December

2015

Final Project Report of Type-2 Hypervisor for MIPS64 Based Systems

This document is the final project report of "Development of Type-2 Hypervisor for MIPS64 Based Systems" project, funded by National ICT R&D Fund Pakistan. The report starts with the introduction of overall project. Then it describes the infrastructure of hypervisor and details of different basic units. The report ends with the performance comparison of hypervisor as compared to bare-metal.

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

Contents

1. Introduction of Project 1

2. Type-2 Hypervisor for MIPS64 2

3. Basic Design Infrastructure 3

3.1 Execution-Flow of Type-2 Hypervisor 4

4. System Modules 5

4.1 Multi-Core Processor Virtualization 5

4.2 ISA Virtualization (using Dynamic Binary Translation) 9

4.2.1 Privileged Instructions 9

4.2.2 Unprivileged Instructions 10

4.2.3 Cavium Specific Instructions 13

4.2.4 Branch and Jump Instructions 13

4.2.5 Control Shifting Instructions 16

4.2.6 Special Instructions 16

4.3 Memory Management Unit 16

4.3.1 GVA to GPA Translation 17

4.3.2 GPA to HVA Translation 18

4.3.3 Page Table 18

4.3.4 Translation Look-aside Buffer (TLB) 18

4.3.5 Cavium Segment Implementation 18

4.4 Timer Unit 19

4.5 Interrupt and Exception Handling 20

4.5.1 SIGFPE: Floating point exception handling 21

4.5.2 SYSCALL: System call handling 22

4.5.3 TLB and Address error Exception Handling 22

4.5.4 External Interrupts 22

4.6 I/O Devices 22

4.6.1 UART 23

4.6.2 Central Interrupt Unit (CIU) 24

4.7 Virtual Disk 26

4.7.1 Virtio Block Configuration 26

4.7.2 Vhost Block Configuration 28
4.7.3 Creation of Virtual Disk for Hypervisor 28
4.8 Networking 30
4.8.1 Virtio-Net 31
4.8.2 Vhost-Net 32
4.8.3 Network device in hypervisor 33
4.8.4 Execution flow of networking 33
5. Performance Evaluation 35
5.1 Booting Time 35
5.2 lmbench 36
5.3 STREAM 39
5.4 Linux Testing Project (LTP) 39

Figures

Figure 1: Overview of Type-2 Hypervisor as a user Process …... 2

Figure 2: Multithreaded Design of Type-2 Hypervisor ……………………………………… 3

Figure 3: Basic Execution Cycle of Processor ……………………………………………... 4

Figure 4: Execution Flow of Processor …………………………………………………….. 6

Figure 5: Multithreaded view of hypervisor and External Devices ……………………....... 7

Figure 6: Execution Flow of Hypervisor with SMP …………………………………..… 8

Figure 7: Memory Mapping of MIPS system ……………………………………………….. 16

Figure 8: Address Translation of Hypervisor ……………………………………………….. 17

Figure 9: Timer infrastructure ……………………………………………………….…...... 19

Figure 10: Exception Handling in User mode ………………………………………………. 21

Figure 11: CIU Interrupt Distribution from External Devices to Core ………………………… 25

Figure 12: Memory Mapping between core and External Device ………………………...… 25

Figure 13: Code Snippet from virtio_blk.c ………………………………………………… 27

Figure 14: Commands for Device Creation ….……………………………………………… 27

Figure 15: Networking Infrastructure of Type-2 Hypervisor ………………………………….. 30

Figure 16: Communication between Virtio-Net, Vhost-Net and Network Device ……………. 34

Figure 17: Comparison of Booting Time …………………………………………………… 35

Figure 18: STREAM Results for Hypervisor and Native System ……………………….…….. 39

Figure 19: Latencies for Stream Operations ……………………………………..…………. 40

Tables

Table 1: Lmbench Output for Different Memory Operations 36
Table 2: Lmbench Bandwidth Results for File mmap 37
Table 3: Lmbench Latency Results for File mmap 37
Table 4: Lmbench Bandwidth Results for File Reading 37
Table 5: Lmbench Latency Results for File Reading 38
Table 6: Lmbench Bandwidth Results for pipe and unix 38
Table 7: Lmbench Results of Network Through-put 38

1. Introduction of Project

Virtualization has rapidly became the foundation of concepts like cloud computation and big data centers. It has found applications in various areas like include server consolidation, convenient software development and testing, dynamic load balancing and disaster recovery. Embedded virtualization is a recent phenomenon being in its early stages as compared to sever and PC virtualization. It holds huge potential and offers challenging and interesting prospects for research and development given the size and diversity of the embedded markets from general consumer devices to high-end enterprise networking solutions. Virtualization is gaining ground in the embedded domain with ARM announcing to incorporate the hardware level support for virtualization in its processors. MIPS community will most likely follow the suit as a substantial portion of the embedded market uses MIPS processor architecture.

This project was not only research effort for exploring virtualization techniques in MIPS64 based systems but also can be developed into industrial project e.g. MIPS based high end networking and communication devices can take the advantages of virtualization for isolation, security and optimal hardware utilization purposes. The possible benefits for exploring virtualization include:

a) Providing strict isolation and security between virtual machines (e.g. segregating routing information system and forwarding tables into two virtual machines inside a router so that any malfunctioning of one doesn't bring the other down.)
b) Running older versions of Linux OS and the user-land programs on a new hardware
c) Running different versions of software stack concurrently without any interference with each other
d) Server consolidation by shifting loads from multiple under-utilized systems to one system such that each virtual machine is virtually equivalent to one under-utilized system.

This hypervisor targets MIPS64 based multi-core embedded systems. MIPS processors have wide spread use in embedded devices and MIPS Technologies is the world's second largest processor IP company, providing the leading processor architecture for embedded systems. Its industry-standard architectures and cores power some of the world's most popular products for

the home entertainment, communications, networking and portable multimedia markets. The main objective of this project was development of an open source Type-2 hypervisor, for Linux-based MIPS64 embedded devices. This project was a successful step towards researching and enabling virtualization in MIPS based embedded systems.

