A Dual-TLB Method for MIPS Heterogeneous Virtualization

Wang Zhenhua*†‡¶, Jin Guojie*†§¶, Wang Wenxiang*†§¶
*State Key Laboratory of Computer Architecture, ICT, CAS
†Institute of Computing Technology, Chinese Academy of Sciences
‡University of Chinese Academy of Sciences
§Loongson Technology Corporation Limited
¶{wangzhenhua, jinguojie, wangwenxiang}@ict.ac.cn

***Abstract*—DBT (Dynamic Binary Translator) can directly translate and execute binary programs, enabling a compatible system by virtualizing one machine (guest) on another (host). However, the memory virtualization of guest brings in a great overhead, due to the several steps it takes to translate GVA (Guest Virtual Address) into HPA (Host Physical Address). For QEMU, a DBT with efficient memory virtualization mechanism, more than 60% of the translated code is used to virtualize memory, thus leading to a low performance of guest. In this paper, we employ the Co-Design methodology to optimize memory access performance of guest. This optimization is primarily focused on two aspects. First, hardware extensions are designed and implemented to conduct GVA to HPA translation directly. Second, we modify QEMU to cooperate with hardware to reduce translated code. By this means, the cost of memory virtualization is completely eliminated, resulting in a significant enhancement of the performance of the Loongson binary translation system. The experimental data implies that the performance of guest has been dramatically improved by 100 times for peak performance and 19.12% for average performance over the previous system.**

***Keywords*—Dual-TLB; Virtual Machine; Binary Translation; Heterogeneous Virtualization; MIPS; QEMU;**

I. Introduction

Currently, the x86 architecture is dominant in the markets of both desktop and server. As most software in these two areas is x86-based, developing a x86-compatible processor has been a heated point of research in both industry and academia[1–4]. Based on this reality, it's urgent to make Loongson processor, a MIPS compatible one, compatible with x86 architecture so as to run a variety of software. Therefore, its software ecosystem is expanded.

Binary translation is a main technology to virtualize heterogeneous guest on host through translating and executing binary program directly[5, 6]. With this technology, we can obtain a compatible system at a low cost. However, due to the several steps it takes to translate GVA into HPA, the memory virtualization subsystem suffers a great overhead. As a result, the performance of guest is not remarkable. For example, by using QEMU, a DBT which supports multiple architectures[7], to virtualize x86 on Loongson processor, more than 60% of the translated code is used to translate address[8].

In recent years, the Co-Design methodology has been widely applied to cut down the expense of memory virtualization[9, 10]. Intel and AMD develop processor supporting EPT (Extended Page Tables)[11] and NPT (Nested Page Tables)[12] to improve memory virtualization. IBM introduces hardware extensions in Power embedded A2 core to enhance virtualization performance[13]. Although these solutions above have a positive impact on performance, they are focused on homogeneous virtualization. For heterogeneous virtualization, Transmeta Crusoe[4, 14] and DAISY[15] uses VLIW instruction sets as their target ISA, gaining a high performance virtual system. Nevertheless, their solutions of implementing compatible system are too expensive.

This paper pays close attention to the expense of memory virtualization of QEMU. By employing the Co-Design methodology, we optimize the performance of guest mainly on two aspects. First, hardware extensions are designed and implemented to improve memory virtualization in Loongson GS464e core[16], which is integrated in Loongson 3A1500 processor. Second, we modify QEMU to cooperate with the hardware extensions to reduce translated code. By this means, the software overhead of guest's memory virtualization is completely eliminated, resulting in a significant enhancement of the performance of the Loongson BTS (Binary Translation System). The experimental data shows that the performance of guest has been dramatically improved by 100 times for peak performance and 19.12% for average performance over the previous system.

The rest of this paper is organized as follows: In Section II, the memory virtualization features of QEMU are discussed and the drawback of x86's performance is indicated. Section III proposes the Co-Design binary translation system which is equipped with our machine isolation technique. In Section IV, we evaluate the performance impact of the optimization on Loongson BTS. Section V concludes.

II. Traditional Memory Virtualization Mechanism of QEMU

QEMU supports a variety of architectures: x86, PowerPC, ARM and SPARC as host; and x86, PowerPC, ARM, SPARC, Alpha, and MIPS as guest[7]. By virtualizing x86 with QEMU, Loongson can get its BTS, which is compatible with x86.

Supported by the National Sci&Tech Major Project (No.2009ZX01028-002-003, 2009ZX01029-001-003, 2010ZX01036-001-002,2012ZX01029-001-002-002,2014ZX01020201), National Natural Science Foundation of China (No.61221062,61133004, 61173001, 61232009,61222204,61432016)the National High Technology Development 863 Program of China (2012AA010901,2012AA011002,2012AA012202,2013AA014301)

For memory virtualization of QEMU, the physical memory of guest is emulated in its virtual address space, shown in Figure 1a. Therefore, it takes two steps to convert GVA to HPA when executing the translated instructions of MARI (Memory Access Related Instruction) of guest. First, the software walks guest's page table to translate GVA into HVA (Host Virtual Address). Second, host uses TLB (Translation Look-aside Buffer) to conduct HVA to HPA translation. There are two results of the latter step: TLB hit, TLB translating HVA into HPA directly; TLB miss, host searching its page table to finish address translation. For typical applications, the address translation conducted by software introduces a huge overhead.

