallelize the translation in a separate thread and give the guest code a 100% hit rate in cache. When a translated block is running, the parallel thread translates the two possible blocks that can be taken next and put in cache. When the execution of current block is finished it will check for the next block in cache which will be already placed in the cache by parallel thread.

There were two approaches to test. (i) To create a thread every time a block is translated. This new thread will start to translate the next two possible blocks in the case of “branch” instruction and one in the case of “jump” instruction. (ii) To create a thread at the initializing time of processor and wake it up only when there is work to be done. First one proved to way too much time consuming. Just creating a thread millions of time, consumes more time than to provide benefit from its translation. Second one cancels the creation time but synchronization problems arises.

The main problem arises due to shared structures between the two threads. The block translation cache and some other variables are required by both threads at different times. Resolving these synchronization issues using mutex locks proves to be inefficient. Other problem was the initial processor design. Processor’s structure is designed without considering parallelization. So there are variables which shouldn’t be updated in the translation parallel thread but they get updated. A lot bigger structural changes would be required to rearrange the variables to reduce the dependencies among the two threads.

This strategy also didn’t proved to be quite effective due mutex lock implementation for resolving synchronization issues.

9.3.4 REDUCTION IN CONTEXT SWITCHING

Whenever hypervisor’s intervention is required, a context switch happens which takes the program’s flow from executing assembly level guest code to hypervisor’s C/C++ code (e.g. when an address translation is required in load/store instruction, assembly to C context switch happens). Also after the block translation, a context switch happens from C/C++ to assembly guest code. These context switches happens number of times even in a single block and puts a cost on performance.

These context switches happens for a number of reasons (e.g. address translation, exceptions, systems calls and fetching next blocks etc.) One of the reasons was when “count/compare” register is written by guest, it should be updated by the timer mechanism which requires context switch in C/C++.

The particular context switch happens when count register is written, can be replaced by some assembly instructions. This makes more continuous guest code and reduces the context switches. As the guest writes count at every time it registers a timer interrupt with the hardware. Replacing this context switch proves to be quite effective.

9.3.5 BLOCK LINKING

Linkage of blocks intuitively suggests that guest code could follow the intended execution path of code with less hypervisor's intervention and hence improving the performance.

For implementing block linking, hypervisor should have knowledge about where the guest will decide to jump. Only "j" and "jal" assembly instructions are deterministic (i.e. we can know where the guest will jump, as the address is embedded in the instruction itself). All the rest of control instructions depend on the non-deterministic data in GP registers of guest (e.g "jr" depends on the GP register in which the address would be stored in runtime by guest). Hypervisor can't predict what contents will be in GP register at the end of translated block execution. Approximately 99% of control instructions that came up during booting are non-deterministic in nature. So, even if block linking is implemented at those points this doesn't give any real improvement.

9.4 CORRECTNESS RELATED BUG FIXES

- **Implementation of atomic instructions:** Some of the instructions were not translated correctly before. Instruction like LL and SC implements atomic load and store operations. They require a lock for its correct implementation. MADD (multiple and add) instruction requires the loading of host's special lo and hi registers before its execution.
- **Flushing data cache:** Hypervisor was showing unexpected behavior by suddenly crashing the program randomly during its execution. But if we add a system call after specific intervals, hypervisor shows the correct execution of the system. This illogical behavior was due to the fact that we were using data memory as an instruction memory in hypervisor code. When data (guest code) is used as an executable instruction then data and instruction caches interleaves with each other, producing illogical results. We resolved this issue by flushing the data cache in hypervisor code whenever we needed to execute a new guest block.
- **Page boundary detection in block fetching:** In user mode, program may or may not be loaded contiguously. For example if virtual page number 0x120001 is mapped to physical

frame number 0x41A36C then virtual page number 0x120002 may or may not be present at physical frame number 0x41A36D. Our previous implementation of block fetching mechanism was on the assumption that memory is contiguous i.e. virtual page number 0x120002 will be present at physical frame number 0x41A36D. But normally this is not the case and this causes fetching of illegal instructions. To avoid this, we implemented a check for page boundary. If block is spanned over boundary of a page then we fetch only part of block which is present in first page and we process remaining part in next fetch cycle. This resolved the unusual kernel crashes.

10 EVALUATION BEFORE AND AFTER OPTIMIZATION

The optimization strategies that have been applied improved the overall timing performance of hypervisor. To evaluate the performance of hypervisor we have calculated different parameters along with and without optimization related changes. The tests have also been performed on the native system.