2. Type-2 Hypervisor for MIPS64

A hypervisor or virtual machine monitor (VMM) provides a software virtualization environment in which other software, including operating systems, can run with the assumption of full access to the underlying system hardware, but in fact such access is under the complete control of the hypervisor. Multiple VMs may be managed simultaneously by a hypervisor. Hypervisors are generally classified as Type 1 or Type 2, depending on whether the hypervisor is directly running in supervisor mode on the physical hardware (Type 1) or is itself hosted by an operating system (Type 2).

Type-2 means that it is a hosted hypervisor which runs on MIPS64 based Linux systems as a Linux process. This process provides the virtual hardware representation to the guest running above it. Virtual hardware consists of software representations of CPU cores, memory, and peripheral devices. In real hardware, CPU cores and devices work concurrently. Fig. 1 shows an overview of Type-II hypervisor. Similar to other user processes, it is executing in the user space of Linux based host OS. Multiple cores can be initialized. Each core is virtualized through as a separate thread. Also the external devices like UART are also implemented as threads. Networking has also been enabled. The guest will be running its applications over the host OS in complete isolation. Hypervisor doesn't have any knowledge or

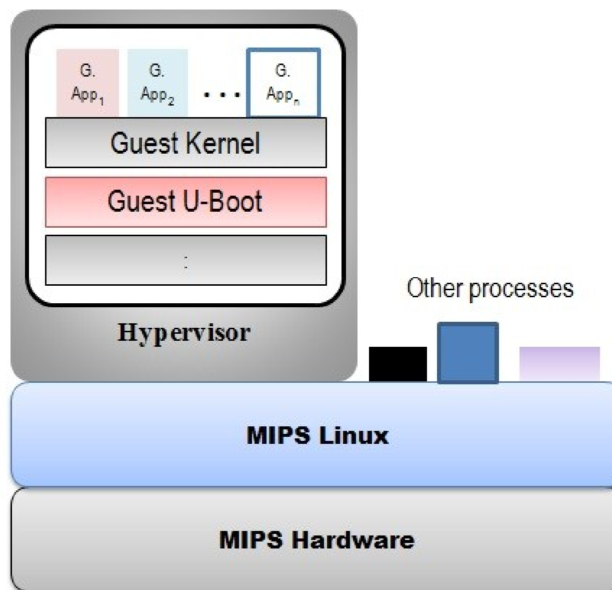


Figure 1: Overview of Type-2 Hypervisor as a User Process

control over guest applications. Hypervisor simply translates a block of guest's instructions into a block of safer instructions that can be executed in user mode.

3. Basic Design Infrastructure

Type-2 hypervisor behaves like a Linux process that could be scheduled by host operating system. However, this process has to present a software representation 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 are implemented as threads in software representation. Multithreaded design for hypervisor, as shown in Fig 2. 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.

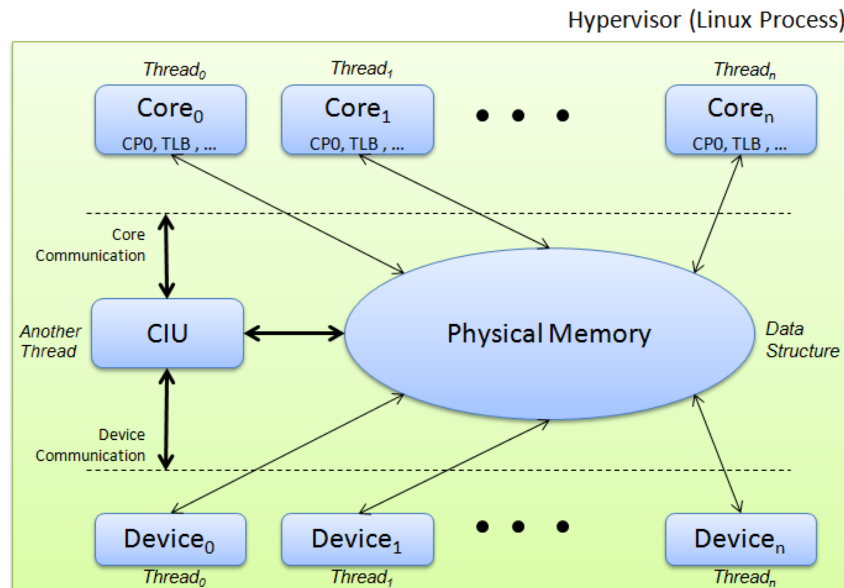


Figure 2: Multithreaded Design of Type-2 Hypervisor

Design of a basic hypervisor requires development of three basic units.

a) ISA Emulation: First and foremost is ISA emulation. For providing virtualization, hypervisor would be able to translate basic assembly instruction code of guest into equivalent host's instruction set.

b) Memory Management: what is the view of physical memory to the guest? How the accessibility is provided for different regions? How the address mapping would be managed by hypervisor? All these questions are accounted in memory management.

c) External Interrupts and Exceptions: For successful implementation of hypervisor, interrupts and exceptions play an important role. Routing of guest interrupts and exceptions are critical in successful working of guest system.

After the successful implementation of core virtual system, external devices play a very important role in making the virtual system usable in real applications. Implementation of external devices like network card, UART and disk are very important.

Implementation decision of these three basic units immensely impacts the overall performance of hypervisor.

3.1 Execution-Flow of Type-2 Hypervisor

After the creation of virtual board, the virtualized processor goes in a simple three step cycle. Fig 3 shows the basic execution cycle of virtual processor.

1. Block Fetching: Basic guest code block is fetched. Block is defined as a consecutive set of instructions ending with a control shifting instruction. Block is fetched from the current value of PC register.
2. Instruction Translation: The fetched guest's assembly instruction block is translated into a set of unprivileged instructions. Each instruction is translated into no. of host assembly instructions. Translation of each instruction is concatenated to form the translated block.
3. Block execution: The translated block is executed directly on hardware. Control may shift back to hypervisor due to following reasons:

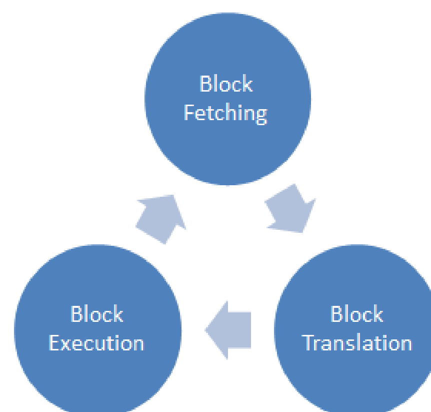


Figure 3: Basic Execution cycle of Processor

a. Address translation for load/store instructions
b. TLB operations
c. System call
d. External interrupt or exception
e. Complete execution of block

The request for particular case is handled and control is shifted back to the last interrupted instruction. In the case of end of block, the next block is fetched from current value of PC, translated and executed.

