This document is version 6 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.
# Table of Contents

1 Project Description ........................................................................................................... 4
2 High Level Design ............................................................................................................. 4
3 Implementation Strategies ............................................................................................... 5
   3.1 Instruction Level Execution Model ............................................................................. 6
   3.2 Block Level Execution Model .................................................................................... 7
4 System Development ......................................................................................................... 9
   4.1 Memory Management Unit (MMU) .......................................................................... 9
      4.1.1 GVA to GPA Translation .................................................................................. 9
      4.1.2 GPA to HVA Translation ................................................................................ 10
      4.1.3 Page Table ...................................................................................................... 10
      4.1.4 Translation Look-aside Buffer (TLB) ............................................................. 10
   4.2 Software Cache ......................................................................................................... 11
      4.2.1 Cache Storage .................................................................................................. 11
      4.2.2 Searching a Block ......................................................................................... 11
      4.2.3 Block Retrieval .............................................................................................. 12
      4.2.4 Replacement policy ......................................................................................... 12
   4.3 Exception Handling ................................................................................................. 12
      4.3.1 SIGFPE: Floating point exception handling ................................................. 14
      4.3.2 SYSCALL: System call handling ..................................................................... 14
   4.4 Central Interrupt Unit (CIU) .................................................................................... 15
   4.5 SMP Support ........................................................................................................... 17
      4.5.1 Inter-Core Communication Through CIU ......................................................... 18
5 Testing Infrastructure ...................................................................................................... 20
   5.1 Test cases ................................................................................................................. 20
      5.1.1 Matching System states .................................................................................. 20
      5.1.2 Execution Path ............................................................................................... 20
      5.1.3 Comparing Console Output .......................................................................... 21
      5.1.4 Progress ......................................................................................................... 21
5.2 Testing with SMP Support ................................................................. 21
6 Test Results ........................................................................................... 22
   6.1 Output of System State Matching Test ........................................ 22
   6.2 Output of Execution Path Test .................................................... 23
   6.3 Output of TLB Testing ................................................................. 23
   6.4 Output of CIU Testing ................................................................. 27
   6.5 Output of Hypervisor Console .................................................... 29
7 Performance Optimization ...................................................................... 30
   7.1 Performance Tuning ..................................................................... 30
   7.2 Performance Improvements ....................................................... 32
      7.2.1 Code Structural Enhancement ........................................... 32
      7.2.2 Hardware Platform ............................................................ 33
   7.3 Potential Future Optimization ...................................................... 34
      7.3.1 Block Linking ................................................................. 34
      7.3.2 Compiler like Translation Optimization ................................ 34
      7.3.3 Data TLB Checking in Epilogue ....................................... 34
      7.3.4 Static Analysis and Modification ...................................... 34
      7.3.5 Performance Counter Monitoring ................................... 34
   7.4 Correctness related bug fixes ...................................................... 35
8 Impact on Project Progress .................................................................... 35
# TABLE OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Multithreaded design of Type-2 hypervisor</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Instruction level execution model</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Block level execution model</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Exception handling in user mode</td>
<td>13</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Code snippet showing the emulation of exception handling</td>
<td>14</td>
</tr>
<tr>
<td>Figure 6(a)</td>
<td>(ciu) Interrupt distribution from external devices to core</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Memory mapping between core and external devices</td>
<td>17</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Multithreaded view of hypervisor with configured number of cores and external device unit (i.e. CIU)</td>
<td>18</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Execution flow of hypervisor with SMP</td>
<td>19</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Output of system state matching test</td>
<td>22</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Output of execution path test</td>
<td>23</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Searching for random TLB</td>
<td>24</td>
</tr>
<tr>
<td>Figure 13</td>
<td>TLB entries in TLB table</td>
<td>25</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Output of TLB</td>
<td>26</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Output of CIU. No pending interrupt on core 1</td>
<td>27</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Output of CIU. No pending interrupt on core 0</td>
<td>28</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Booting log of hypervisor</td>
<td>29</td>
</tr>
</tbody>
</table>
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 is augmented by implementing cases where one sensitive instruction is replaced by more than one instruction. In 4th deliverable, dynamic code patching is applied on demand. In 5th deliverable, guest kernel booting completes and starts creating user mode processes. In 6th deliverable, SMP support is added to whole infrastructure and many performance related bugs were fixed.

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.

![Hypervisor (Linux Process)](image)

**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.

![Instruction Level Execution Model Diagram](image-url)
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: Block level execution model
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.

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/O 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, page table functions are called and GVA is searched in the page table. If found, 
corresponding HVA is returned, otherwise a new memory region is allocated using mmap() and 
its address is returned. Current implementation does not impose any restriction on memory
allocation (i.e. it will be implemented in future deliverables). To reclaim guest memory, one possible solution is to use OOM killer of guest kernel.

4.2 **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.

### 4.2.1 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.

### 4.2.2 SEARCHING A BLOCK

Software cache is capable of searching any block in time $O(\log n)$ 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 $O(1)$. 
4.2.3 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.

4.2.4 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.