To reduce this expense, QEMU employs an efficient software TLB mechanism to translate guest address. The mechanism consists of three parts: a data structure to store GVA-HVA mapping, an algorithm to walk the GVA-HVA mapping to translate address and a series of functions to handle software TLB miss. They are discussed as below.

A. The Data Structure of QEMU's Software TLB

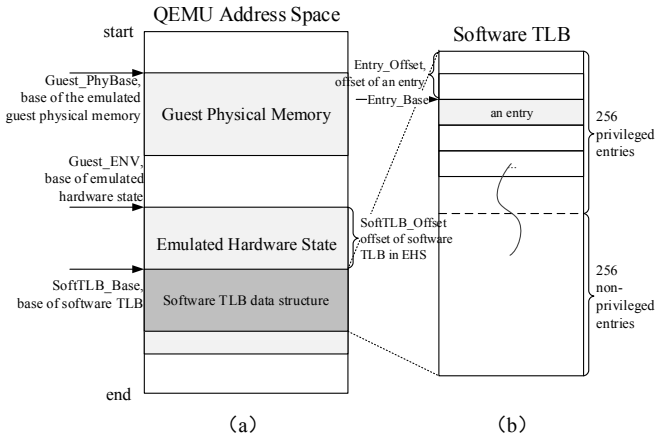

Fig. 1. The Structure of QEMU's Software TLB

The data structure of QEMU's software TLB is a hash table, which is part of the emulated guest's hardware state, shown in Figure 1a. The structure contains 256 entries representing privileged state and 256 entries representing non-privileged state, shown in Figure 1b. These items are used by memory access operations in corresponding state to convert the address. A GVA is mapped to a unique entry which consist of three GVB (Guest Virtual page Base) and an `addr_end`, shown in Figure 2a. The three GVB are: `GVB_read`, address of read operation; `GVB_write`, address of write operation; and `GVB_code`, address of executable code. Besides, these GVB also control the permissions of read, write, and execution of guest's page: if the TLB entry has a valid `GVB_read` (`GVB_code`) address, the page is readable (executable); if `GVB_write` is valid and has the writable flag (the 3th low bit of it) is 0, the page can be written. The `addr_end` is the difference of HVA to GVA, and the three GVB are all mapped to it, shown in Figure 2c. When the structure is used to convert the address, the algorithm of the mechanism will extract one of three GVB according to the kind of memory access. Therefore, there is no need to conduct a bit manipulation to check read/write permission. For example, when performing a translated store operation with the entry shown in Figure 2b, the second field will be extracted and used directly. Furthermore, due to the simple mapping of GPA (Guest Physical Address) to HVA ($HVA = Guest\_PhyBase + GPA$), QEMU does not have to maintain GPA-HVA mapping. These characteristics reduce the complexity of the software TLB mechanism and benefit address translation.

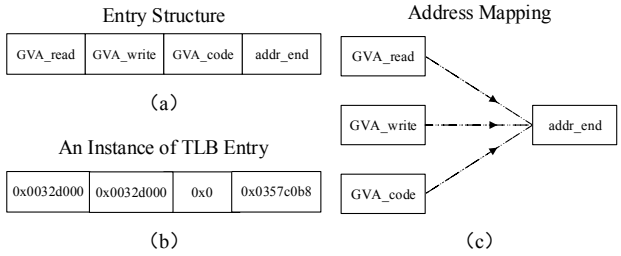

Fig. 2. The Structure of QEMU's Software TLB Entry

B. The Algorithm of QEMU's Software TLB

The algorithm of QEMU's software TLB walks the data structure to convert GVA to HVA. There are two situations of the walking: software TLB hit, QEMU getting `addr_end` from the software TLB; software TLB miss, QEMU traversing x86 page table to conduct address translation. The specific steps of the algorithm are shown in Figure 3.

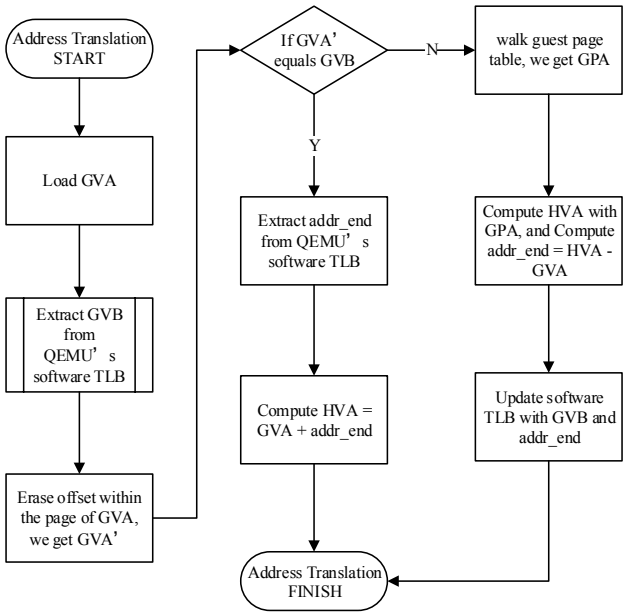

Fig. 3. The Flowchart of QEMU's software TLB Algorithm

As we can see from Figure 3, there are many steps of the algorithm. When it comes to virtualizing x86 on Loongson, Practically, QEMU generates ten more MIPS instructions for each x86 MARI to accomplish the address translation. An instance of the instruction translation is shown in Figure 4, executing the algorithm shown in Figure 3. Typically, 30% of the instructions of a program are memory access related[17]. After translation, 60% of the instructions are used to translate address[8], leading to a slow guest. So, when compared with hardware TLB, the mechanism of QEMU's software TLB

suffers a great overhead. Optimization for memory access instructions can significantly improve the performance of guest.