Fig. 4 shows the detailed and complete execution flow of block level dynamic binary translation. The first block is fetched from the guest virtual reset vector address 0xbfc00000 (Step 1). The fetched block is translated (Step 2,) cached (Step 3) and executed (Step 4 and 5). When execution of the translated block is completed, the control returns to the hypervisor for fetching the next block (Step 6, 7 and 8). The presence of next target block is checked in the cache. The control is simply transferred to block, which has been found into the cache-blocks (Step 13). If the new block is not found among the cache-blocks, it is fetched (Step 9), translated and also cached. The cache-blocks may contain some fixed number of translated blocks in it. When cache is full, one block from the cache-blocks is replaced by the new translated block randomly. During execution of block if control is transferred to handler for any other request than new block, Step 10 is taken. If the request in not handled properly, system will terminate (Step 14). If the request is handled successfully, the execution of the current block is resumed (Step 12).

4. System Modules

The whole infrastructure of hypervisor is consisted of different modules. Each one is explained briefly ahead.

4.1 Multi-Core Processor Virtualization

For each processor, a separate thread is created. Each thread provides the virtualization of a single processor. System can be initialized with multiple cores. All cores will be working in

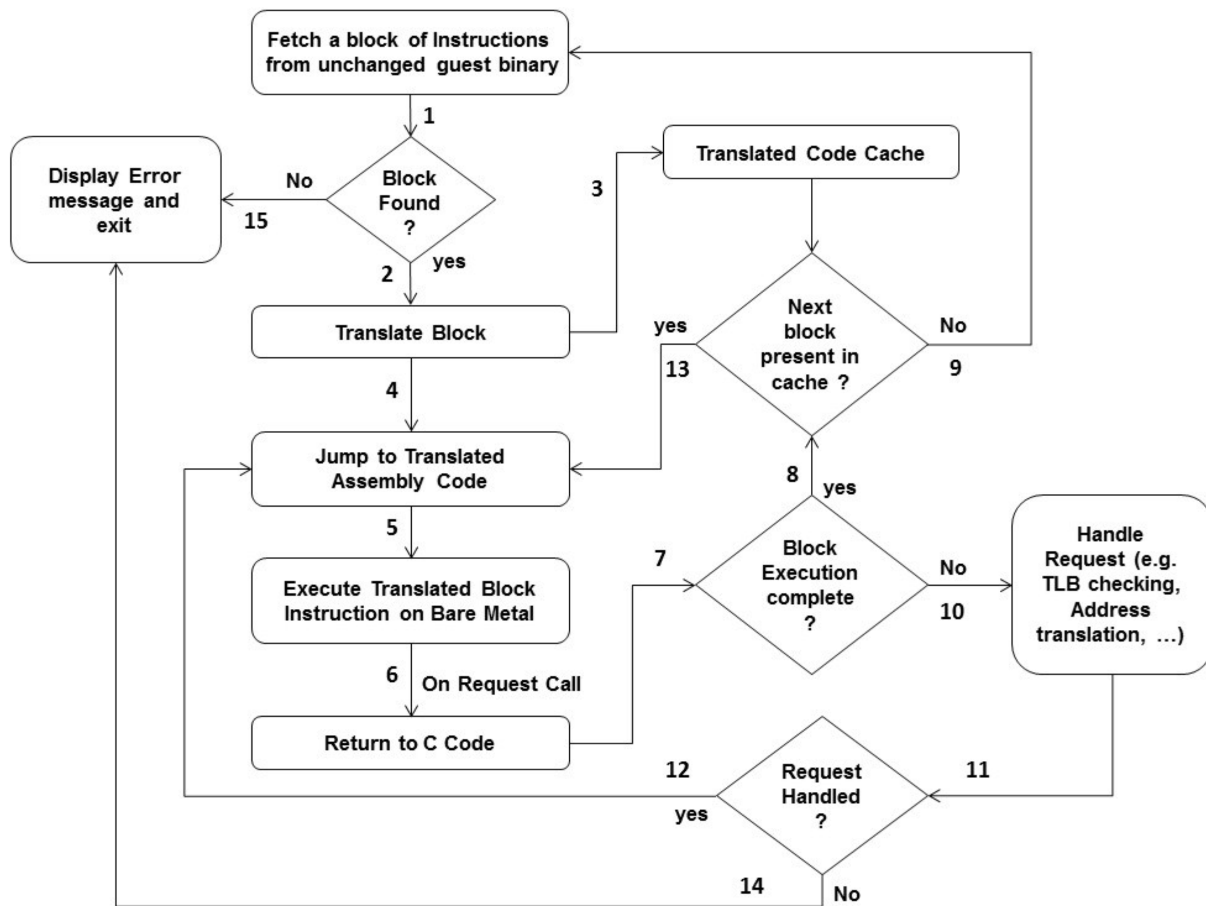


Figure 4: Execution Flow of Processor

parallel. The basic three execution steps are followed by each processor, which are described previously.

Figure 5 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. Figure 6 shows the modified flow chart of hypervisor.

