ARMvisor: System Virtualization for ARM

Jiun-Hung Ding
National Tsing Hua University
adjunhon@sslab.cs.nthu.edu.tw

Chang-Jung Lin
National Tsing Hua University
chucklin@sslab.cs.nthu.edu.tw

Ping-Hao Chang
National Tsing Hua University
phchang@sslab.cs.nthu.edu.tw

Chieh-Hao Tsang
National Tsing Hua University
jeffzaza@sslab.cs.nthu.edu.tw

Wei-Chung Hsu
National Chiao Tung University
hsu@cs.nctu.edu.tw

Yeh-Ching Chung
National Tsing Hua University
ychung@cs.nthu.edu.tw

Abstract

In recent years, system virtualization technology has gradually shifted its focus from data centers to embedded systems for enhancing security, simplifying the process of application porting as well as increasing system robustness and reliability. In traditional servers, which are mostly based on x86 or PowerPC processors, Kernel-based Virtual Machine (KVM) is a commonly adopted virtual machine monitor. However, there are no such KVM implementations available for the ARM architecture which dominates modern embedded systems. In order to understand the challenges of system virtualization for embedded systems, we have implemented a hypervisor, called ARMvisor, which is based on KVM for the ARM architecture.

In a typical hypervisor, there are three major components: CPU virtualization, memory virtualization, and I/O virtualization. For CPU virtualization, ARMvisor uses traditional "trap and emulate" to deal with sensitive instructions. Since there is no hardware support for virtualization in ARM architecture V6 and earlier, we have to patch the guest OS to force critical instructions to trap. For memory virtualization, the functionality of the MMU, which translates a guest virtual address to host physical address, is emulated. In ARMvisor, a shadow page table is dynamically allocated to avoid the inefficiency and inflexibility of static allocation for the guest OSes. In addition, ARMvisor uses R-Map to take care of protecting the memory space of the guest OS. For I/O virtualization, ARMvisor relies on QEMU to emulate I/O devices. We have implemented KVM on ARM-based Linux kernel for all three components in ARMvisor. At this time, we can successfully run a guest Ubuntu system on an Ubuntu host OS with ARMvisor on the ARM-based TI BeagleBoard.

1 Introduction

Virtualization has been a hot topic and is widely employed in data centers and server farms for enterprise usage. Today's mobile devices are equipped with GHz CPU, gigabytes of memory and high-speed network. With the advancement of computing power and Internet connection in embedded devices, system virtualization also assists to address security challenges, and reduces software development cost for the mobile and embedded space. For instance, the technique of consolidation is capable of running multiple operating systems concurrently on a single computer and the technique of sandboxing enhances security to prevent a secure system from destruction by an un-trusted system. The benefits and the new opportunities have prompted several companies to put virtualization into mobile and embedded devices. Open Kernel Labs announced that the OKL4 embedded hypervisor has been deployed on more than 1.1 billion mobile phones worldwide to date. Hence, research into the internal design and implementation of embedded hypervisors also attact more attention in academic communities. In this paper, we have worked on the construction of ARM based hypervisor without any hardware virtualization support.

Essentially, the ARM architecture was not initially designed with system virtualization support. In the latest variant, the ARMv7-A extension which was an-

nounced in the middle of 2010, ARM will start to support hardware virtualization and allow up to 40-bit physical address space. However, the widely used variants of ARM processors, such as ARMv5, ARMv6 and ARMv7, lack any hardware extension for virtualization, making it difficult to design an efficient hypervisor when well-known full virtualization method is being applied. In fact, according to the virtualization requirements proposed by Popek and Goldberg in 1974 [2], ARM is not considered as a virtualizable architecture [1]. There exist numerous non-privileged sensitive instructions which behave unpredictably when guest operating system is running in a de-privileged mode. These critical instructions increase the complexity of full virtualization implementation, and some of them, such as LDRT/STRT/LDRBT/STRBT instructions, will cause huge performance degradations. Traditionally, techniques like trap-and-emulation and dynamic binary translation (DBT) are adopted to handle the sensitive instructions as well as critical instructions for CPU virtualization. In ARMvisor, a lightweight paravirtualization method takes the place of DBT. Merely hundreds of codes are necessarily patched into guest operating system source codes.

For memory virtualization, the shadow paging mechanism is also hard to be manipulated. First, the address translations, access permissions and access attributes described in guest page tables and other system registers must be correctly emulated by the underlying hypervisor. Second, the hypervisor must maintain coherences between guest page tables and the shadow ones. Third, the hypervisor needs to protect itself from destruction by guest, as well as to forbid user mode guest to access kernel memory pages. According to the above three requirements, the ARM hypervisor still suffers performance degradation, especially during the sequence of process creation. To reduce the overhead, a lightweight memory trace mechanism for keeping shadow page tables synchronized with the page tables of the guest OS is proposed by ARMvisor.

The experimental results of ARMvisor have shown leaping performance improvement with up to 6.63 times speedup in average on LMBench. Additionally, in embedded benchmark suite such as MiBench, the virtualization overhead is minimal, meaning that performance is fairly close to native execution.

The rest of the paper will be organized as follows. Related works will be discussed in Section 2. Software architecture of our embedded hypervisor is presented in Section 3. A cost model for hypervisor are formed in Section 4. Section 5 describes the optimization methodologies for CPU and memory virtualization. Experimental results and analysis are covered in Section 6. At last, we conclude the paper in Section 7.

2 Related work

A variety of virtualization platforms for ARM have been developed in the past few years as a result of the long leap in computing power of ARM processor. Past work [3] mentioned several benefits of virtualizing ARM architecture in embedded systems. It was noted that the hypervisor provides the solution to embedded system including security issues in online embedded devices due to the downloading of third party applications by isolating hardware resource by virtual machines. An embedded hypervisor also enables heterogeneous operating system platform to deal with conflicting of API in different operating systems.

Others [1] have analyzed the requirement for designing a hypervisor for embedded system. For instance, the hypervisor must be equipped with scheduling mechanisms for latency critical drivers to meet the real-time requirements, and must also support various peripheral assignment policies such as directly assigned, shared assigned devices or run-time peripheral assignment. Other important factors include accelerating the boot time of the guest OS, utilizing the utmost performance of embedded hardware as well as reducing the code size of hypervisor to prevent from potential threat.

Numerous hypervisors have been designed for newer capabilities for embedded ARM platforms. OKLab has developed OKL4 Microvisor [4] based on L4 microkernel for ARM, catering to the merits of embedded virtualization. They claim that the hypervisor has been ported to millions of mobile handsets and supports system such as Windows and Android to run atop of OKL4 Microvisor. They claim that the OKL4 Microvisor supports a secure embedded platform. Other commercial VMMs include VMware's MVP [5], Trango [6] and VirtuaLogix [7]. Nevertheless, none of these solutions, including the OKL4 Microvisor, are open-sourced and thus the insights of their design are not available.