4.3 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 is 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 used 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 4 shows the overall flow and Figure 5 shows a snippet of hypervisor code, dealing with exception handling.

![Figure 4: Exception handling in user mode](image)

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. Two case studies are discussed below to elaborate the implementation.

### 4.3.1 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.

### 4.3.2 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.

```c
case HANDLE_SYSCELL_INT: {
    printf("Syscall\n");
    Gobj->prev_PC = *blockStartPC;
    Gobj->prev_PC = TLB_Exception::Perform_Exception(core0->getPC0(), HANDLE_SYSCELL_INT);
    BlockManipulation::setUContext(Gobj->prev_PC,FETCH_NEXT_BLOCK);
    fetchPlaceBlock(GVA);
    return true;
    break;
}
```

```c
default: {
    switch(cntrlMark) {
    case HANDLE_TLB_PROBE: {
        if(DB0)printf("HANDLE_TLB_PROBE\n");
        core0->getTLB()->tlb_probe();
        if(DB0)printf("Index Register = 0x\0161lx\n",core0->getTLB()->get_index());
        return true;
        break;
    }
```

**FIGURE 5:** Code snippet showing the emulation of exception handling
4.4 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 6 (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 6 (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.
FIGURE 6(A): (CIU) Interrupt distribution from external devices to core

FIGURE 6(B): Internal working of CIU, inwards arrow comes from external devices and outward arrow goes to all cores
In integrated code, shared memory regions are defined for CIU to work with other components of virtual board (see figure 1). Figure 7 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.

4.5 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 8 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 in 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 9 shows the modified flow chart of hypervisor.

4.5.1 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 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.
Our 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 9: Execution flow of hypervisor with SMP
5 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.

5.1 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.

5.1.1 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.

5.1.2 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”.

5.1.3 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.

5.1.4 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.

5.2 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.

6 TEST RESULTS

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

6.1 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 10.

![Output file](Output_file.png)

**Figure 10: Output of system state matching test**
6.2 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 11.

![Output of execution path test](image)

**FIGURE 11:** Output of execution path test

6.3 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 12.
FIGURE 12: 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.
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 13. 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 14(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 14(b). Then whole page table is printed in figure 14(c) and reverse page table, shown in figure 14(d), is also managed to use for future testing of hypervisor.
FIGURE 14: 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.
6.4 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 15, 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 15 shows that initially cause register is initialized by a garbage value.

![Source Code and Output](image)

**FIGURE 15: 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 xxxxxxxxxx6ac8 is changed to xxxxxxxxxx66c8 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. xxxxxxxxx6770 changes to xxxxxxxx6370.

In figure 16, 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. xxxxxxxxdac8 changes to xxxxxxxxcac8 in the output. For core1, nothing is enabled so all three bits would be cleared i.e. xxxxxxxxd770 changes to xxxxxxxxc370.

FIGURE 16: Output of CIU. No pending interrupt on core 0
6.5 OUTPUT OF HYPERVISOR CONSOLE

During execution, hypervisor makes a call to the code written for console I/O. On console attachment, the binaries, executing within hypervisor, can start printing on the hypervisor console. To validate virtual execution of binaries, hypervisor console output (e.g. shown in Figure 17) was compared with that of real host system console. (For complete log please look at the VM booting log file attached)

FIGURE 17: Booting log of hypervisor
7 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.

7.1 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 Table 1. The data shows that address translation is taking much more time as compared to other functions and it should be optimized.

**TABLE 1: SORTED FLAT PROFILE BEFORE OPTIMIZATION (SHOWING MOST TIME CONSUMING FUNCTIONS ONLY).**

<table>
<thead>
<tr>
<th>Count</th>
<th>Function Name</th>
<th>Net Time (sec)</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>49107203</td>
<td>MMUTranslator::GVAtOGPA</td>
<td>2326.3283</td>
<td>4572.8963</td>
</tr>
<tr>
<td>29513958</td>
<td>BlockExecController::fetchnPlaceBlock</td>
<td>2283.8921</td>
<td>3057.8145</td>
</tr>
<tr>
<td>50000001</td>
<td>BlockExecController::handleRequest</td>
<td>1836.5265</td>
<td>5972.9834</td>
</tr>
<tr>
<td>49107203</td>
<td>GPAtoHVATranslator::GPA_to_HVA</td>
<td>1221.9575</td>
<td>1511.028</td>
</tr>
<tr>
<td>49107203</td>
<td>MMUTranslator::verify_priviliges</td>
<td>1063.2158</td>
<td>1378.4329</td>
</tr>
<tr>
<td>49107203</td>
<td>GVAtHVATranslator::GVAtOHVA</td>
<td>852.3648</td>
<td>5487.4371</td>
</tr>
<tr>
<td>47218428</td>
<td>MMUTranslator::look_staic_translation_32bit</td>
<td>789.6234</td>
<td>896.4091</td>
</tr>
</tbody>
</table>
**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 Table 2. 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.