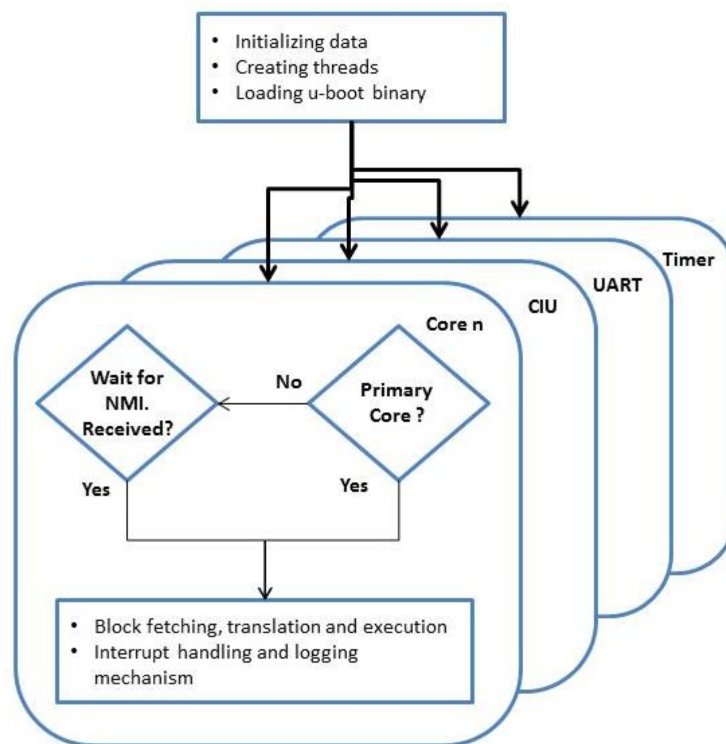


Figure 5: Multithreaded view of Hypervisor and external devices

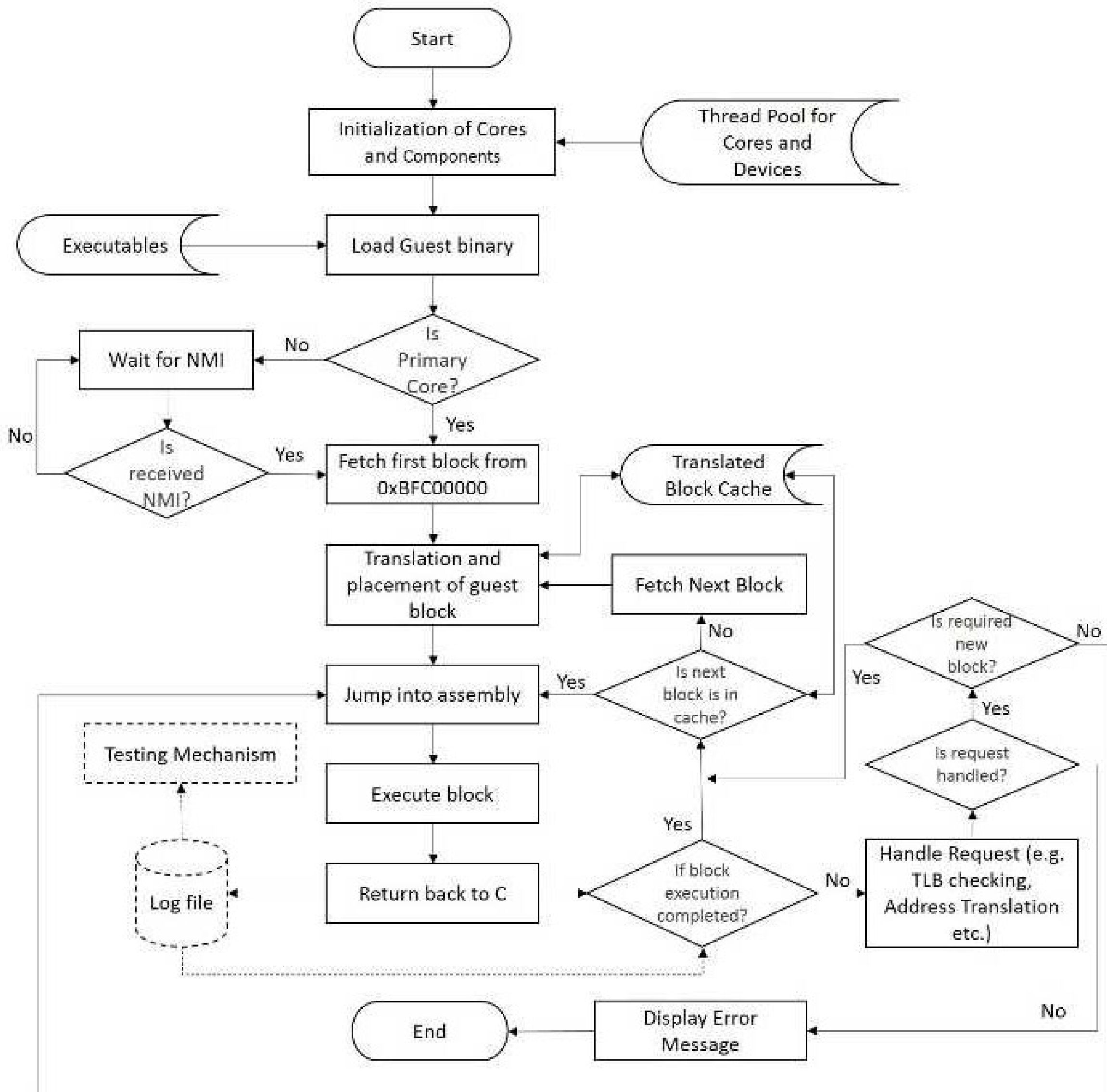


Figure 6: Execution Flow of Hypervisor with SMP

4.2 ISA Virtualization (using Dynamic Binary Translation)

Dynamic binary translation was used to provide ISA virtualization. Block of guest's assembly instructions are translated into host set of instructions which can be executed in user space. The idea is to not completely emulate the instruction but either change the registers embedded in instruction or replace it with other assembly instructions which will perform the equivalent operation 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.

A memory based copy of all registers is present (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, 3 gp registers were used. 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 guest's memory based registers.

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 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/ mtc0 are implemented to move the contents of Co-processor register to and from gp registers, using the memory location of registers. tlbr/ tlbwi/ tlbwr/ tlbp are completely handled in handler written in C/C++ code, assembly instructions just shift the control back to hypervisor

when these instructions occur. di/ei are translated into instructions which change bits in status register to disable and enable interrupts respectively. In “eret” implementation first erl bit of status register is checked to be 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/saad" implementation is different from simple store instruction on the account of fact that it directly accesses the memory location and adds a value and store the resultant at same memory location. This operation is done atomically. 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.