Native MIPS64 system is Cavium OCTEON CN5700. It was used for both debugging and experimenting purposes. It has 12 cores, 800MHz clock rate, and 2MB L2 Cache. There are 4 slots for 1G RAM each. Type-2 hypervisor emulates the hardware of this board. The comparison is performed between the execution of hypervisor emulating OCTEON CN5700 (with and without optimization) and actual native system i.e. real board OCTEON CN5700.

10.1 BOOTING TIME

The first and foremost comparison is the amount of time taken during the successful booting of system. Time consumed to completely boot the linux system over native vs virtual machine is shown in the Figure 30. These values are mean of the multiple readings. The time taken by the native system is just 47 sec. Before optimization hypervisor takes 443 sec (which is 7.5 min roughly) and after optimization is 75 sec (i.e. 1 min 15 sec).

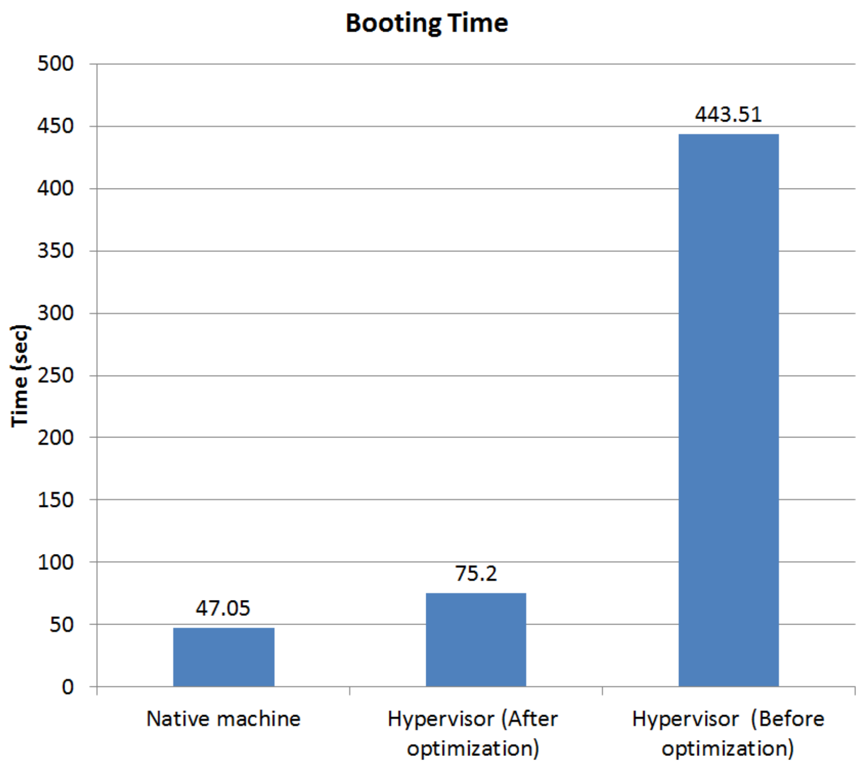


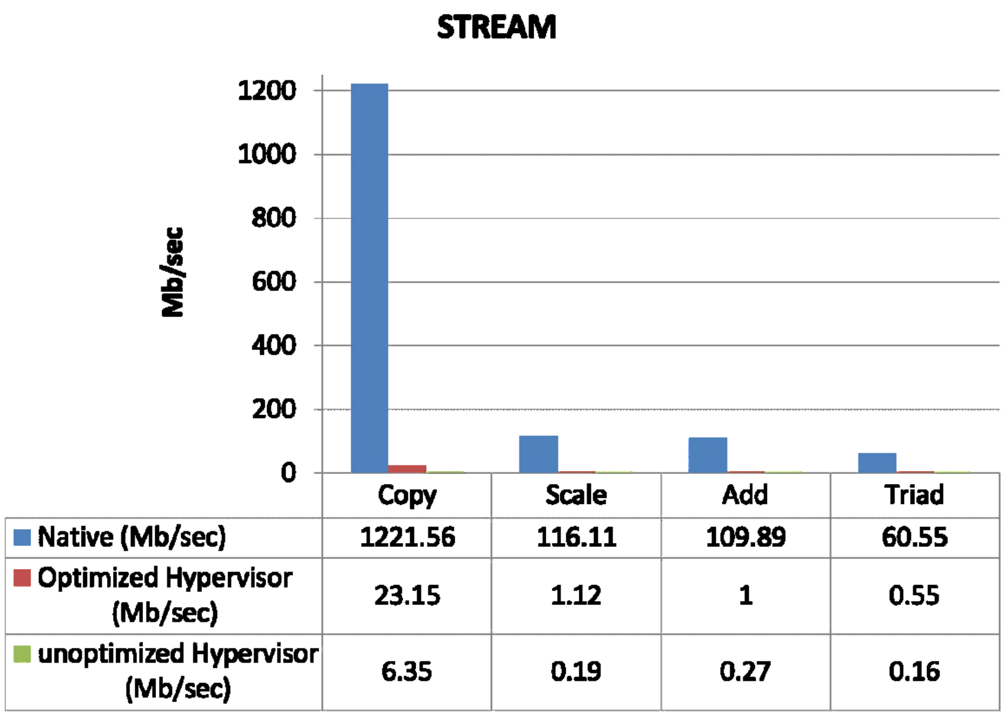
FIGURE 30: Comparison of Booting Time

10.2 STREAM