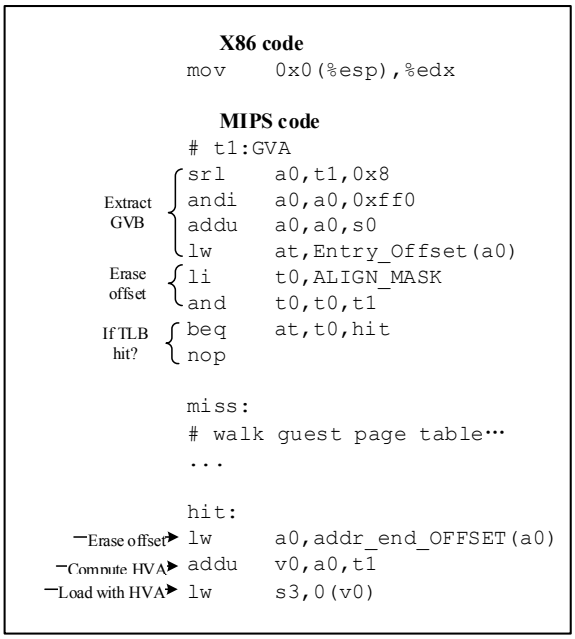

Fig. 4. An Instance of Instruction Translation of QEMU's Software TLB Mechanism

C. The Handler of Software TLB Miss Exception of QEMU

When the software TLB miss exception occurs during the procedure of address translation, QEMU will call a handler to walk x86's page table to convert the address. The handler is called right after the TLB lookup operation, labeled as `miss` in Figure 4. It breaks the execution of guest, leading to a virtual machine exit. The handler aims at searching x86 page table to convert GVA to HVA, updating the software TLB with the GVA-HVA mapping and emulating the x86 instruction which triggered software TLB miss exception. The details of the handler are ignored in this paper.

III. The Co-Design Loongson Heterogeneous Memory Virtualization

As talked in last section, QEMU has a great overhead due to the memory virtualization of guest. To eliminate the cost, this paper employs the Co-Design methodology to propose an optimization. Similar with QEMU's software TLB mechanism, the optimization mainly consists of three parts. 1. **The structure:** the machine isolation technique is introduced, including a specialized TLB and a specialized instruction which are designed and implemented in Loongson GS464e core. The specialized TLB directly translates GVA into HPA while the specialized instruction distinguishes guest's memory access from the host's. 2. **The algorithm:** with the hardware extensions introduced, we optimize the instruction translation rules of QEMU to reform the algorithm of address translation, reducing the size of translated code significantly. 3. **The handler**: while the algorithm of address translation modified, the virtual machine exit will be triggered by hardware. To handle the virtual machine exit, a software mechanism is designed, guaranteeing the stability of binary translation system. Together with these three parts, we isolate address space of the host and guest. Therefore, in Loongson BTS, the address of guest can be directly used by the specialized TLB built in host processor, leading to the execution of guest's MARI is as fast as the host's.

A. The Machine Isolation Technique Built in Loongson GS464e Core

As described in Subsection II-B, even with a well-designed software TLB mechanism, QEMU still suffers from a great overhead when it translates guest's address. To reduce the cost, a straightforward idea is using hardware TLB to translate GVA into HPA directly, shown in Figure 5.

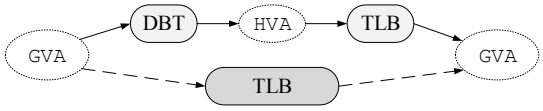

Fig. 5. Simplify Address Translation with Hardware TLB

However, when virtualizing x86 on MIPS, it is impossible to translate all x86 addresses by using MIPS HTLB (Host TLB). The reason is that the address space mechanism of MIPS and x86 are different. For MIPS, the most significant bit of the address can only be used in privileged state and addresses in `0x8000_0000` - `0xBFFF_FFFF` never use TLB. For an x86 machine running Linux, the privileged address space of 32-bit address space is `0xC000_0000` - `0xFFFF_FFFF`. Both privileged and non-privileged x86 address is translated by TLB. To sum up, there are two difficulties: x86 and MIPS have different privileged configuration of address space and x86 address between `0x8000_0000` - `0xBFFF_FFFF` cannot be translated by MIPS HTLB.