"seqi/snei" instructions 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" is 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 the contents of these registers are moved to hardware registers and then execute multiplication.

mtm0, mtm1, mtm2, mtp0, mtp1 and mtp2 instructions moves the contents of gp register to special purpose register (MPL0, MPL1, MPL2, P0, P1 and 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.
- 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.

4.3 Memory Management Unit

It is the most important unit of a computer system. The memory of typical MIPS system has different regions as shown by Fig 7. Every region differs on the basis of

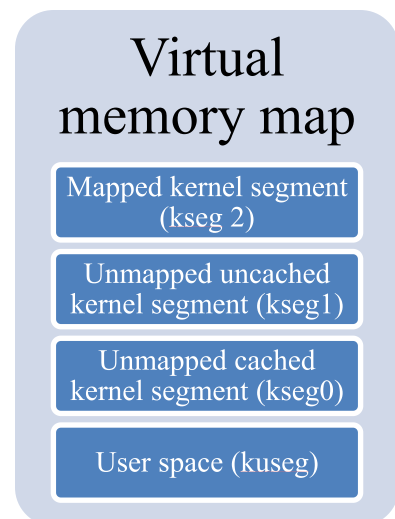


Figure 7: Memory mapping of MIPS system

accessibility and mapping. 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 these rules were implemented 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, same method as used by the hardware was followed. For translation of GPA to HVA, hash map is used to store information of all regions mapped in host virtual address space.

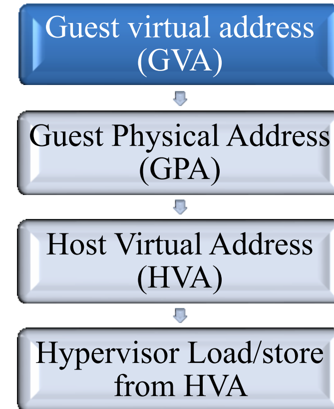


Figure 8: Address Translation of Hypervisor

4.3.1 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 TLB is searched for address translation. For 111, region is kseg3 which is not directly mapped and TLB is searched 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 TLB is checked 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 TLB is searched 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.

4.3.2 GPA to HVA Translation

All physical memory regions of a machine are mapped in virtual address space of hypervisor. Once guest's GVA (Guest virtual address) is translated into valid GPA (Guest Physical Address) that physical address is identified as memory region or I/O device. According to the region it is translated to HVA (Host Virtual Address) using hash map. Once a valid GVA-to-HVA translation is done, the instruction involving the address can be executed with this translated address.

4.3.3 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.

4.3.4 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, hypervisor jumps to the exception handler entry point from where the kernel determine which kind of exception it is and service the exception.

4.3.5 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.

4.4 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. In current implementation 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. Fig 9 shows the timer infrastructure is works serially in each processors and only "on demand".

Whenever the guest needs to get time from hardware it reads the count or CVMcount register. Hypervisor 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

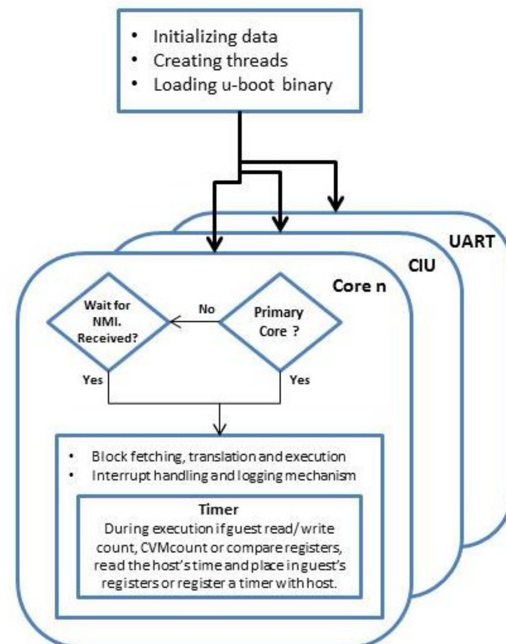


Figure 9: Timer infrastructure

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.

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:

a) An instruction like trap or syscall, itself shifts control to an exception routine. Exceptions like these are called programmed exceptions.
b) 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.

In case of programmed exceptions, exception-causing instructions are replaced with innocuous instructions that explicitly transfer control back to a hypervisor's 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 is caught to handle the exception. Once the control is available in hypervisor, exception handling routine could be called to do the rest.

In current 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, the status register is checked and then accordingly set pc back to the address from where exception has actually occurred. Figure 10 shows the overall.

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.

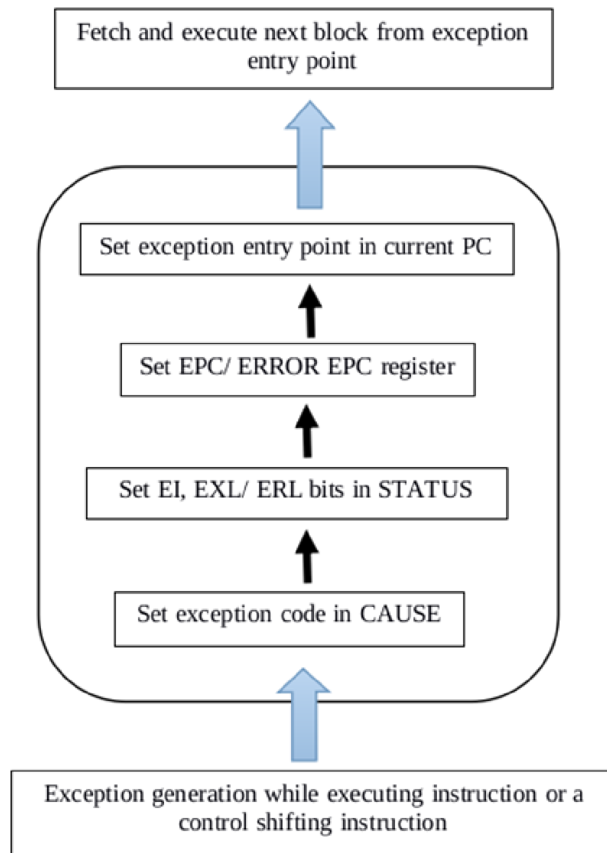


Figure 10: Exception handling in user mode

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. Hypervisor provide a handler to catch this signal. When control comes to this handler, hypervisor 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 occur 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.