On the other hand, Xen [8] is a well-known hypervisor for system virtualization, and has been successfully

ported to ARM architecture in Xen version 3.02 [9]. "Xen for ARM" required the guest code to be para-virtualized by adding hyper-calls for system events such as page table modification. However, code that needs to be para-virtualized is not revealed in the paper. The cost of maintenance with generations of different guest operating system soars higher as heavier as the guest's code being modified for virtualization.

"KVM for ARM" [1] also implemented an embedded VMM under KVM in ARMv5. They proposed a lightweight script-based approach to para-virtualize the kernel source code of the guest OS automatically, by switching various kinds of non-privileged sensitive instructions with pre-encoded hyper-calls that trap to hypervisor for later emulation. Nonetheless, they applied a costly memory virtualization model when deprivileging guest system in the user mode of ARM architecture. First, they did not apply the "reverse map" mechanism in memory virtualization for keeping the coherency of guest's and shadow page table. In their model, each modification results in a page table flush since the hypervisor is unaware of the correspondence of guest's page table and shadow page table. Furthermore, the benchmarking or profiling results are not yet revealed, so it is hard to evaluate the performance results of running virtual machines on their work.

In contrast, ARMvisor introduces a lightweight memory virtualization model mechanism to synchronize the guest page table, which is more suitable for use in embedded system due to the performance and power consumption concern. Detailed measurement and analysis of system and application level benchmarks will be reported in the following section. We proposed a cost model to measure overhead due to virtualization in general use cases. According to the profiling results, we can design several optimization methodologies.

3 Overview of ARMvisor

The proposed hypervisor is developed based on the open source project KVM (Kernel-based Virtual Machine) which was originally designed for hardware virtualization extension of x86 architecture (Intel VT, AMD SVM) to support full-virtualization on Linux kernel. It has been included in the mainline Linux since kernel version 2.6.20. KVM is composed numerous loadable kernel modules which provide the core functions of the virtualization. A modified QEMU is used to create the

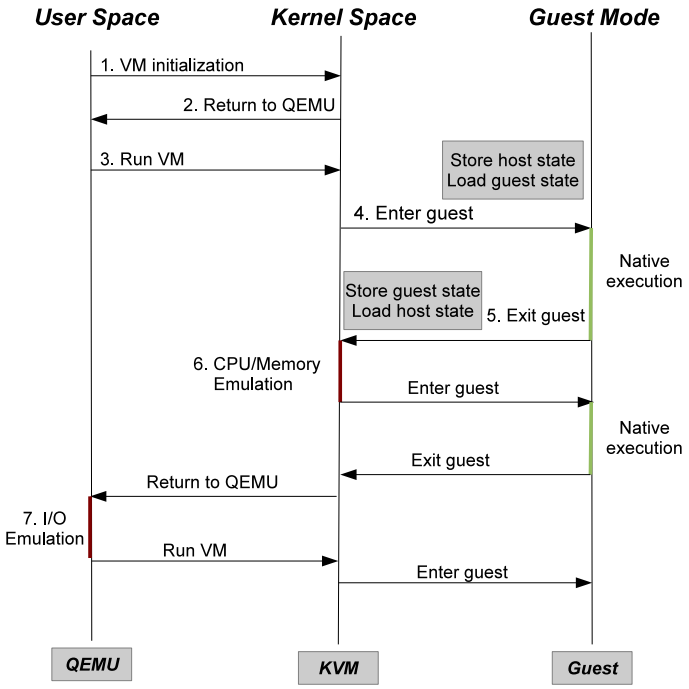

Figure 1: KVM execution path

guest virtual machine and to emulate the I/O devices. Figure 1 illustrates the execution flow when providing a virtualization environment for the guest using KVM. A virtual machine (VM) is initiated in QEMU by using the system call interface (`ioctl`) provided by the modules of KVM. After the VM has finished its initialization procedures, KVM changes the execution context to emulated state of the guest OS and starts to execute natively. Guest exits its execution context whenever an exception occurs. There are two kinds of traps in KVM: *lightweight* and *heavyweight* traps. In lightweight traps, the emulation is handled in internal functions of KVM, implemented in the kernel space of Linux. In contrast, heavyweight traps that include I/O accesses and certain CPU functions will be handled by QEMU in user space, so context switches are required. Hence, the cost of a single heavyweight trap is higher than a lightweight one.

Numerous hardware virtualization extension primitives of the x86 architecture are leveraged by KVM to provide fast and stable workloads for virtual machines. A new execution *guest mode* is designed to allow direct execution of non-privileged yet sensitive x86 instructions. Instead of unconditionally trapping on every privilege or sensitive guest instruction, a subset of those instructions can execute natively in the virtual ring 0 provided by the guest mode. They modify the shadowed CPU states indicated on *VMCB* (Virtual Machine Control Block)

to emulate the desired effect as is natively executed. VMCB is an in-memory hardware structure which contains privileged registers of the guest OS context and the control state for VMM. VMM can fetch guest's running status directly as well as setup the behavior of exceptions from the VMCB. Furthermore, functions are defined for the switch between guest and host mode. For instance, after VMM finishes its emulation tasks like page table filling or emulating I/O instruction, `vmrun` is called to reload the state of virtual machine from the modified VMCB to physical registers and resume its execution in guest mode. Aside from the hardware assistance for CPU virtualization, specific features were proposed to aid MMU virtualization on x86. Intel's *EPT* and AMD's *nested paging* were designed to tackle the issue of "shadow paging" mechanism traditionally applied in MMU virtualization.

However, due to the lack of hardware assistance in our experimental ARM architecture, KVM's performance suffers in various aspects on ARM. In CPU virtualization, software techniques are adopted due to the lack of hardware support, such as the x86 *guest mode* for virtualization. Lightweight traps are generated by adding hyper-calls for non-privileged sensitive instruction emulation. To apply the trap and emulation model for emulating instruction, guest system is de-privileged to execute in user mode. Besides, without the help of VMCB, a context switch interface is needed for KVM/guest switch. The state save/recovery for both lightweight and heavyweight traps results in extra overhead and thus largely degrades the system performance. Details of the methodologies we took and the optimization applied will be illustrated in the following sections.

3.1 CPU Virtualization

Guest Operating Systems finish critical tasks that access systems resources by executing sensitive instructions. According to the definition [2], there are two categories of sensitive instructions: Control Sensitive and Behavior Sensitive instructions. Control Sensitive instructions are those that attempt to change the amount of resource and the configuration available, while Behavior Sensitive instructions are those that behave depending on the configuration of resources. For example, "CPS" in ARM modifies CPU's status register, a.k.a. CPSR, to change the execution mode or to enable/disable interrupt and has control sensitivity. Moreover, executing CPS in user mode results to NOP effect thus it is also behavior sensitive. Such instructions must be handled properly to keep the guest being correctly executed.