To deal with this problem, we introduce the machine isolation technique. Contrast with the traditional hardware-assisted memory virtualizations which are focused on virtualizing guest of the same architecture as host, this machine isolation technique accelerates both homogeneous and heterogeneous virtualization. The key idea of machine isolation technique is isolating address space of different machines by labelling their MARI. In practice, for virtualizing x86 on MIPS, this paper proposes MID (Machine IDentifier) to distinguish address of MIPS (host) and x86 (guest). When deployed on Loongson GS464e core, the technique includes two aspects: a specialized TLB with MID built in that it can directly translate address of a heterogeneous guest; a specialized instruction indicates the MID of next instruction, distinguishing guest's memory access from the host's.

1) The Specialized TLB Equipped with MID: Since traditional MIPS HTLB cannot convert x86 address, we tend to design a GTLB (Guest TLB) to conduct the address translation. The designing of GTLB should follow two principles: GTLB's structure and its management mechanism should be similar to HTLB's to minimize the modification to host's microarchitecture; GTLB must be functionally equal to QEMU's software TLB to inherit its features.

Luckily, in 2009, MIPS introduced the MMU configuration including VTLB (Variable-page-size TLB) and FTLB (Fixed-page-size TLB). MIPS VTLB entry contains five domains,

shown in Figure 6a. VPN2 (Virtual Page Number) are the high-order bits of a program address. ASID (Address Space IDentifier) holds the current address space. PageMask can be used to create entries that map pages bigger than 4 KB, which means the mapped page of each VTLB entry may have its own size. PFN (Physical Frame Number) is the physical page base. G represents privileged state when it's set to 1. Flag domain is composed by three subfields: V indicates whether the mapped page is valid (mapped to a physical page). D indicates whether the page is writeable, the write operation on a read-only page will be trapped. C describes the cache configuration. G and C bit are both neglected in this paper. Different from VTLB, FTLB doesn't have PageMask for each entry, shown in Figure 6b, while it has a global PageMask instead. FTLB's global PageMask is used to indicate the page size of all entries in FTLB, since their page sizes are all equal. [18] and [19] provide more details on MIPS MMU configuration.

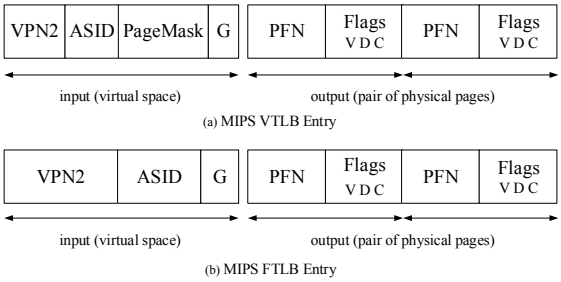

Fig. 6. Entry Structure of VTLB and FTLB

In the dual-TLB configuration, these two TLB have the same program interface. Index and Random register have been expanded to support FTLB. When software fills TLB entry using `tlbwr`, the entry will be written into FTLB randomly if the page size indicated by PageMask equals FTLB's page size, otherwise, a randomized VTLB entry will be updated. By configuring these two TLB with different page sizes, VTLB as 16KB and FTLB as 4KB, we can maintain them separately. In this way, VTLB and FTLB are taken as HTLB and GTLB, and we will quote them as above in the rest of this paper.

When MIPS TLB translates address, the input is $< ASID, PageMask, VPN2 >$, hardware will select the right entry automatically. If TLB hits, the output is $< PFN >$; otherwise, TLB exception is triggered. The G bit and the Flag domain are corresponding with TLB matching. For example, when a store operation is being executed, the translation will still fail under the situation: there is a TLB entry matches the input, but its D bit is 0.

In spite of the fact that HTLB stores host address mapping and GTLB caches guest address mapping, GTLB is not able to translate x86 address because it follows MIPS address translation mechanism. To fix this up, we implement a 2-bit MID (`0 - 3`) on reserved bits of each TLB entry.[16, 20] Once matching TLB entries, MID describes the corresponding machine which the entry belongs to: MID `0` or `1` indicates the address following MIPS address mechanism, which, `0` is used by native MIPS software, `1` is used for homogeneous virtualization[21]. MID `2` or `3` indicates the address belonged to a heterogeneous guest. It's attractive that the address translation mechanism of MID `2` or `3` is flexible and configurable.

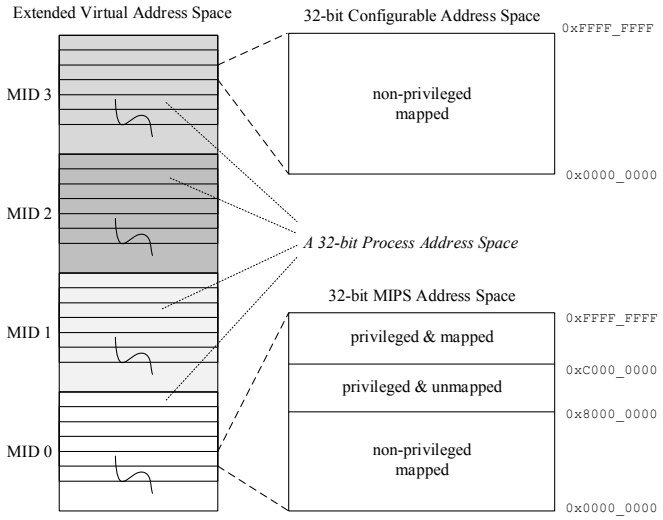

Fig. 7. Extended Virtual Address Space of Machine Isolation Technique

Meanwhile, the default MID is `0`, so legacy applications can run on our system without any modification. In this way, an extended virtual address space $< MID, ASID, Address >$ is formed. In which, an extended virtual address indicates an accurate virtual page (via VPN2 and PageMask) of one process (via ASID) of one machine (via MID), benefiting both homogeneous and heterogeneous virtualization. Figure 7 exhibits the 32-bit $< MID, ASID, Address >$ address space.

