NestCloud: Towards Practical Nested Virtualization

Zhenhao Pan
Tsinghua University, China
frankpzh@gmail.com

Qing He
Intel Asia-Pacific Research and Development Ltd
qing.he@intel.com

Wei Jiang
Tsinghua University, China
jwhust@gmail.com

Yu Chen
Tsinghua University, China
yuchen@tsinghua.edu.cn

Yaozu Dong
Intel Asia-Pacific Research and Development Ltd
eddie.dong@intel.com

***Abstract*—This paper describes a nested virtualization solution, which allows virtual machine monitor (VMM) with virtual machine to run within another virtual machine with low overhead. Previous nested virtualization solutions on x86 platform are mainly based on emulation, which result in poor performance and poor usability. We propose and implement `NestCloud`, a practical high performance nested virtualization architecture, which fully employs the hardware virtualization extensions. Furthermore, three optimizations are provided to reduce the overhead of nested guests: (1) Guest Page Fault Bypassing, which permits nested guests to handle page faults without VM Exit; (2) Virtual EPT (Extended Page Table), which eliminates unnecessary page faults introduced by shadow page table in nested VMM; (3) PV VMCS, which provides more effective VMCS accessing for nested VMM. Experimental results show that the performance of NestCloud guest is close to single level guest in both CPU-intensive and memory-intensive benchmarks. The CPU overhead is 5.22% and the memory overhead is 5.69%, which makes the nested guest of NestCloud comparable with a conventional one.**

***Index Terms*—Nested Virtualization, Virtual Machine Monitor (VMM), Virtual-Machine Control Structure (VMCS)**

I. Introduction

Virtualization has been widely used nowadays. In data centers and cloud computing environments, virtualization can largely reduce the hardware costs and resource costs[1], [2], [3]. There are commercial VMM (Virtual Machine Monitor) implementations such as VMware[4] and Microsoft Hyper-V[5], and open source implementations such as Xen[6], KVM[7], [8], VirtualBox[9] and lguest[10]. Para-virtualization and full virtualization are two common virtualization techniques. Para-virtualization modifies the guest OS (operating system) to provide virtualization on legacy processors. Full virtualization, on the other hand, virtualizes the guest OS without any modification[11].

Nested virtualization, which is also known as recursive virtualization[12], allows one VMM to run within a virtual machine provided by another VMM. Although nested virtualization has not been widely used, we can still list several important usage models. We believe some of them have great potential in the future.

- Some latest OS features and applications are necessary to run in virtualization environments. Windows XP mode[13] is an example, which runs traditional Windows XP upon Windows 7 by virtualization. It is impossible to run Windows XP mode when the Windows 7 is already running in a virtual machine.
- Recently, embedded virtualization technology (A.K.A hypervisor in firmware) has been adopted in some servers, which means the booted OS is already in a virtual machine. Nested virtualization can enable the traditional VMM working normally.
- With help of nested virtualization, it is easy and efficient to debug and monitor guest OS upon a VMM, and even VMM itself.
- For future cloud computing environment, the different virtualization solutions may vary much from each other, like diverse OS currently. Guest OS images for different VMMs may not be able to run on or live migrate[14] between different VMMs. The nested virtualization is a solution to this problem.

To make virtualization much easier and faster, hardware vendors like Intel and AMD have added extensions to x86 architecture[11], [15]. Previous researches[16], [17], [18], [19] show that virtualization can achieve very high performance with these extensions. But on nested virtualization, current solutions have far worse performance than conventional virtualization. Previous studies of nested virtualization was designed using micro-kernels[20] or on special virtualizable hardware architecture[12]. Among the recent virtualization implementations, only KVM can support nested virtualization. Furthermore, KVM only supports AMD-V in nested virtualization[21]. On Intel processor, KVM can only use QEMU[22] emulation, which has a very low performance and is not practical in reality.

Based on the hardware extensions for virtualization, this paper proposes a new nested virtualization architecture called NestCloud. NestCloud uses the VMX instructions as interfaces, which is general and easy enough to apply to most VMMs on x86 platform. Benchmark results indicate that NestCloud only introduces 5.22% CPU overhead and 5.69% memory overhead.