Actually, in order to prevent guest from ruining the system by controlling hardware resource with privilege and allow hypervisor to manage the resource of system, guest operating system is de-privileged to execute in non-privilege mode while hypervisor is located in privilege level for resource management. Pure virtualization that executes guest OS directly in user mode is proposed under the premise that the architecture is virtualizable, i.e. all the sensitive instructions trap when executing in non-privilege mode. Hypervisor intercepts such traps and emulates the instruction in various fashion. The approach requires no modification of guest OS to run in a virtual machine, so the engineering cost for porting and maintaining multiple versions of guest OS is minimal. However, pure virtualization is not feasible in contemporary architectures since most of them including x86 and ARM are not virtualizable and there exist some non-privilege sensitive instructions, called *critical instructions*, which behave abnormally when being deprivileged in user mode.

Solutions have been proposed to ensure the exactness of system execution in non-virtualizable architecture like x86 without hardware extension. Past work [13] imports dynamic binary translation (DBT) techniques to overcome the obstacles in virtualizing x86 architecture to support full virtualization. Guest binary is transformed into numerous compiled code fragments (CCF) and chained together by the VMM. In essence, most of the guest code is identical except those sensitive instructions. Sensitive instructions are either translated into non-sensitive ones, or into jumps to a callout function for correct handling. Nonetheless, DBT seems prohibitive in embedded system since the translated code accounts for large portion of memory and RAM size is comparatively smaller than in large server or work station.

Besides, paravirtualization [8] methodology that replaces the non-privilege sensitive instructions with sets of pre-defined hyper-calls for x86 is also introduced. During execution, the hyper-call traps to the hypervisor for corresponding handling for events such as page table pointer modification. In spite that the performance presents promising run-time results, engineering cost is expensive for porting a guest OS to the virtual machine in Xen and thus adds difficulties for maintain-

ing new versions of guest OS distribution for performance issues. To solve such limitations of paravirtualization, new technique called pre-virtualiztion [14] has been proposed. They presented a semi-automatic sensitive instruction re-writing mechanism in the assembler stage, and assert that compared with paravirtualization, the approach does not only require orders of magnitude fewer modification of guest OS, but also achieves almost the same performance result as paravirtualization.

Given that ARM's ISA is non-virtualizable, ARMvisor chooses paravirtualization techniques to handle guest's non-privilege sensitive instruction. Guest kernel and applications are de-privileged to execute in ARM's user mode, while ARMvisor executes in ARM's supervisor mode to avoid guest from crashing the host system. ARM's SWI, accompanied with a dedicated number, is inserted manually before each sensitive instruction as hyper-calls for instruction emulation. Traps will be triggered and sent to the Dispatcher in the hypervisor. ARMvisor acknowledges the software interrupt triggered with a specific number and then decodes and emulates the sensitive instructions by effectively modifying the "Virtual Register File", which represents the virtual CPU context of guest system.

In practice, trapping on each sensitive results in huge degradation in performance, due to the high cost of traps in modern computer design. As stated earlier, hardware extension of x86 architecture provides extra mode for virtualization to address such issue. Many sensitive instructions can directly execute in guest mode rather than trapping to hypervisor for later handling thus improving performance. Vowing to lower the considerable overhead results from "trap and emulation" without hardware extension in contemporary ARM architecture, we further proposed two optimizing technique to accelerate the performance. Insights of each optimizing heuristic will be discussed in later section.

3.1.1 Instruction emulation

ARMv6 defines 31 sensitive instructions, of which 7 are privileged instructions and will trap when guest system is being de-privileged. The rest of them are critical instructions, which required code patching to prevent non-deterministic behaviour in user mode. Table 1 lists the counts of various types of critical instructions that need to be modified in Linux kernel 2.6.32 to boot

Instruction_type	Count
Data Processing (movs)	4
Status Register Access (msr/mrs, cps)	34
Load/Store multiple (ldm(2,3), stm(2,3))	8

Table 1: Sensitive instruction count

Cache/TLB's invalidation and clean
BTB Flush
TTBR Modification
Domain configuration
Context ID
Processor status (ex: CPU ID, Page Fault Info)

Table 2: ARM co-processor operations

on ARMvisor. We figured that the critical instructions exist in files for three separate purposes in Linux kernel: kernel boot-up, exception handling and interrupt enable/disable. Macros composed of instructions that setup ARM Status Register for interrupt controlling are defined in header files, and are widely deployed in large portion of kernel code. Moreover, numerous critical instructions actually account for large portion of the code in ARM Linux's exception handling entry and return. In our measurement, we found that merely forcing those non-privilege sensitive instructions to trap for instruction emulation in ARMvisor brings about intolerable performance loss in guest system. In fact, these instructions only involve virtual state change and no hardware modification is needed. Consequently, we eliminate the traps by replacing the instructions with Shadow Register File Access (SRFA) in guest's address space.

In contrast to the critical instructions, a few sensitive instructions like ARM co-processor's operation (MCR/MRC) are privileged, and numerous essential operations are accomplished through the execution of those privilege instructions. Table 2 lists several functions achieved by co-processor operations. Traps will be invoked when executing these instructions in ARM's user space and certain hardware alteration is performed by ARMvisor to correctly emulate guest's desired behavior.

To relieve the overhead suffered in emulating those instructions, we concisely analyze all of the operations and propos several refinement methodologies to achieve performance improvement. First, guest's TLB operations and BTB flush is dynamically replaced with NOP's by ARMvisor during execution, since TLB and BTB

will be thoroughly flushed during every context switche between guest and the hypervisor. Secondly, operations that read information in co-processors, such as memory abort status and cache type, can also be replaced with SRFA since only virtual state is fetched. Finally, cache operations involving certain hardware configuration must be trapped and finished in privilege level. To minimize the overhead of emulation, technique called *Fast Instruction Trap* (FIT) is applied to reduce costs of context switching and handle those operations in guest address space. Implementation details of SRFA and FIT will be narrated in later section. TTBR and Domain Register modification is comparatively complicated so they are emulated in ARMvisor through lightweight traps.

We conclude that many software optimizations can be applied with paravirtualization techniques to effectively mitigate the run-time overhead for virtualization at the expense of higher engineering cost. It depends on the VM distributor to decide the pros and cons of performance elevation with the trade-off of the cost to maintain versions of guest OS.

3.1.2 CPU virtualization model

The hypervisor gathers system exceptions such as interrupts, page faults and system calls, and properly handles them for each virtual machine. Hardware extensions in the x86 architecture enable guest OS exceptions to be handled by its operating system natively without the interference of hypervisor by setting typical bits in VMCB.

In the ARM architecture, which lacks such hardware assistance, the hypervisor is responsible for distinguishing and virtualizing exceptions of the guest OS by delivering traps to each virtual machine. Synchronous exceptions such as memory access abort, system calls as well as undefined access have higher priority and should be injected to guest virtual machine immediately. Asynchronous exceptions such as interrupt could be delivered later when the pending queue for synchronous exceptions of that virtual machine is empty.

Additionally, exceptions may be invoked for virtualization events such as hyper-calls for instruction emulation and extra shadow page table miss, which are non-existent when executing the OS on bare hardware.