In Loongson BTS, we use MID `0`, `2` and `3`. MID `0`(bound to HTLB) is used by MIPS kernel and legacy applications by default. MID `2` works as x86 privilege state while MID `3` as x86 non-privilege state. They are bound to GTLB. As a result, address of MIPS is translated by HTLB while x86 address translation is converted by GTLB. GTLB's D bit describes the writable control of one page: 1 is writable while 0 is read-only. The V domain represents whether the GVA-HPA mapping is valid. Figure 8 presents an example of how an x86 address is translated by GTLB. By this means, the GTLB inherits nearly all features of QEMU's software TLB so that it can perfectly conduct GVA to HPA translation.

2) The Specialized Instruction Indicating the MID: After introducing GTLB equipped with MID, two problems emerge: for any memory access instruction, the processor is required to determine which machine it belongs to; for TLB management operation such as refilling and flushing, the processor has to decide which TLB should be processed. A prefix instruction `SETMID` is implemented to deal with the first problem, and a register is modified to fix the latter one.

`SETMID` indicates the MID of next instruction, making the extended virtual address a complete one. The format of the prefix instruction is shown in Figure 9a. By using `SETMID + MIPS MARI`, we can emulate various types of memory access instruction. One example of increasing an x86 variable by one in user state is shown in Figure 9b. [20, 22, 23]

Index, Random register and HTLB bit of Diagnostic register are used to maintain HTLB. When modifying a TLB entry by using `tlbwi` (`tlbwr`), Index (Random) is used to select the desired entry. In this situation, the value of the Index is

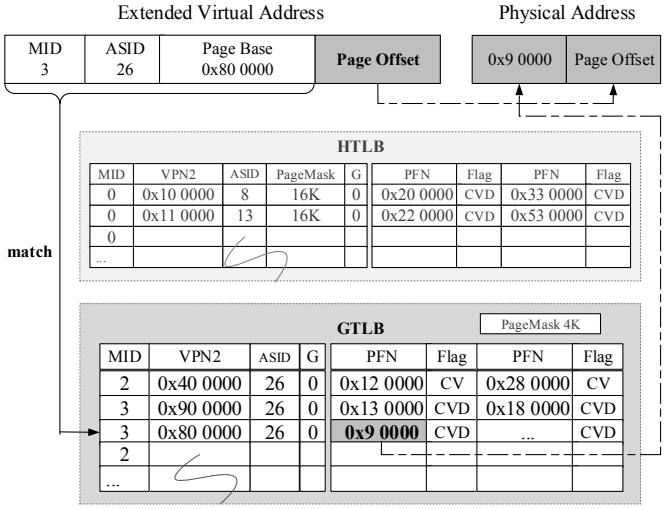

Fig. 8. Address Translation by Using Machine Isolation Technique

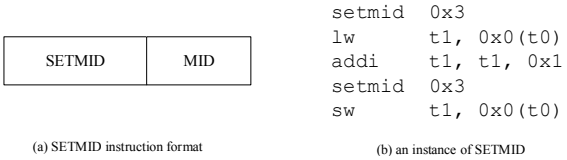

Fig. 9. The Format of SETMID Instruction

specified by software, while Random's value is controlled by the hardware randomizing. With the additional 1024 GTLB entries, Index and Random are hereby extended from 0-63 to 0-1087, and select HTLB (GTLB) when they are between 0 (64) and 63 (1087). The HTLB bit of Diagnostic is used to flush the entire HTLB when software attempts to write 1 to it. GTLB bit is implemented on a reserved bit of Diagnostic to control the GTLB flushing operation. If it is configured by 1, the entire GTLB will be flushed. [20, 22, 23]

With the implementation of the `SETMID` and the GTLB bit of Diagnostic, the original HTLB management software can be correctly executed without any modification, and the GTLB management software can refer to HTLB's design. Based on the work above, we can use and manage the dual-TLB of Loongson GS464e core at a low cost of both hardware and software.

B. Eliminate the Overhead of QEMU's Memory Virtualization

The machine isolation technique built in Loongson GS464e core, described in last Subsection, makes conducting GVA-HPA translation directly possible. Because the job of QEMU's memory virtualization is translating GVA into HVA, the traditional algorithm of traversing QEMU's software TLB is no longer required. The optimized algorithm simply indicates the MID and GVA of a memory access operation when translating MARI from x86 to MIPS. For example, a conventional x86 memory access instruction will be translated it into the instruction pair `SETMID + MIPS MARI` instead of original ten more instructions (Figure 4). The instruction pair uses the x86 virtual address directly, thus eliminating the overhead of QEMU's memory virtualization. An instance of x86 memory access instruction in non-privileged state (MID `3`) and its translated instructions are shown in Figure 10. For comparison, Figure 4 is the instance using the original mechanism. Obviously, by reforming the QEMU's software TLB algorithm, 80% size of translated code is cut down.