The remainder of this paper is organized as follows: Section II introduces the hardware extensions for virtualization (VMX) and KVM. Section III gives a description to the architecture design of NestCloud. Section IV explains the implementation of NestCloud. Section V discusses three optimizations to the NestCloud. Section VI uses well-known benchmarks such as SPEC CPU 2006, kernel build and SysBench to evaluate NestCloud's performance. At last, Section VII is the related work and Section VIII is the conclusion and future work.

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

II. Background

In this section, we introduce the background of Intel's hardware extensions for virtualization and KVM (Kernel-based Virtual Machine).

A. Hardware Extensions for virtualization

The classic x86 architecture is not virtualizable according to Popek and Goldberg's virtualization requirements[23]. Current implementations of virtualization on x86 need either patches on the guest kernel, or hardware changes such as Intel VT[11] and AMD-V[15]. Xen[6] is an example of the formal one, and KVM[7] is an example of the latter one. NestCloud is based on Intel VMX extension[11].

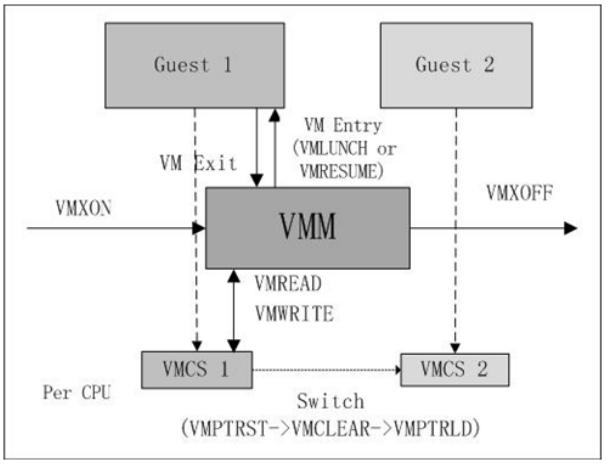

Fig. 1. VMX instruction, interaction of VMM and Guest

Fig.1 represents the instructions of VMX and how VMM and guests interact with each other. In terms of VMX, two operation modes are provided. Root operation mode is fully privileged and used by VMM. On the other hand, non-root operation mode is not fully privileged and used by guest OS. Software can enter VMX non-root operation mode using VM Entry instruction (VMLAUNCH or VMRESUME). In contrast, VM Exit is triggered by certain instructions and events in VMX non-root operation mode, and leads the processor to root operation mode.

VMX contains a structure called VMCS (Virtual-Machine Control Structure). Each logical processor associates a memory region for VMCS, which is called VMCS region. VMCS regions are organized into six groups: Guest-State area, Host-State area, VM-execution control fields, VM Exit control fields, VM Entry control fields, and VM Exit information fields. Each of them contains one aspect of VMX information. For example, both Guest-state area and Host-state area contain the fields that corresponding to different components of processor state. When VM Exits happen, processor states of guest are saved to the Guest-state area and processor states are loaded from the Host-state area to restore host context. As shown in Fig.1, VMX also provides several instructions to manage VMCS regions.

The remaining parts of this paper frequently use VMCS to refer to a VMCS region associated to one logical processor.

EPT (Extended Page Table)[11] is a hardware extension for optimizing performance of memory virtualization. When EPT is active, separate page tables are provided to translate guest-physical addresses to the host-physical addresses. Meanwhile the traditional page tables finish the translation from guest-liner address to guest-physical address.

EPT takes over the technique of shadow page table, avoids the expensive VM Exits and complex handling procedures of guest page faults, and therefore brings programming flexibility and performance improvement. Besides, EPT avoids memory usage of shadow page table which needs a whole copy of guest page tables.

B. KVM

KVM (Kernel-based Virtual Machine)[7] is a virtualization solution integrated in Linux kernel, which consists of a loadable kernel module that provides the core virtualization infrastructure and a processor specific module. As a kernel module in Linux, KVM leverages existing Linux features and provides an integrated VMM approach. Virtual CPUs (vCPUs) of KVM guests are normal threads in the host OS, while memories of KVM guests are mapped into the memory space of their corresponding threads. KVM is a relatively new but mature virtualization solution for Linux on x86 architecture. Studies show the KVM has comparable performance to Xen[24].