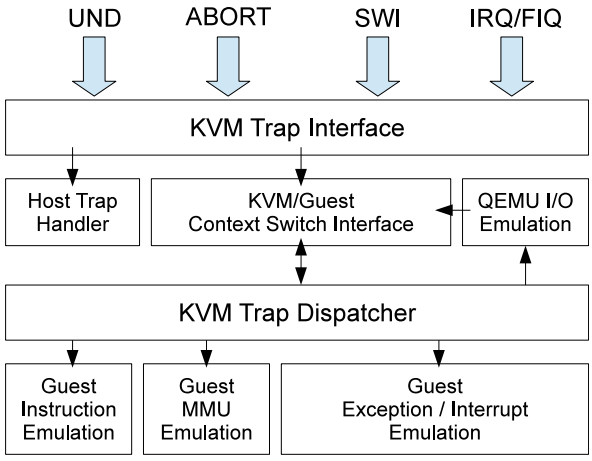

Figure 2: VCPU virtualization flow

These traps are handled internally by the hypervisor and the guest is actually unaware of such events.

As depicted in Figure 2, exceptions are routed to ARMvisor for unified administration by replacing host kernel's exception vector with *KVM Vector* when KVM module is loaded. Therefore, system's exceptions are re-directed to the Trap Interface inside ARMvisor for later handling. The interface verifies the host and guest exceptions and branch for separate handling path. Whenever guest exceptions are discovered, KVM/Guest Switch Interface saves the guest interrupt state and switches to KVM's context for later handling. In fact, since KVM executes in a different address space from the guest, page tables must be changed for consequent trap handling. However, unlike the hardware extension of x86 architecture that restores the page table pointer register automatically in VMCB for exception exits, KVM/Guest Switch Interface must be contained in a shared page which is accessible in both guest and KVM address space, otherwise the behavior would be undefined after the modification of TTBR (Translation Table Base Register) in ARM. As shown in Figure 3, the interface is contained in a page between the address of `0xffff0000` to `0xffff1000` (if high vectors are used). This page is write-protected to prevent from malicious attacks crashing the system.

The trap dispatcher in ARMvisor forwards lightweight and heavyweight traps to different emulation blocks. Several traps, such as hyper-calls, memory access abort and interrupt, are handled by emulation blocks in ARMvisor respectively as illustrated in Figure 2. After all KVM's internal emulation blocks finish their emula-

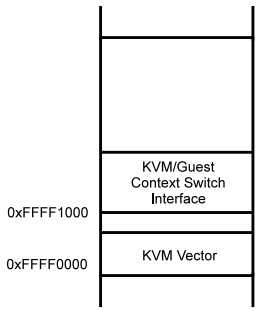

Figure 3: KVM vector implementation in ARMvisor

tion, KVM/Guest Switch Interface verifies if the trap is lightweight, and if so, restores the guest interrupted context to continue its previous execution. Otherwise, the trap is heavyweight, so a context switch is required since subsequent emulation tasks are to be finished in QEMU. After a heavyweight trap finishes its emulation, such as I/O access and CPU related operations through the interfaces provided by QEMU, another context switch is taken to resume the guest's previous execution.

3.2 Memory virtualization

Generally, the VMM provides a virtualized memory system for a guest virtual machine. When the user-level applications and operating system run in a VM, their memory access will be precisely controlled and remapped to host physical memory. For security, the VMM necessarily protects itself from illegal access by any guest and isolates one VM from another one. In practice, implementing memory virtualization is relatively more complex than CPU virtualization. The efforts involve working on guest physical memory allocation, shadow paging mechanism, and virtual Memory Management Unit (vMMU) emulation. Our implementation and the considerations for ARM memory architecture will be discussed later.

3.2.1 Guest physical memory allocation

The guest physical memory can be allocated in two ways: static or dynamic allocation. A static allocation will reserve continuous memory pages from host physical memory. The allocated memory pages are occupied and unlikely to be shared with VMM or other VMs, thus the host memory resources are not being well utilized. In contrast, the dynamic allocation method

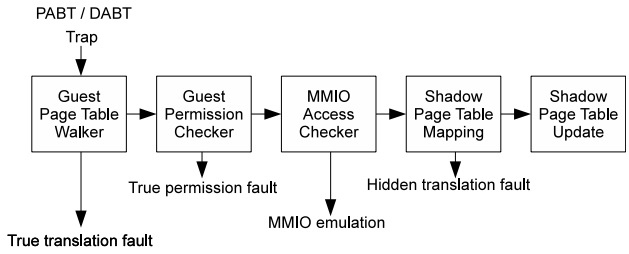

Figure 4: The emulation flow of shadow paging

maps a region of host virtual memory as guest physical memory. The host physical memory pages are allocated dynamically, as the technique of demand paging. In [10], VMware ESX server further provides a ballooning driver for guest to reclaim unused memory pages. Current KVM manages guest memory resources using the existing functionalities within the Linux kernel including buddy allocator, slab/slub allocator, virtual memory subsystem. Once a VM is created, KVM considers the guest physical memory as part of the user space memory allocated in the VM process.

While executing guest binary code, all the memory accesses will be remapped to host memory by a series of address translation processes. First, a guest virtual address (GVA) can be translated to a guest physical address (GPA) by walking through guest page tables. Then the host virtual address (HVA) is generated by the information stored in the GPA-HVA mapping table. Eventually the HVA will be translated to the host physical address (HPA) by host page tables. It has no efficiency if every guest memory access is forced to the surplus translation. The general solution of memory virtualization is to use a mechanism called shadow paging, which maintains a shadow of the VM's memory-management data structure to directly translate GVA to HPA.

3.2.2 Shadow paging

The overall design of shadow paging shown in Figure 4. When the hardware Prefetch Abort (PABT) trap or Data Abort (DABT) trap is caught by the ARMvisor, the trap will be handled following the shadow paging mechanism. The first step is to obtain the mapping information of the trapped GVA by walking through the guest page table. If the mapping does not exist, ARMvisor will deliver a true translation fault to guest. Otherwise, ARMvisor will translate the GVA to GPA and will check

whether the access permission is allowed by interpreting the guest page table entries. If the memory access is illegal, ARMvisor will inject a true permission fault. When first two steps are finished and no true guest fault is generated, the trap would be handled as a certain hidden faults such as MMIO emulation fault, hidden translation fault or hidden protection fault in the following steps. MMIO access checker will examine whether the accessed memory region is located in the guest I/O address space. If it is a MMIO access, the trap will be handled by the I/O emulation model in QEMU. Otherwise, the trap is forwarded to the last two steps with respect to shadow page table (SPT) mapping and update.

Modern ARM processors use hardware-defined page tables to map virtual address space to physical address space. The translation table held in main memory has two levels: the first-level table holds section, super section translations and pointers to second-level tables. The second-level table holds large and small page translations. Four types of mapping size are defined as supersection (16MB), section (4MB), large page (64KB) and small page (4KB). For each guest page table (GPT) of guest process, ARMvisor will allocate one SPT to map it. All of the SPTs are cached and are searched while switching guest processes. Initially, the SPT loaded into host translation table base register (TTBR) is empty when running guest. As a result, any memory access will trigger a hidden translation fault. The fault GVA is used by ARMvisor to walk through the current first-level SPT and second-level SPT to fill new entries for the translation miss. The filled entries contain the mapping to host physical memory and the permission setting in guest's page table. ARMvisor will restore the execution context of the trapped instruction afterwards, then the memory access will natively be handled by the host MMU. Generally, the maintenance of SPTs has two considerations: One is how to emulate guest's permission under de-privileged mode; the other is how to keep the coherence between GPTs and SPTs. They will be explained in the next two subsections.