We also perform STREAM benchmark, which performs these 4 basic memory operations. It measure transfer rates and latencies. “Copy” simply copy one element from memory to another. “Scale” multiples a value from memory and save the result back to memory. “Sum” is performing sum operation and saving it back to memory. “Triad” performs the scaling and summation operation.

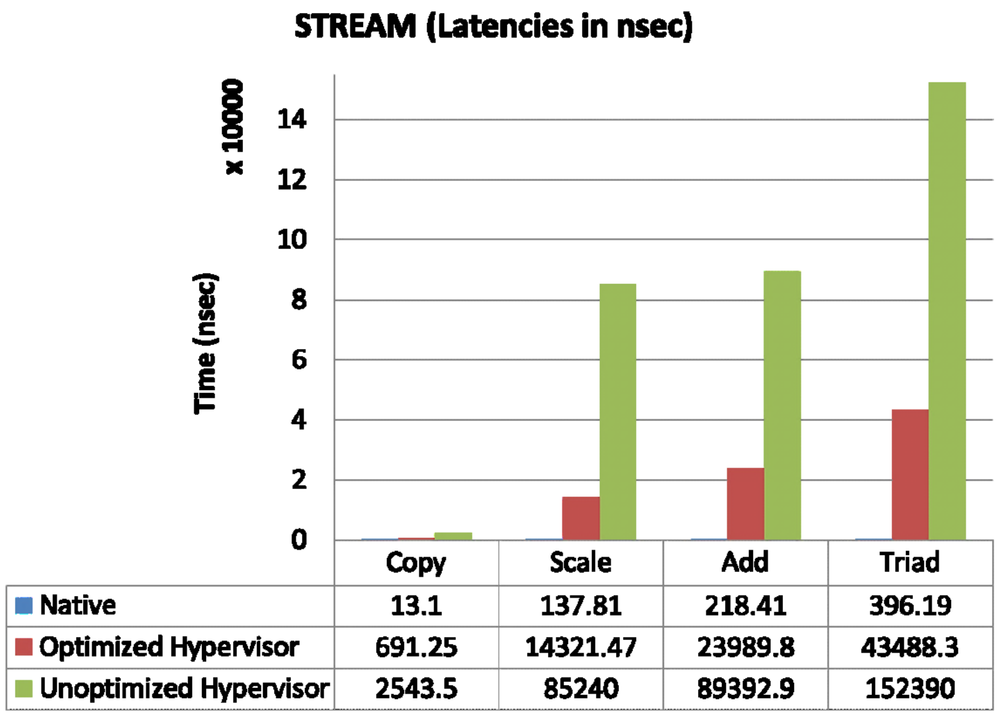
- copy a(i) = b(i)
- Scale a(i) = q*b(i)
- Sum a(i) = b(i) + c(i)
- Triad a(i) = b(i) + q*c(i)

The Figure 31 shows the transfer rates for all above operations on native system, as well as hypervisor with and without optimization. The Figure 32 shows the latencies experienced during these operations.



	Copy	Scale	Add	Triad
🟦 Native (Mb/sec)	1221.56	116.11	109.89	60.55
🟥 Optimized Hypervisor (Mb/sec)	23.15	1.12	1	0.55
🟩 unoptimized Hypervisor (Mb/sec)	6.35	0.19	0.27	0.16

FIGURE 31: Transfer Rate for Stream Operations



	Copy	Scale	Add	Triad
🟦 Native	13.1	137.81	218.41	396.19
🟥 Optimized Hypervisor	691.25	14321.47	23989.8	43488.3
🟩 Unoptimized Hypervisor	2543.5	85240	89392.9	152390

FIGURE 32: Latencies for Stream Operations

10.3 LMBENCH

Memory bandwidth or data transfer rate is fundamental to evaluating a system. lmbench has been used in profiling the system's memory bandwidth for different memory operations.