X86 code

```
mov     0x0(%esp),%edx
```

MIPS code

```
# t0: GVA
setmid  0x3
lw      t1,0(t0)
```

Fig. 10. The Instance of SETMID Instruction

C. The Handler of the Virtual Machine Exit

While the translated code can be greatly reduced by utilizing the machine isolation technique, the trigger of virtual machine exit changes. As described in Subsection II-C, in previous Loongson BTS, the virtual machine exit triggered by software TLB miss is handled by QEMU itself. While for the Co-Design system equipped with machine isolation technique, virtual machine exists when the address of `SETMID + MIPS MARI` cannot match any GTLB entry.

For traditional MIPS architecture, TLB exception is divided into three types: TLB miss, TLB invalid and TLB modify. TLB miss means no entry matching the input address exists in TLB, and is the most frequently happened one. TLB invalid occurs when a matched entry exists but its V bit is 0. TLB modify is trapped if software attempts to write data to a read-only page. Similar to traditional MIPS TLB exception, the GTLB of Loongson GS464e core has three exceptions: GTLB miss, GTLB invalid, GTLB modify. There are certain difficulties in handling these exceptions. For the reason that when modifying GTLB, the software should meet at least two requirements: the mapping of GVA-HPA, which are used to update GTLB; the authority to update GTLB, which means the software should run in privileged mode. As we known that Linux kernel is allowed to update GTLB, but does not have no HPA. On the contrary, non-privileged program QEMU does not the permission to modify GTLB. Obviously, handling the GTLB exceptions requires the cooperation of kernel and QEMU.

To deal with this problem, we design a software mechanism which contains three parts: A customized MIPS kernel with GTLB exception handler, QEMU with signal handler and GTLB maintainer, and a kernel module KQ used to help QEMU to modify GTLB. Figure 11 is an example of how the GTLB miss exception is handled, and the specific operations are described as below. A pair of translated instructions of x86 MARI is being executed, whose address cannot match any GTLB entry, and thus GTLB miss is triggered. The GTLB miss exception handler is activated by hardware automatically. It saves the runtime context of virtualized x86, and then injects the context to QEMU by sending SIGILL signal. QEMU's signal handler takes control, and analyzes the context that kernel passed over. The handler gets the PC (Program Pointer),

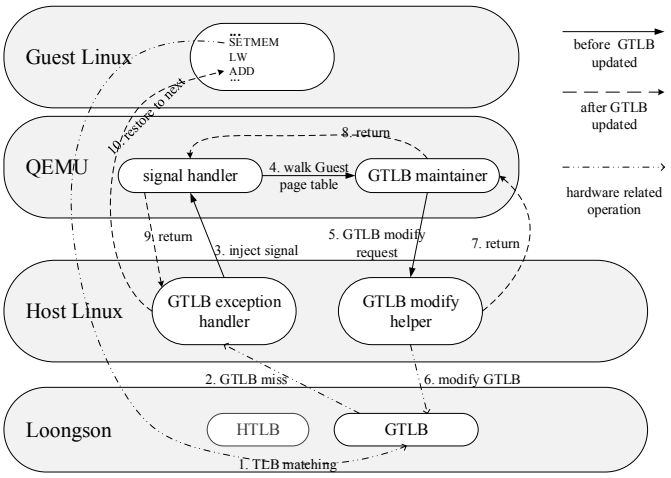

Fig. 11. The Handling System of GTLB Miss Exception

MID, GVA and memory access type of the instruction pair (`SETMID+LW`) which causes GTLB miss exception. Then it calls the relevant function to search x86 page table to get GPA, converts GPA into HVA, and emulates the memory access. After that, the maintainer transmits data, including GVA, HVA, MID and read/write permission, to KQ via the `proc` file system. KQ, the GTLB modify helper inserted into the customized MIPS kernel dynamically, then walks MIPS page table and fills a GTLB entry as QEMU's request. Up to now, the GTLB miss exception processing ends and the control flow starts to return. The calling path returns step by step to QEMU's signal handler. Right on this spot, the PC of x86 virtual machine is pointed at the instruction right after the instruction pair, and then returns to MIPS kernel. MIPS kernel restores the modified runtime context. The emulation of x86 continues.

GTLB modify exception has similar procedure, while GTLB invalid exception will never occur. The reason is that GTLB miss exception handler has already emulated the memory access instruction, which means that each GTLB entry's corresponding MIPS physical page has already been allocated. With the software mechanism presented in last paragraph, the GTLB exception can be handled perfectly, ensuring the correct execution of the translated code of guest's MARI.

IV. Performance Evaluation

This section evaluates the performance of the Co-Design Loongson BTS by simulating the system with different configurations on EVE platform. EVE is a speedy hardware simulation system developed by Synopsys[24], which can help to verify hardware/software system before taping out.