3.2.3 Permission model & Synchronization model

In the ARMv6 architecture, access to a memory region is controlled by the current processor mode, access domain and access permission bits. The current processor mode will be either non-privileged mode (User mode) or privileged modes (FIQ, IRQ, Supervisor, Abort, Undefined and System modes). The type of access domain is specified by means of a domain field of the page table entry and a Domain Access Control Register (DACR). Three kinds of domain access are supported: The type of NA (No access) will generate a domain fault for any memory access. The type of Client will check access permission bits to guard any memory access. The type of Manager will not check any permission, so no permission fault can be generated. The access permission bits are encoded in the page table entries with several fields, APX, AP, and XN. Table 3 shows the encoding of the access permissions by APX and AP. The XN bit acts as an additional permission check. If set to 1, the memory region is not executable. Otherwise, if XN clear to 0, code can execute from the memory region.

APX:AP[1:0]	Privileged mode	User mode
0:00	NA	NA
0:01	RW	NA
0:10	RW	RO
0:11	RW	RW
1:01	RO	NA
1:10	RO	RO

Table 3: The encoding of permission bits

Guest OS will use the permission model to separate kernel space from user spaces. Any user process can only access its user space. Access to kernel space or other user spaces is not allowed. Additionally, guest OS may use Copy-On-Write mechanism to speed up the process creation. Hence, hypervisor is obliged to maintain the equivalent memory access permission to reflect the desired guest behavior correctly. However, fully virtualizing guest's access permission is difficult since guest is executed in de-privileged mode for security issues. For example, once the permission of a memory region is set as (RW, NA) by guest OS, it means the memory region can be arbitrarily accessed in the guest kernel mode, while the access is forbidden in the guest user mode. In such case, since guest is physically executed in ARM's non-privilege mode (user mode), the access permission is remapped to RW/RW to emulate the exact memory access behavior in virtual privilege mode. Therefore, a thorough flush of SPTs is required for any mode switch between virtual privilege and non-privilege level because the access permission set in SPT differs based on the virtual privilege level of guest. Moreover, LDRBT/LDRT/STRBT/STRT in-

GPT Privilege mode	GPT User mode	SPT Privilege mode	SPT User mode
NA	NA	NA	NA
RW	NA	RW	RW
RW	RO	RW	RW
RW	RW	RW	RW
PO	NA	RO	RO
RO	RO	RO	RO

Table 4: Access Permission bits remapping in Shadow Page Table for kernel space region

structions must be treated as sensitive instructions since they access memory with user permission while the processor is in the privileged mode. Trapping each of these instructions results to huge slowdown in program execution, since they are frequent used by operating system to copy data from/to user space. Previous work [1] actually proposed a memory access permission model using permission remapping mechanism in SPT mentioned above and suffered great performance loss due to large amount of SPT flush and sensitive instruction traps.

In view of cutting down the overhead results from permission transformation, we proposed a methodology called double shadow paging in ARMvisor. We allocate two SPTs for one corresponding GPT: kernel SPT (K-SPT) and user SPT (U-SPT). Whenever translation misses occur, each newly allocated entry which maps the kernel space region in K-SPT is translated from GPT entry by ARMvisor to emulate guest's privilege level access, while entries in U-SPT have the same permission as in GPT. Table 4 demonstrates mapping of the access permission bits in kernel space region from GPT to SPT. ARMvisor loads either K-SPT or U-SPT to ARM's TTBR (Translation Table Base Register) depending on whether the virtual CPU mode it is switching to is privileged or non-privileged.

The mechanism of double shadow paging eliminates the need for SPT flushing after virtual mode switch, as was required in the previous single-table permission remapping methodology. However, the maintenance of double PTs adds complexity to the design of the synchronization model for GPT and SPT for shadow paging mechanism in ARMvisor. Memory usage in the system will also grow since more pages are allocated as first/second level of U-SPT/K-SPT during the lifecycle of a process comparing with the single SPT model.

	GUD	GKD
Virtual User Space	Client	No access
Virtual Kernel Space	Client	Client

Table 5: Domain configuration in Shadow Page Table

To reduce memory usage in the double shadow paging model, we utilize ARM's Domain control mechanism and map one GPT to only one SPT. Permission bits of the SPT entries that map to kernel space region are translated by ARMvisor as in the double shadow paging model. However, the Domain Access Control Register (DACR) is configured by ARMvisor to reflect the same protection level on the corresponding SPT. Table 5 lists the Domain configuration of ARMvisor. GUD and GKD represent the domain bits of SPT that maps to "Guest User space" and "Guest Kernel space" respectively. In Virtual User Space, GUD is set as client and GKD is set as No access to protect the guest kernel space from invalid access by its user space. In the Virtual Kernel Space region, both GUD and GKD are set as client. DACR is modified by ARMvisor for events that involve in virtual CPU mode switch such as system call, interrupts and return to user space.

ARMvisor uses memory trace to maintain the coherence between GPT and SPT. Once a SPT is allocated to map a GPT, the memory page of this GPT is write protected. This allows ARMvisor to determine when the guest OS tries to modify the GPT entries. The synchronization model is implemented by using a RMAP data structure which records the reverse mapping from guest physical pages to SPT entries. When a guest physical page is identified as GPT, the SPT entries pointing to the page will have their write permission bits disabled. Therefore, later modification of GPT will trigger a protection fault and ARMvisor will update the SPT to prevent it from becoming inconsistent with GPT.

3.2.4 Virtual MMU emulation

Other than guest physical memory allocation and shadow paging mechanism, ARMvisor must emulate a virtual MMU for guest OS to access. The ARM processor provides coprocessor, CP15, to control the behavior of MMU. The controls include enabling/disabling MMU, reloading TTBR, resetting Domain register, accessing FSR/FAR, changing ASID as well as operating

Cache and TLB. All of them are handled in the memory virtualization.

4 Cost model

In a virtualized environment, guest applications and operating systems inevitably suffer certain degree of performance degradation, which correlates to the design of hypervisor. For hypervisor developers, it is crucial to have a cost model to assist them on analyzing performance bottlenecks before adopting optimization. In this section, we will propose a cost model to formulate hypervisor overheads so as to discover the deficiencies of hypervisor.