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 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 External Interrupts

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.

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.

4.6 I/O Devices

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 two IO devices are implemented

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

4.6.1 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.

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, hypervisor have separate threads called Receiver Thread and Transmitter thread.

- **Receiver Thread:** The purpose of receiver thread is to handover data to processor which the user 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.

4.6.2 Central Interrupt Unit (CIU)

CIU is a Cavium specific unit and 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. Interaction of CIU, external devices and cores is shown in Figure 11. 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, whereas interrupt identification/handling is done in software.

A simplest abstraction of CIU has been implemented. 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.

In integrated code, shared memory regions are defined for CIU to work with other components of virtual board. Figure 12 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

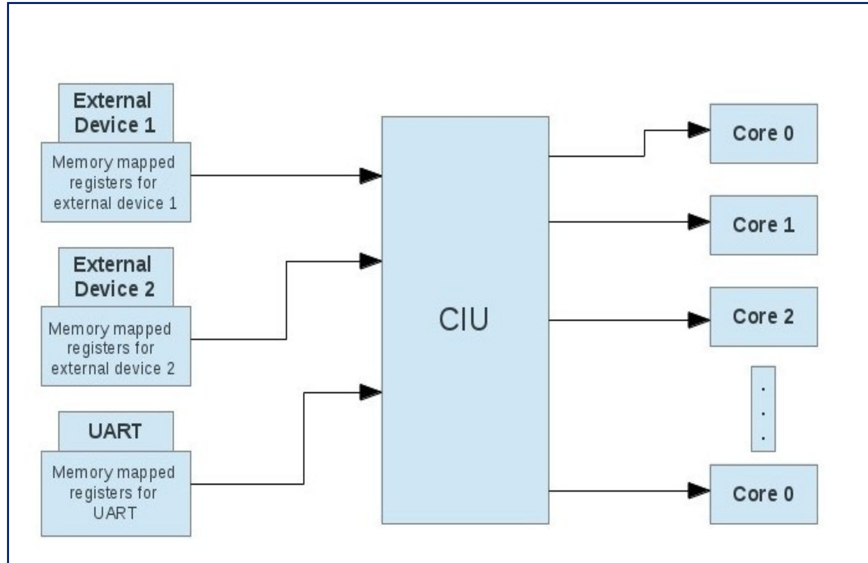


Figure 11: CIU interrupt distribution from external devices to core

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.

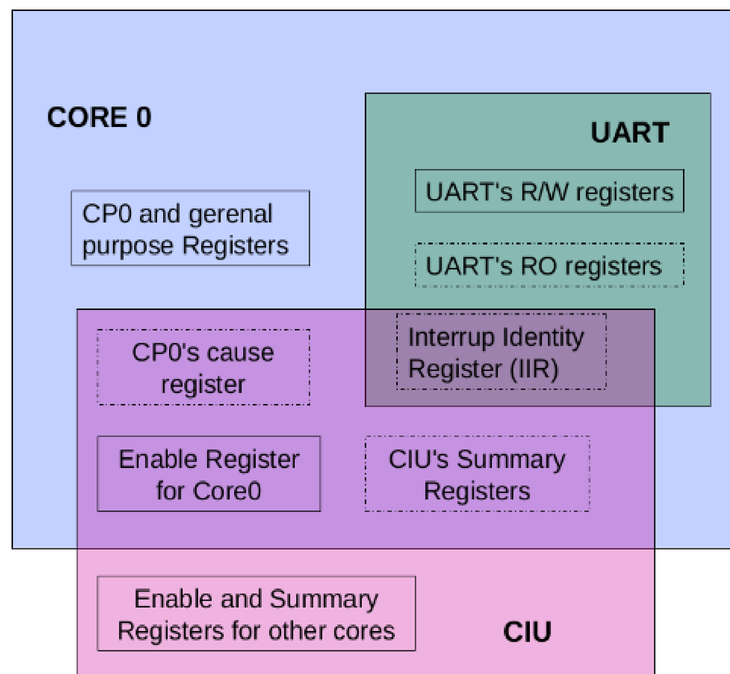


Figure 12: Memory mapping between core and external device

4.7 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, Virtio_Blk or Vhost_Blk cannot be used directly for this purpose and some changes have to be made.

4.7.1 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 13 shows the code that needed to be added in 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.

A file should be created 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, command "mknod /dev/vda b 253 0" is used (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 14.

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.

```
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 13: Code Snippet from virtio_blk.c file

```
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 14: Commands for Device Creation

4.7.2 Vhost Block Configuration

Vhost block is used 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 with CAVIUM MIPS64 kernel as well as with Vhost implementation. Once the module is built and loaded in host kernel, a file for this module should be created. 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.

4.7.3 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.

4.8 Networking

For enabling networking in type-II hypervisor, implementation of a network device was inevitable. 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 15 shows the complete 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.

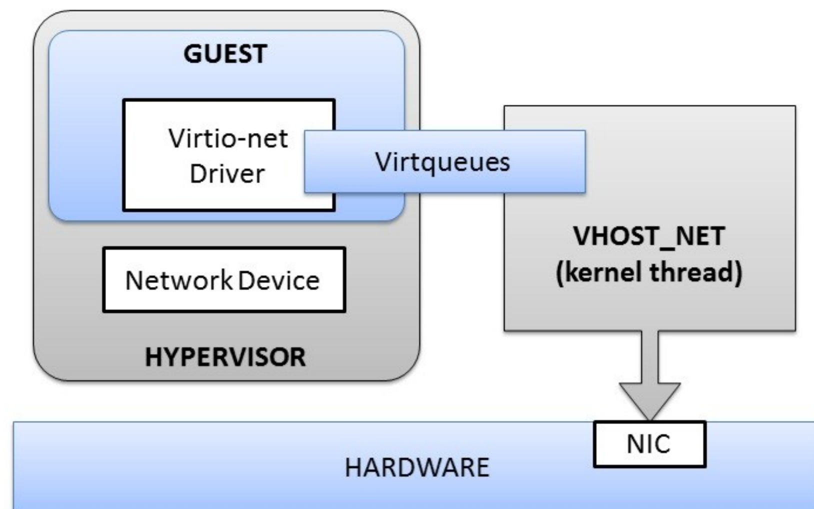


Figure 15: Networking Infrastructure of Type-2 Hypervisor

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.