III. Architecture

Using QEMU[22], KVM is able to run nested virtualization with low performance compare to conventional virtualization. Guest's code can be accelerated on the physical processor by virtualization extensions. In the nested environment however, there is only one VMM can run on the real hardware and utilize hardware extensions. The nested VMM only has a hardware layer provided by the underlying VMM, which has no hardware extension.

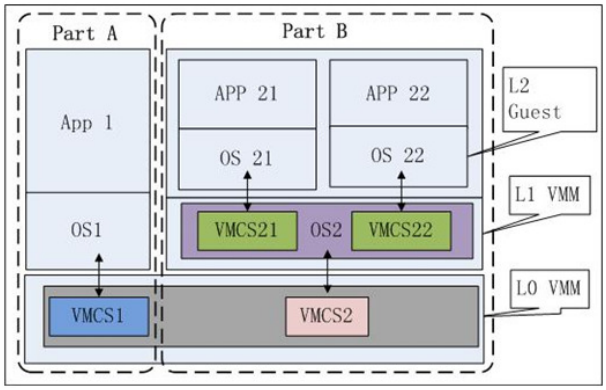

Fig. 2. Three-Level Nested Virtualization Architecture

We designed NestCloud, a three-level architecture for nested virtualization. NestCloud provides the ability to use the hardware extensions for the nested VMM. Fig.2 represents the

architecture of NestCloud. Fig.2 can be separated into two parts. Part A is the traditional architecture, which includes a normal guest and a VMCS associated with the vCPU (virtual CPU) where guest OS OS1 runs on. Part B is the architecture of NestCloud, which consists of three levels. In level 0 runs L0 VMM, which is a modified VMM running on the real hardware. Components in level 1 can either be a guest or a VMM. Component in level 1 is called L1 VMM when it is a VMM, and L0 VMM is transparent to it. Hardware layer of L1 VMM is provided by L0 VMM. Like a typical VMM, L1 VMM can create its own guest. Components on Level 2 are nested guests, which are called L2 Guest in this paper.

In NestCloud, no modification on L1 VMM or L2 Guest OS is needed. Optimizations provided in the following sections may need slight modification on L1 VMM, and we will discuss it later.

Focusing on VMX extension, only L0 VMM runs in VMX root operation mode. L1 VMM and L2 Guest run in VMX non-root operation mode. NestCloud provide a nested VMX interface to L1 VMM in order to accelerate L2 Guest using VMX extension. The following subsections explains the nested interface.

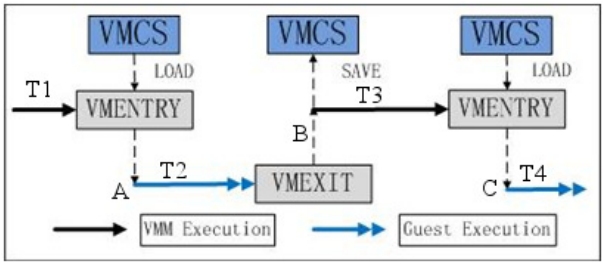

Fig. 3. Non-Nested Virtualization CPU Execution Flow

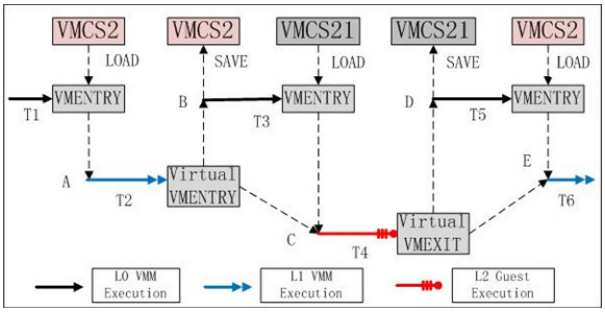

Fig. 4. Nested Virtualization CPU Execution Flow

A. Nested VMX Interface

As we described in Section II, VMCS, which controls the transition of two operation modes, is the most important component in VMX. In conventional virtualization, one VMCS is associated with one logical processor. In nested virtualization, the L1 VMM not only has its own logical processor (intrinsic vCPU), but also has L2 Guest's logical processor inside (shadow vCPU). When L2 Guest is running, the VMCS of its logical processor is supposed to be associated with the physical processor, thus the support of VMCS needs to be extended.