The cost model defines *Guest Performance Ratio* (GPR) to evaluate a hypervisor design. As shown in (1), GPR is defined as the ratio of the execution time for an application running natively on the host (T_{host}) versus the time taken in the virtual machine (T_{guest}). The range of values is $0 < GPR <= 1$. Theoretically, T_{guest} is greater than or equal to T_{host}. If GPR is close to 1, this means the guest application runs in native speed. Otherwise, if GPR is close to 0, this means the guest application suffer huge overheads from virtualization.

$$GPR = T_{host}/T_{guest} \tag{1}$$

In (2), T_{guest} is divided into two parts: T_{native} represents the instruction streams of a guest application, which can be natively executed on the host, while T_{virt} represents the virtualization overheads. In practice, the value of T_{native} s usually constant and relats to the type of guest application. Meanwhile, the value of T_{virt} is related to both guest application type and hypervisor design. For example, in matrix manipulation applications, most time is spent on non-sensitive instructions, and as a result, the value of T_{native} is close to T_{guest}. If the application has many sensitive instructions, the value of T_{virt} will be higher.

$$T_{guest} = T_{native} + T_{virt} \tag{2}$$

Developers will optimize the T_{virt} which is decomposed into five parts, as shown in 3.

$$T_{virt} = T_{cpu} + T_{mem} + T_{suspend} + T_{idle} + \varepsilon \tag{3}$$

The components are described as follows.

T_{cpu} represents the cost of emulating sensitive instructions and exception generating instructions, such as software interrupt. It can be formed as following formula:

$$T_{cpu} = \sum Cs(i) \cdot T_{inst}(i) + \sum Ce(j) \cdot T_{excpt}(j)$$

T_{mem} represents memory virtualization overheads including emulating guest memory abort, handling shadow paging, and maintaining consistency between guest page table and shadow page table. It can be formed as following formula:

$$T_{mem} = Cm(0) \cdot T_{abt} + Cm(1) \cdot T_{shadow} + Cm(2) \cdot T_{sync}$$

T_{io} represents the cost of I/O virtualization, including memory mapped I/O and port mapped I/O. Both are emulated using the device model provided by the hypervisor. It can be formed as following formula:

$$T_{io} = \sum Ca(i) \cdot T_{mmio}(i) + \sum Cb(j) \cdot T_{portio}(j)$$

$T_{suspend}$ represents time during which the guest is temporally suspended by the hypervisor. For example, when an interrupt is coming, guest is trapped into hypervisor, and then hypervisor will handle this interrupt by its ISR. Switching to other virtual machines or host thread, the running guest will be suspended for a while.

ε is used to represent as the side effect from virtualization. For instance, cache or TLB flushing may cause guest application suffer the penalty of cache miss or TLB miss.

According to the cost model, we can figure out the optimization approaches in three directions:

1. Reducing the virtualization trap counts: try to reduce Cs, Ce, Cm, Ca, Cb variables
2. Reduce the emulation time : try to reduce T_{inst}, T_{excpt}, T_{abt}, T_{miss}, T_{sync}, T_{mmio}, T_{portio}, $T_{suspend}$, ε
3. Changing virtualization model by paravirtualization or hardware assist. This can eliminate some variables.

In the following section, we will follow these guidelines to optimize the CPU and Memory virtualization. For CPU virtualization, we proposed Shadow Register file

(SRF) to reduce traps from sensitive instructions. Additionally, Fast Instruction Trap (FIT) is used to reduce the emulation time of some sensitive instructions. In virtualizing ARM's MMU, we use paravirtualization technique to eliminate the synchronization overhead; this will be further illustrated in later sections.

5 Optimization

5.1 CPU Optimization

Frequent lightweight traps for instruction emulation result in significant performance loss of the guest system. As a result, software techniques are important to minimize the frequency of "trap and emulation" count for sensitive instructions. Beside the lightweight and heavyweight traps mentioned before, two abstractions *Direct Register File Access* (DRFA) and *Fast Instruction Trap* (FIT) are proposed to accelerate the procedure of instruction emulation and reduce the overhead in CPU virtualization. The optimizing results showed great promise and will be demonstrated in later section.

We proposed *Shadow Register File* (SRF), which maps virtual CPU shadow states of the Register File into a memory region accessible by both the VMM and guest with read/write permission. Rather than unconditionally trapping on every sensitive instruction, DRFA speeds up the execution by *replacing* SWIs and sensitive instructions with a sequence of load or store instructions that access the SRF in guest address space. The methodology can be applied to instructions that only read or write the SRF and do not need privilege permission for subsequent emulation. Furthermore, VMM security is ensured because the SRF contains only the virtual state of guest, which if corrupted, would not affect the operation of the VMM.

Currently, DRFA is successfully employed to read and write the ARM PSR (Program Status Register), LDM/STM (2) and CP15 c1, c5 and c6 access in ARMvisor. To illustrate, pseudo-code for replacing a `mcr` instruction with DRFA is shown in Figure 5. The instruction is substituted by loading and effectively modifying the register copy in SRF to ensure the desired action. However, since SRF only contains subset of the VCPU state, it must be coherent with the VCPU Register File copy which ARMvisor acknowledges and without which the guest system's behavior would be unpredictable. In fact, to reduce cost of keeping them coherent, the synchronization is only held on demand and the overhead is actually low.

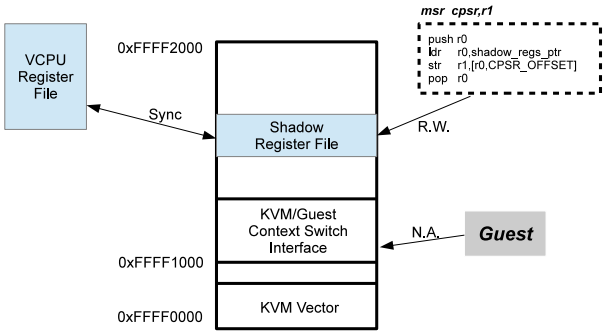

Figure 5: Shared memory mapping between KVM and Guest

Unlike the previous instructions which can be replaced with DRFA, there are other instructions which require extra emulation and which must be finished in privileged mode. As mentioned in Section 3.1.1, several sensitive instructions that relate to ARM co-processor operations (including cache operations) require higher privilege for correct emulation. Vowing to simplify the emulation path for such cases, we replaced those instructions with *Fast Instruction Trap* (FIT). This consists of a series of pre-defined macros which encode information of the replaced instructions. ARMvisor's FIT handler is actually mapped in guest address space in high memory sections because the decoding process of instructions emulation is not necessary for FIT's. Thus, in contrast to Lightweight instruction traps, the emulation overhead of FIT is considerably lower since instructions can be handled without a context switch to ARMvisor.