4.8.1 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.

Network device is implemented using virtio-net. In native kernels, virtio devices are implemented as standard PCI devices but emulation of PCI bus would be less efficient and more time consuming. MMIO (Memory Mapped Input Output) device is implemented 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, this device needs to be registered as platform device and specify its base address and interrupt line.

4.8.2 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.

4.8.3 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.

4.8.4 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 16 shows the communication between the units involved in network implementation.

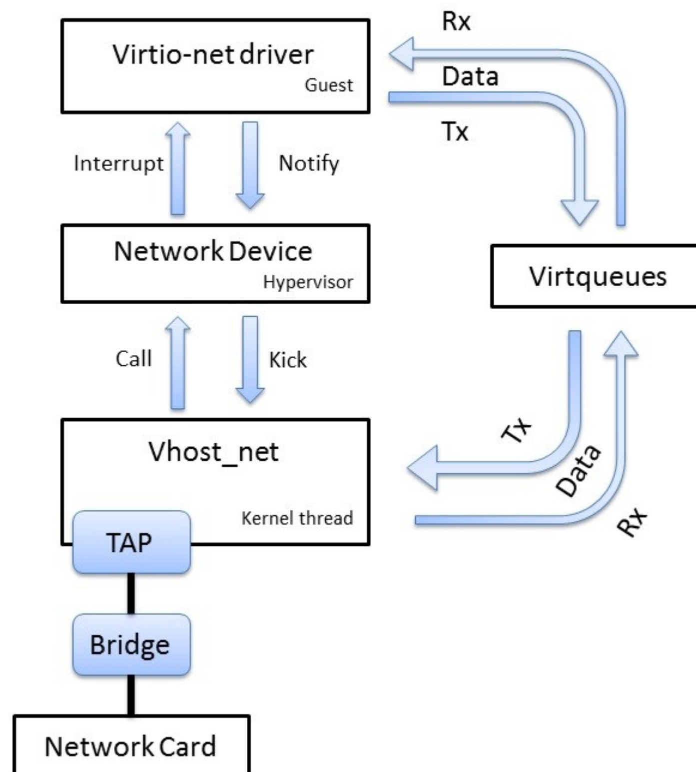


Figure 16: Communication between Virtio-Net, Vhost-Net and Network Device

5. Performance Evaluation

For evaluating the performance of hypervisor, different parameters are calculated. Tests used for evaluation were both performed on native system and hypervisor. 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. So, the comparison is performed between the execution of hypervisor emulating OCTEON CN5700 and actual native system i.e. real board OCTEON CN5700.

5.1 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 Fig 17. These values are mean of the multiple readings. The time taken by the native system is just 47 sec. While hypervisor takes approximately 75 sec.

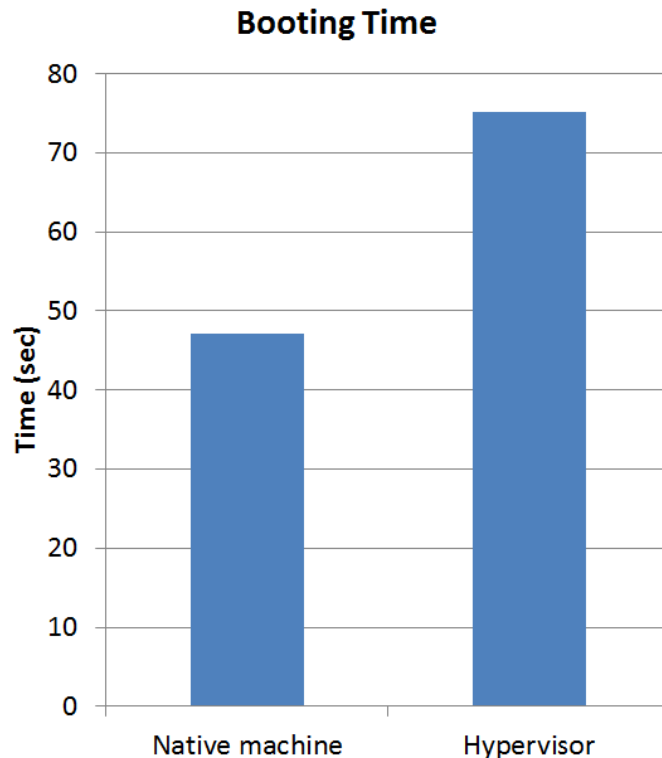


Figure 17: Comparison of Booting Time

5.2 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 1 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 and native hardware.

Table 1: Lmbench Output for Different Memory Operations

Operations	Native System (Mb/sec)	Hypervisor (Mb/sec)
rd	1870	126
wr	8190	134
rdwr	1544	50
cp	749	65
fwr	3055	40
frd	840	38
fcp	520	20
bzero	3123	24
bcopy	673	20

The table 2 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 3 shows the latencies (in millisecond) in

native system and hypervisor, while performing the file mmap. Table 4 and 5 shows the bandwidth and latencies for 2Mb file reading respectively.