A. The Configurations of Evaluation

On EVE, for comparison, we run benchmark with two configurations of Loongson BTS: *Origin* and *Optimize*. The hardware of both configurations is Loongson 3A1500 processor with four GS464e cores built in. GS464e core is upgraded from GS464 core[25] which is also developed by Loongson Technology. The upgrade is mainly focused on five aspects: improving the performance of memory access and the accuracy of branch prediction, enlarging the number of items in the queue, increasing the capacity of Cache and TLB, implementing the MIPS DSP instruction set, and embedding hardware support for both homogeneous and heterogeneous virtualization[16]. While these four cores of Loongson 3A1500 processor are labelled as 0-3, we only boot core 0, ignoring the multiple cores issues. For software, the *Origin* configuration is running original QEMU on conventional MIPS Linux kernel, where the hardware extensions are not enabled. On the other side, the *Optimize* configuration is executing optimized QEMU on the customized MIPS Linux kernel. This is the system we proposed in Section III which utilizes the hardware extensions. In the end, the hit rate of GTLB is evaluated.

For each evaluating configuration, there are four steps: boot MIPS Linux kernel 2.6 on EVE simulation platform, run QEMU on MIPS Linux kernel, emulate x86 Linux kernel 2.6 with QEMU and run x86 memcpy (memory copy) on x86 Linux to get test results. Since the optimization is transparent to guest, x86 Linux kernel and x86 applications can run without any modification. By running memcpy we evaluate the peak performance, while by booting x86 Linux kernel we evaluate the average performance.

B. Evaluating Peak Performance with Memcpy

We implement a benchmark using the memcpy function of C library to copy a certain size of contiguous memory, thus assessing the result of the optimization for memory intensive program. The working size of each test varies from 100KB to 400MB, running for different times, shown in Figure 12. For example, `256KB*100` represents the test of copying a 256KB array for 100 times, whose working size is 25MB.

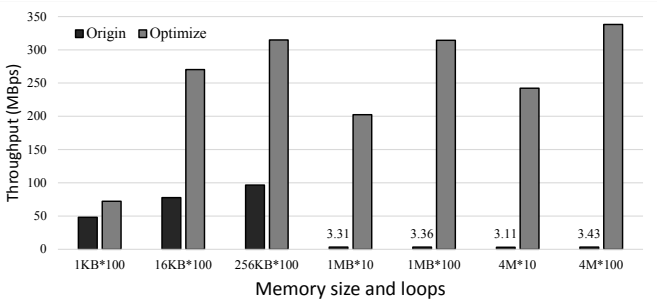

Fig. 12. Throughput of Memory Copy of Different Size

Obviously, the optimization results are outstanding in a variety of sizes and show the trend that the more intensive memory access leads to faster applications. Figure 12 indicates that the peak throughput of *Optimize* is 302.37 MBps while the *Origin* is only 3.30 MBps at the same size. The evidence points out that the performance of intensive memory access operations has been incredibly improved by 100 times.

On the other side, the results of *Origin* are not stable when working size is small (e.g. `1KB*100, 16KB*100, 256KB*100`) due to the random error of our test platform. Luckily, as the working size increasing, the error is decreasing.

C. Evaluating Average Performance by Booting x86 Linux Kernel

Since Linux is widely used in industry and easy to customize to support new hardware features, we evaluate optimization on practical workload by booting x86 Linux kernel on it.

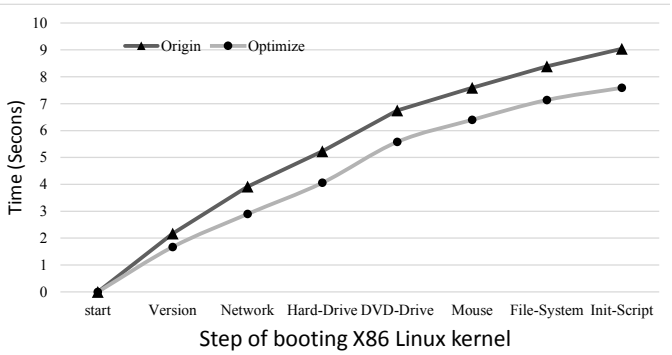

Fig. 13. Time Used to Boot x86 Linux kernel

We evaluate the time it takes to boot x86 Linux kernel 2.6 by different configurations. As the results shown in Figure 13, *Optimize* boots 19.12% faster than *Origin*. The booting has been divided into several steps, the incremental performance of each step is shown in Figure 14. It's obvious that the performance of I/O intensive steps (e.g. detect DVD drive, mouse and file system) has not been remarkably improved. The reason for this is that despite MMIO (Memory Mapped I/O) operations are translated as MARI, but the GTLB maintainer (described in Subsection III-C) will never update GTLB for these operations, which means each MMIO operation will be trapped. What's more, I/O devices are totally emulated by QEMU, thus leading to a drawback of performance.