5.2 Memory Optimization

To reduce the overhead of memory virtualization, ARMvisor paravirtualizes the guest operating system to support shadow paging. We found that many protection faults happened during guest process creation when applying the synchronization model mentioned previously to ARMvisor. Second-level guest page table is usually modified by guest OS for Copy-On-Write or remapping usage. After analyzing such behavior in detail, we simply add two hyper-calls in the guest source code to hint ARMvisor that modifications of second-level GPT were made by guest OS. One hyper-call is added as the guest OS sets the page table entry; the other one is added to

Device	Description
CPU	ARM Cortex-A8
CPU Clock Frequency	720 MHz
Cache hierarchy	16KB of L1-Instruction Cache 16KB ofL1-Data Cache 256KB of L2 Unified cache
RAM	256MB
Board	TI BeagleBoard

Table 6: Hardware configuration

notify ARMvisor when guest OS frees a second-level page table. These notifications will free the synchronization overhead for guest second-level page tables.

	kvm_orig	kvm_opt
sen_inst_trap	12825	813
irq	6	3
fast_trap	5034	5562
dat_trans	1301	25
protection	299	0
mmio	95	56
Dabt_true	178	186
Pabt_kvm (inst_trans)	758	6
Pabt_true	106	106
mem_pv	0	714
Total_trap	20602	7471

Table 7: Profiling count for nothing

6 Evaluation

We currently support ARMv7 for host and ARMv6 for guest. For the host, we used ARM TI BeagleBoard with ARM Cortex-A8 as our platform. The detailed hardware configuration is showen in Table 6. The software parts consists of the ARMvisor (Linux 2.6.32), QEMU-ARM (0.11.1) as well as para-virtualized Guest (Linux 2.6.31.5). In the guest, we use Realview-eb ARM11 as our hardware environment for guest OS.

To demonstrate the overhead of virtualization on ARMvisor, we first measured the slowdown of ARMvisor compared with native performance when running the micro-benchmarks LMBench [11]. Then we analyze the overhead in depth by an internal profiling tool and we develop a trap cost model to explain what other overheads exist. Finally, groups of application benchmarks will also be evaluated the performance of ARMvisor.

6.1 Profiling counter

Table 7 shows the trap counts when executing a simple program in the guest that does nothing but returned immediately. Column kvm_orig shows trap counts without CPU and Memory optimization, while in column kvm_opt column all optimizations are enabled. This do-nothing program actually measures the overhead of process creation. During the procedure of fork and exec, page tables are modified frequently for mapping libraries or data into the address space. Common Operating Systems use *Demand Paging* to manage the memory usage. Nonetheless, the approach generates great overhead in Memory Virtualization due to the design of our page table synchronization model. In our synchronization model, the same page table is repeatedly being modified when forking and loading program for execution. Since we write-protect the page table when the corresponding shadow page table is found, repeated filling of entries in the page table generates consequent permission fault, and causes the corresponding shadow page being zapped often. We provide a solution to such use case by adding hyper-calls for page table modification. Page table protection traps disappear because the synchronization model is not necessary anymore. Furthermore, the subsequent page fault is eliminated since we map the memory address in shadow page in the hyper-call for page table entry modification before the guest accesses the page.

As illustrated above, kvm_opt has far less sensitive instruction traps since abundant portion of traps were replaced with direct access to SRF in the guest address space. On the other hand, the translation misses for both data and instruction access are reduced tremendously in kvm_opt, since we map the corresponding memory address in shadow page table when the hyper-call for guest page table modification is triggered. The protection trap count for page table modification also comes down to zero since any attempts by guests to modify the second level page table is acknowledged by KVM through hyper-calls and the protection model is removed. Even hyper-calls for guest page table set/free introduce certain overhead to the system; the total trap count of Memory Virtualization does decline enormously. In conclusion, after the optimization in CPU and Memory Virtualization are applied, the total trap count is only about 36% compared to original version.

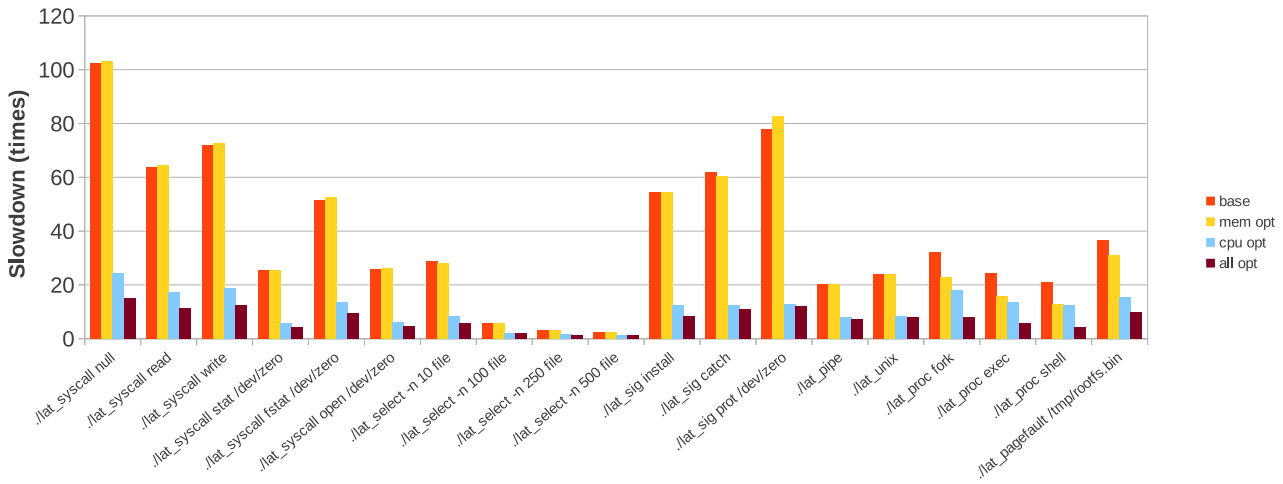

Figure 6: Ratio of performance slowdown on LMBench

6.2 LMBench

To evaluate the performance when applying different optimization on ARMvisor in guest system's basic operation, LMBench is measured to help us understand the bottleneck of our current design. LMBench contains suites of benchmarks that are designed to measure numerous system operations such as system call, IPC, process creation and signal handling. We refer to the paper [12] to choose sets of benchmarks in LMBench and measured them in various environments and compare them with native performance.

Figure 6 presents the ratio of slowdown on 3 different extensions of features comparing to native Linux kernel. The *simple* version means no optimization applied in the ARMvisor, all sensitive instructions will be trapped into hypervisor and the guest OS does not inform ARMvisor for creating or modifying a page table so hypervisor needs to trace the guest page table. The *cpu-opt* version improves the performance of ARMvisor in three aspects: firstly it use SRF technique to reduce the trap overheads from several sensitive instructions for instructions as MSR, MRS and CPS. Secondly, it uses binary translation to change guest TLB/Cache operations on-the-fly. All of the TLB operations are translated into NOP operations since the guest no longer needs to maintain hardware TLB, and the hypervisor will assist the guest to maintain TLB according to the shadow page tables. And finally, the overhead of exception/interrupt handling in guest OS is reduced by finishing the combination logic of guest code in ARMvisror to largely reduce the traps of critical instructions. The *full-opt* version further comprise the memory para-virtualization which uses hyper-calls to inform ARMvisor about guest page table modification and finalization, and ises SWI fast trap to reduce the memory tracing overhead and SWI delivery overhead.