Table 2: Lmbench Bandwidth Results for File mmap

Memory bandwidth for file mmap	Native System (Mb/sec)	Hypervisor (Mb/sec)
open2close	856	9.5
mmap_only	1454	68

Table 3: Lmbench Latency Results for File mmap

Latencies for file mmap	Native System (msec)	Hypervisor (msec)
open2close	1.2	100.5
mmap_only	0.7	15

Table 4: Lmbench Bandwidth Results for File Reading

Memory bandwidth for file reading	Native System (Mb/sec)	Hypervisor (Mb/sec)
open2close	810	13
io_only	830	13.3

Table 5: Lmbench Latency Results for File Reading

Latencies for file reading	Native System (msec)	Hypervisor (msec)
open2close	2.5	153
io_only	2.5	151

Table 6 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 6: Lmbench Bandwidth Results for pipe and unix

	Native System (Mb/sec)	Hypervisor (Mb/sec)
bw_pipe	850	2.75
bw_unix	1450	3.7

The table 7 below shows the network through put of hypervisor and native system. It a client-server test program with a message size of 65536 bytes.

Table 7: Lmbench Results of Network Through-put

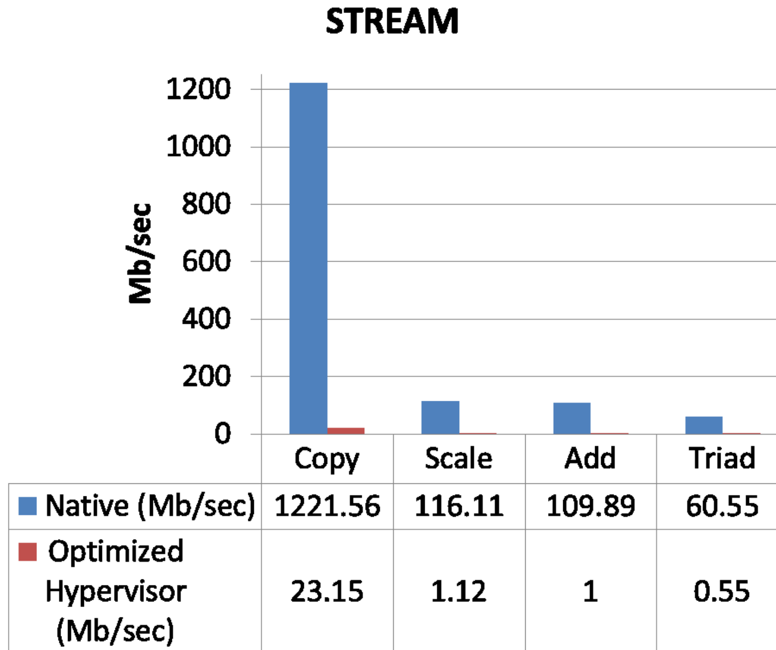
	Native System (Mb/sec)	Hypervisor (Mb/sec)
bw_tcp	11.57	2.01

5.3 STREAM

STREAM benchmark is also executed, which performs these 4 basic memory operations and measure transfer rates. “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.

1. copy a(i) = b(i)
2. Scale a(i) = q*b(i)
3. Sum a(i) = b(i) + c(i)
4. Triad a(i) = b(i) + q*c(i)

The figure 18 shows the transfer rates and figure 19 shows the latencies for all above operations of both hypervisor and native.



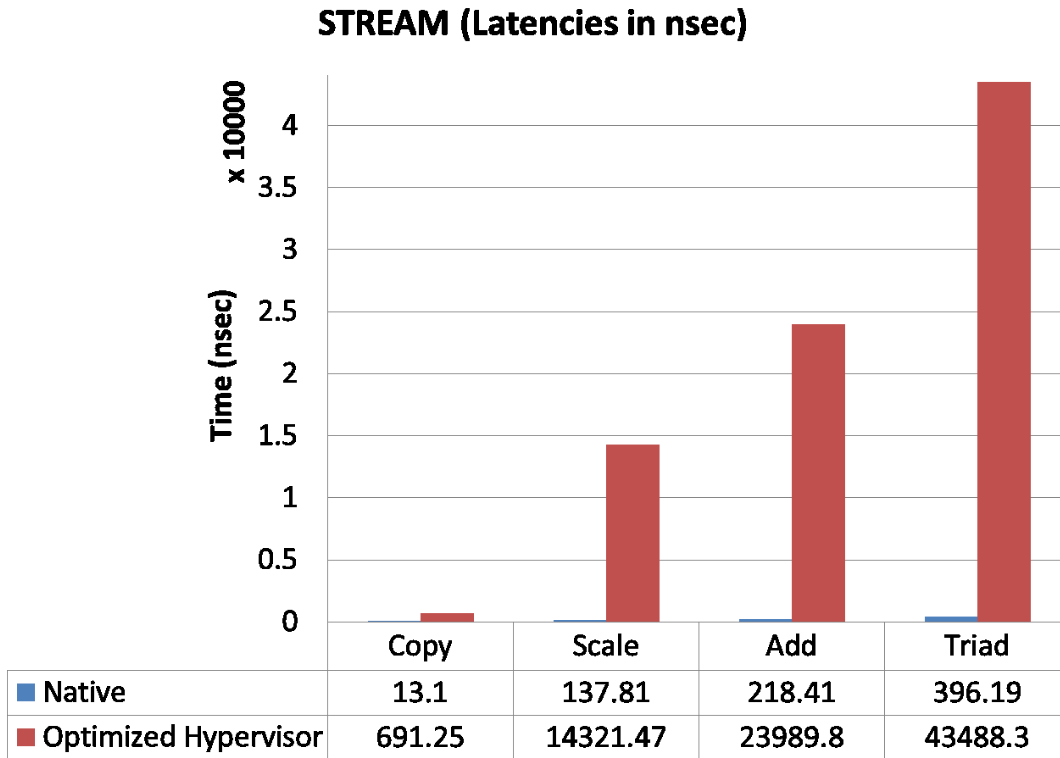
	Copy	Scale	Add	Triad
Native (Mb/sec)	1221.56	116.11	109.89	60.55
Optimized Hypervisor (Mb/sec)	23.15	1.12	1	0.55

Figure 18: STREAM Results for hypervisor and Native System

5.4 Linux Testing Project (LTP)

For functional testing of Type-II hypervisor, LTP’s syscall tests were executed. 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.



	Copy	Scale	Add	Triad
Native	13.1	137.81	218.41	396.19
Optimized Hypervisor	691.25	14321.47	23989.8	43488.3

Figure 19: Latencies for Stream Operations