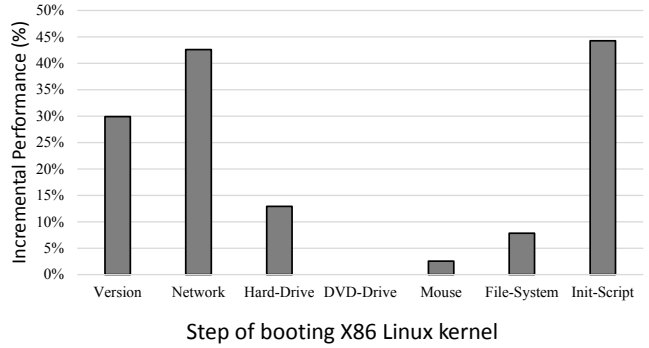

Fig. 14. Increment of Booting x86 Linux kernel

D. Evaluating GTLB Hit Rate

When booting x86 Linux kernel with *Optimize* configuration, we monitor the behavior of GTLB. The GTLB hit rate of each step of booting procedure is shown in Figure 15. Obviously, GTLB has a high hit rate, and the average is 99.86%, illustrating that the Co-Design system is effective and efficient. This is partly because the GTLB has been filled when a mapped x86 physical page is allocated.

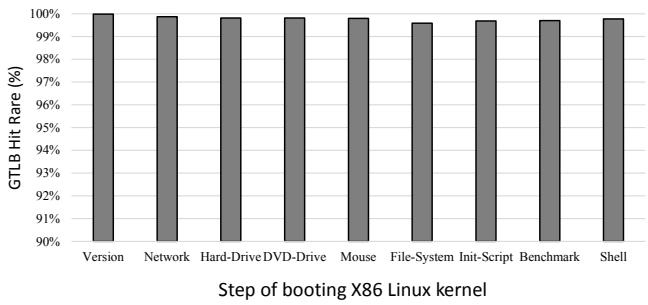

Fig. 15. GTLB Hit Rate When Booting x86 Linux kernel

V. Conclusion and Future Works

In this paper, we describe the design and implementation of machine isolation technique to improve memory virtualization. The technique, including a specialized TLB and instruction, is built in Loongson GS464e core, on which the next generation of Loongson BTS is based. This optimization completely eliminates the overhead of memory virtualization of QEMU. With this technique, the average performance of x86 emulated on Loongson 3A1500 has been improved by 19.12%. And, for memory-intensive applications, the system is dramatically 100 times faster than before. The flexible and configurable solution proposed in this paper is universally applicable for heterogeneous virtualization.

The experimental results of Subsection IV-D shows the fact that after eliminating the expense of memory virtualization, the I/O virtualization turns into bottleneck. In the future, Co-Design solutions focusing on improving I/O virtualization (e.g. graphics, storage) will constantly spring up.

References

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 2–13, 2006.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 191.

[3] J. Wang, "Co-design and co-optimization of x86 virtual machine on general risc platform," Ph.D. dissertation, Graduate University of Chinese Academy of Sciences, 2008.

[4] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2003, pp. 15–24.

[5] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communications of the ACM*, vol. 36, no. 2, pp. 69–81, 1993.

[6] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[7] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[8] Q. Liu, "The design, implementation and optimization of binary compatible system," Ph.D. dissertation, Graduate University of Chinese Academy of Sciences, 2010.

[9] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 476–487.

[10] Y. Tai, W. Cai, Q. Liu, G. Zhang, and W. Wang, "Comparisons of memory virtualization solutions for architectures with software-managed tlbs," in *Networking, Architecture and Storage (NAS), 2013 IEEE Eighth International Conference on*. IEEE, 2013, pp. 125–130.

[11] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[12] A. M. Devices, *AMD-V Nested Paging*, 1st ed., 2008.

[13] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, and Y. Zhang, "Improving virtualization in the presence of software managed translation lookaside buffers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 120–129.

[14] A. Klaiber *et al.*, "The technology behind crusoe processors," *Transmeta Technical Brief*, 2000.

[15] K. Ebcioğlu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 26–37.

[16] R. WU, W. WANG, H. WANG, and W. HU, "Loongson gs464e processor ip architecture," Oct. 2014, under reviewing.

[17] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[18] D. Sweetman, *See MIPS run*. Morgan Kaufmann, 2010.

[19] MIPS, *MIPS Architecture For Programmers Volume III: MIPS64 / microMIPS64 Privileged Resource Architecture*, 6th ed., MIPS Technologies Inc.,, March 2014.

[20] Loongson, *Loongson 3A1500 User Manual 2nd HalfProcessor*, 0th ed., Loongson, Technology, May 2014.

[21] Y. Tai, "The design and implementation of hybrid virtualization system on mips," Ph.D. dissertation, University of Chinese Academy of Sciences, 2014.

[22] Loongson, *LoongsonISA Instruction Set Manual Volume I: Instruction List*, 1st ed., Loongson, Technology, March 2014.

[23] Loongson, *LoongsonISA Instruction Set Manual Volume II-a: Customized General Purpose Extended Instruction*, 1st ed., Loongson, Technology, March 2014.

[24] Synopsys, "Industrys fastest emulation system," http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/default.aspx, 2014, [Online; accessed 19-May-2014].

[25] W.-W. Hu, F.-X. Zhang, and Z.-S. Li, "Microarchitecture of the godson-2 processor," *Journal of Computer Science and Technology*, vol. 20, no. 2, pp. 243–249, 2005.