As can be seen in Figure 6, system call and signal handling related benchmarks suffer great performance loss in ARMvisor. Performance slowdown is less significant in `lat_select` benchmark, typically when the number of selected fd increases since the accounted time portion for native execution increases. Nonetheless, the slowdown still reaches 30 times when selecting 10 files. After measuring those benchmarks using the proposed cost model, we figured that the performance gap could be mainly attributed to the frequent lightweight traps for instruction emulation during the execution path. Each trap includes a pair of context switches between guest and VMM, which is time consuming. Decoding individual sensitive instructions for correct emulation also generate latencies for the total execution. Moreover, since the SWI exception is indirectly generated by ARMvisor's virtual exception emulator, the cost of exception delivery adds additional slowdown to the system call operation. However, after applying CPU-opt in ARMvisor, performance improves largely since the times of lightweight trap for instruction emulations reduces. Mem-opt hardly contributes any performance improvement for system call and signal han-

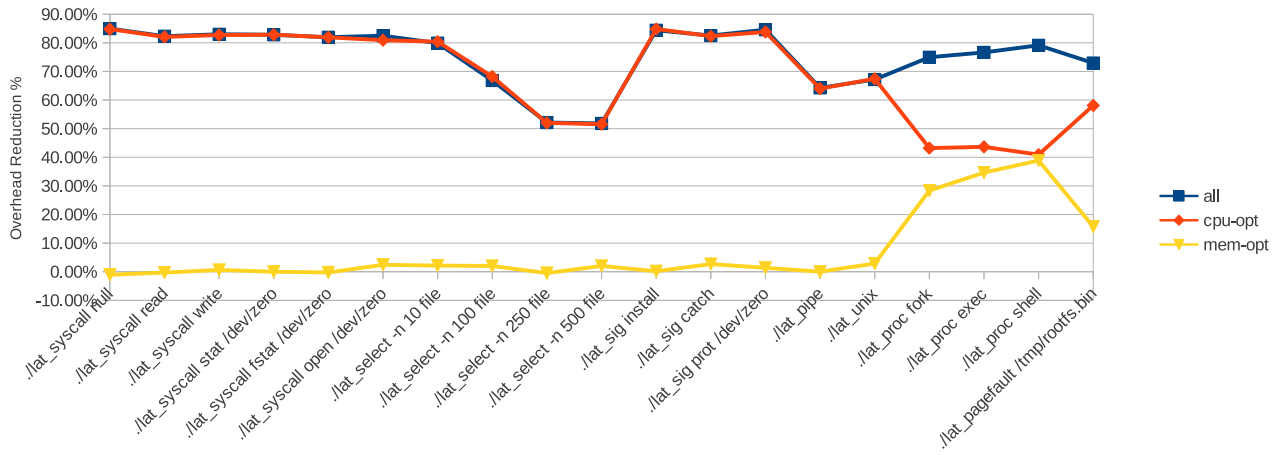

Figure 7: Ratio of performance improvement on LMBench

dling since those operations also would not require large amounts of page table modification. Figure 7 demonstrates that All-opt improves benchmarks' performance in categories of system call, signal handling and select approximately over 80% on average.

The All-opt version has less profound improvements in IPC related benchmarks including `lat_pipe` and `lat_unix`. According to our profiling results, I/O access rate in IPC operation is much higher contrast to the three types of benchmarks mentioned above. The behaviors are quite different in pipe and Unix Socket operation. In the scenario of communicating through pipe, the guest kernel scheduler is often called to switch the process for inter-process message sending and receiving. Linux Kernel scheduler fetches hardware clock counter as TSC (Time Stamp Counter) for updating the run queue reference clock for its scheduling policy. Since we currently focus on minimizing CPU and Memory virtualization overhead, All-opt improve less significantly in pipe operation, since the overhead results in frequent heavyweight traps exists for I/O emulation.

Process creation benchmarks like `lat_procfork`, `lat_procexec` and `lat_procshell` have similar behavior as the do-nothing program. As previously mentioned, synchronization for page tables in Memory Virtualization results in considerably larger overhead for process operations than in the other benchmarks. As shown in Figure 7, the performance of process creation operation improves about 45% on average in Mem-opt solely.

Benchmark `lat_pagefault` tests the latency of page fault handling in operating system. Even though we map the page in shadow on the PTE write hyper-calls to prevent further translation miss; performance improvement in Mem-opt is smaller than in process creation benchmarks. Profiling by our cost model, we found that sensitive instruction traps account for larger proportion of total traps than those of process creation. As a result, the optimization for CPU virtualization has more notable effect on performance improvement.

7 Conclusion and future works

In this paper, we investigated the challenges of constructing virtualization environments for embedded systems by the implementation of an ARM based hypervisor, ARMvisor. ARMvisor assumes that ARM processor has no hardware virtualization extension. The experimental results show that ARMvisor suffers huge performance degradation when techniques such as trap-and-emulation and shadow paging are being adopted. As a result, we formulized a cost model for discovering the performance bottlenecks of hypervisor in depth. Furthermore, based on the cost model, several optimization methodologies are proposed to reduce the overheads of CPU and Memory virtualization. The experimental results have shown leaping performance improvement with up to 4.65 times speedup in average on LMBench by comparing with our original design. We conclude that to virtualize embedded ARM platforms without hardware virtualization extension in current architecture, the cost model is crucial to assist developers to further optimize their hypervisors.

References

[1] Christoffer Dall and Jason Nieh Columbia University, "KVM for ARM," in *the proceeding of Linux Symposium 2011*

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412-421, 1974.

[3] Gernot Heiser, "The role of virtualization in Embedded Systems", in *IIES '08 Proceedings of the 1st workshop on Isolation and integration in embedded systems*

[4] Gernot Heiser, Ben Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors", in *APSys '10 Proceedings of the first ACM asia-pacific workshop on Workshop on systems*

[5] VMware. "VMware Mobile Virtualization, Platform, Virtual Appliances for Mobile phones", http://www.vmware.com/products/mobile/

[6] Trango. "Trango: secured virtualization on ARM", http://www.trango-vp.com

[7] VirtualLogix. "VirtualLogix Real-Time Virtualization and VLX", http://www.osware.com

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, "Xen and the Art of Virtualization", in *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*

[9] Joo-Young Hwang; Sang-Bum Suh; Sung-Kwan Heo; Chan-Ju Park; Jae-Min Ryu; Seong-Yeol Park; Chul-Ryun Kim; Samsung Electron. Co. Ltd., Suwon , "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones", in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*

[10] Carl A. Waldspurger VMware, Inc., Palo Alto, CA, "Memory Resource Management in VMware ESX Server", in *ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*

[11] McVoy, L. W., & Staelin, C. (1996). "lmbench: Portable tools for performance analysis," In *USENIX annual technical conference*. Barkely (pp. 279-294), Jan. 1996.

[12] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, Attila Bilgic. "Performance Evaluation of Para- virtualization on Modern Mobile Phone Platform", in *Proceedings of International Conference on Electrical, and System Science and Engineering*

[13] Keith Adams , Ole Agesen, "A comparison of software and hardware techniques for x86 virtualization", *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 21- 25, 2006, San Jose, California, USA

[14] Joshua, LeVasseur, et al. "Pre-Virtualization: Slashing the Cost of Virtualization", http://l4ka.org/publications/2005/previrtualization-techreport.pdf, 2005.