**TABLE 2: SORTED FLAT PROFILE AFTER SOFTWARE CACHE IMPLEMENTATION**

<table>
<thead>
<tr>
<th>Count</th>
<th>Function Name</th>
<th>Net Time (sec)</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29513958</td>
<td>BlockExecController::fetchnPlaceBlock</td>
<td>2283.8921</td>
<td>2780.8145</td>
</tr>
<tr>
<td>50000001</td>
<td>BlockExecController::handleRequest</td>
<td>1804.6144</td>
<td>4351.9108</td>
</tr>
<tr>
<td>49107203</td>
<td>GVAToHVATranslator::GVAToHVA</td>
<td>742.5720</td>
<td>1468.1078</td>
</tr>
<tr>
<td>49106050</td>
<td>GVAToHVATranslator::Check_cache</td>
<td>449.7157</td>
<td>449.7157</td>
</tr>
<tr>
<td>35715</td>
<td>MMUTranslator::GVAToGPA</td>
<td>1.306426</td>
<td>2.650513</td>
</tr>
<tr>
<td>35716</td>
<td>GPAToHVATranslator::translate_GPA_to_HVA</td>
<td>0.270219</td>
<td>0.270219</td>
</tr>
<tr>
<td>26628</td>
<td>MMUTranslator::look_staic_translation_32bit</td>
<td>0.178782</td>
<td>0.310867</td>
</tr>
</tbody>
</table>

**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 3: SORTED FLAT PROFILE AFTER IN-PLACE BLOCK EXECUTION IMPLEMENTATION

<table>
<thead>
<tr>
<th>Count</th>
<th>Function Name</th>
<th>Net Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>215982611</td>
<td>Processor_IPE::handleRequest()</td>
<td>14.03</td>
</tr>
<tr>
<td>124936958</td>
<td>Processor_IPE::fetchNextBlock()</td>
<td>12.65</td>
</tr>
<tr>
<td>124936958</td>
<td>TranslatedBlockCache_IPE::doesExists()</td>
<td>9.77</td>
</tr>
<tr>
<td>39934396</td>
<td>TransInsAllocator&lt;unsigned int&gt;::construct()</td>
<td>7.64</td>
</tr>
<tr>
<td>86515038</td>
<td>Processor_IPE::handleLDST()</td>
<td>6.82</td>
</tr>
<tr>
<td>215982610</td>
<td>Processor_IPE::C2Assembly2C()</td>
<td>6.77</td>
</tr>
<tr>
<td>87261169</td>
<td>Processor_IPE::incrementTrapCount()</td>
<td>6.22</td>
</tr>
</tbody>
</table>

#### 7.2 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.

##### 7.2.1 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.

### 7.2.2 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**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>2GB</td>
</tr>
<tr>
<td>TLB_entries</td>
<td>32</td>
</tr>
<tr>
<td>CPU Model</td>
<td>Cavium Octeon II V0.3</td>
</tr>
</tbody>
</table>

The new evaluation board has the following specifications:

**TABLE 5: SPECIFICATION OF NEW EVALUATION BOARD**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>8GB</td>
</tr>
<tr>
<td>TLB_entries</td>
<td>128</td>
</tr>
<tr>
<td>CPU Model</td>
<td>Cavium Octeon II V0.10</td>
</tr>
</tbody>
</table>
7.3 **Potential Future Optimization**

Various other optimizations are also possible to avoid performance degradation of code executed in virtual environment. We have plans to implement few promising techniques along the course and their effect will be available in the future deliverables. Here we briefly describe the potential optimization techniques.

### 7.3.1 Block Linking

In-place execution also necessitates the linkage of blocks so that guest code could follow the intended execution path of code. It is expected to improve the performance due to less intervention of hypervisor.

### 7.3.2 Compiler Like Translation Optimization

In current implementation of hypervisor, execution context is confined to single instruction of the block. Data produced/consumed by the instruction is updated immediately. On the other hand, compilers often have a broader vision and do not update data immediately if it is going to be consumed by some following instruction(s). We may opt for this optimization in future deliverables.

### 7.3.3 Data TLB Checking in Epilogue

Moving data TLB checking to the epilogue of a block is likely to reduce the C-assembly code jumps. Software exception handling is expensive and this optimization could lead to infrequent software exceptions.

### 7.3.4 Static Analysis and Modification

Preprocessing of guest binaries by static analysis is likely to reduce the runtime overhead. Static code modifications could avoid block translation at runtime. However, there are some corner cases that could only be handled until we have the runtime information.

### 7.3.5 Performance Counter Monitoring

Instead of using instrumentation of functions, we could fine tune performance using performance monitoring unit (PMU) of modern hardware. The real challenge of this optimization is to select
the most appropriate performance counters. A deep system level understanding is required to collect and analyze the data collected using performance counters.

7.4 Correctness related bug fixes

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.

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.

8 Impact on Project Progress

Hypervisor development is a complex task. We face and overcome many technical challenges along the way. The timing performance we were facing in the previous deliverable is majorly resolved. Now booting time is reasonable. Addition of SMP has introduced multithreading and complex communication between cores through external interrupt distribution unit (i.e. CIU). Interrupt dispatching mechanism needs to be more developed for proper working of system.

References