Table 6 shows the memory bandwidth for different memory operations. The readings were taken for 2MB memory size. "rd", "frd", "fwr" and "wr" shows data transfer rate by processor for reading and writing 2MB contents. "rdwr" measures the time to read data into memory and then write data to the same memory location and shows transfer rate for 2MB memory size. "cp" and "fcp" shows data coping rate. "bzero" and "bcopy" measures how fast the system can bzero and bcopy memory respectively. The results are shown for both hypervisor (with and without optimization) and native hardware.

TABLE 6: Lmbench Output for Different Memory Operations

Operations	Native System (Mb/sec)	Optimized Hypervisor (Mb/sec)	Un-Optimized Hypervisor (Mb/sec)
rd	1870	126	17.8
wr	8190	134	20
rdwr	1544	50	6.8
cp	749	65	9.1
fwr	3055	40	4.9
frd	840	38	4.77
fcp	520	20	2.45
bzero	3123	24	6.48
bcopy	673	20	4.05

The table 7 shows the transfer rate of mmap 2MB file for both native and hypervisor. The "open2close" includes the I/O operations (e.g opening /closing a file) involved in file mmap, while "mmap_only" excludes the I/O operations. Table 8 shows the latencies (in millisecond) in native system and hypervisor, while performing the file mmap. Table 9 and 10 shows the bandwidth and latencies for 2Mb file reading respectively.

TABLE 7: Lmbench Bandwidth Results for File Mmap

Memory bandwidth for file mmap	Native System (Mb/sec)	Optimized Hypervisor (Mb/sec)	Un-optimized Hypervisor (Mb/sec)
open2close	856	9.5	2.3
mmap_only	1454	68	8.8

TABLE 8: Lmbench Latency Results for File Mmap

Latencies for file mmap	Native System (msec)	Optimized Hypervisor (msec)	Un-optimized Hypervisor (msec)
open2close	1.2	100.5	450
mmap_only	0.7	15	113

TABLE 9: Lmbench Bandwidth Results for File Reading

Memory bandwidth for file reading	Native System (Mb/sec)	Optimized Hypervisor (Mb/sec)	Un-optimized Hypervisor (Mb/sec)
open2close	810	13	2.36
io_only	830	13.3	2.46

TABLE 10: Lmbench Latency Results for File Reading

Latencies for file reading	Native System (msec)	Optimized Hypervisor (msec)	Un-optimized Hypervisor (msec)
open2close	2.5	153	840
io_only	2.5	151	810

Table 11 shows the data transfer rates for two operations. “bw_unix” measure how fast the parent process can read the data in size-byte chunks from the pipe. “bw_pipe” measures data transfer rate between two processes through pipe.

TABLE 11: Lmbench Bandwidth Results for Pipe And Unix

	Native System (Mb/sec)	Optimized Hypervisor (Mb/sec)	Un-optimized Hypervisor (Mb/sec)
bw_pipe	850	2.75	0.74
bw_unix	1450	3.7	0.86

The table 12 below shows the network throug h put of hypervisor and native system. It a client-server test program with a message size of 65536 bytes.

TABLE 12: Lmbench Results of Network Through-Put

	Native System (Mb/sec)	Optimized Hypervisor (Mb/sec)
bw_tcp	11.57	2.01

10.4 Linux Testing Project (LTP)

For functional testing of Type-II hypervisor we executed LTP's syscall tests. This test is executed on bare metal (Cavium OCTEON CN5700) and also on hypervisor for comparison. Total 954 system calls tests were performed on the native system. 253 out of 954 tests were skipped or failed due to configuration issues for MIPS or the some commands are not supported by the native system. 701 tests were successful on native system. These successful tests were performed on the hypervisor to test the behavior of virtual system. All of the tests passed on the native system were also passed on hypervisor.

11 Impact on Project Progress

In 10th deliverable, we have integrated new networking infrastructure in the hypervisor. The previous infrastructure for networking was complex in implementation and also inefficient. The new infrastructure makes use of para-virtualization. A virtio-net device is created in guest and vhost-net which is kernel module on host is used. To connect the two, a network device is implemented in hypervisor. In the extension time period, optimization effort has improved the previous performance of hypervisor.

Now type-II hypervisor is a complete prototype hypervisor, which can be further developed to run on any MIPS vendor.

References

[1] Cavium Networks OCTEON Plus CN54/5/6/7XX, Hardware Reference Manual CN54/5/6/7XX-HM-2.4E, January 2009, chapter 14 (CIU).

~*~*~*~