NestCloud proposes three concepts of VMCS: the intrinsic VMCS (iVMCS), the shadow VMCS (sVMCS) and the physical VMCS (pVMCS). The first two are correspond to the L1 VMM's VMCS and the L2 Guest's VMCS. The last one is the VMCS used by the physical processor. They have the relationship as following:

$$pVMCS = \begin{cases} iVMCS & \text{when running in L1 Guest} \quad (1) \\ sVMCS & \text{when running in L2 Guest} \quad (2) \end{cases}$$

For VMX instructions, NestCloud uses the traditional trap-and-emulate method. VMX instructions issued by L1 VMM will cause VM Exit and be trapped into L0 VMM. Using instruction parameters got from VM Exit reasons, L0 VMM handles the requests and operations on the real VMX extension. In this way, L1 VMM can use VMX extension to improve the performance of L2 Guests.

B. Nested CPU Execution Flow

In a non-nested guest, the execution flow with VMX is shown in Fig.3. At time A, the VMM issues a VM Entry instruction to wake up the guest, and the system turns into non-root operation mode. During T2, guest's instructions are executed on the physical processor directly. At time B, VM Exit happens, and the processor execution turns back to the VMM to handle the VM Exit event.

Fig.4 is the CPU execution flow in NestCloud, which involves the three levels' interaction. At time A, L0 VMM issues a VM Entry to turn on L1 VMM. L1 VMM issues the virtual VM Entry at time B, which causes a VM Exit and the switch of VMCS from VMCS2(iVMCS) to VMCS21(sVMCS). At time C, L0 VMM issues the real VM Entry which calls up L2 Guest. So far, the L2 Guest can get a running opportunity during T4. The L2 Guest keeps running on the physical processor until a virtual VM Exit happens at time D.

C. Handling VM Exits

The procedure of handling VM Exits from L2 Guest differs in NestCloud. Unlike non-nested situation, where VM Exits are all handled by the VMM. In NestCloud, L0 VMM needs to decide the handler of VM Exits. If a VM Exit is due to L0 VMM, shadow page faults and external IRQs for example, L0 VMM handlers will handle it.

If L1 VMM is responsible for the VM Exit, L1 VMM should be turned on to handle it. In this situation, pVMCS needs to be switched to iVMCS, and a virtual VM Exit needs to be injected into L1 VMM. The virtual VM Exit is constructed according to EXIT_REASON in vVMCS. If the switch is due to virtual IRQs, a new EXIT_REASON is generated.

If the VM Exit is due to L2 Guest, L0 VMM will inject a virtual VM Exit to L1 VMM, and L1 VMM will read the VM Exit reason and inject it to L2 Guest. Events such as L2 page faults are handled this way.

IV. Implementation

In this section, we describes the implementation details of NestCloud.

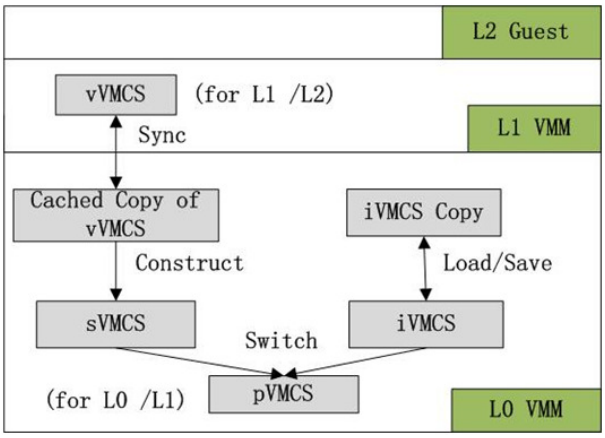

Fig. 5. Nested VMCS Design

A. Nested VMCS Implementation

In nested VMCS implementation, the iVMCS for L1 VMM is in the L0 VMM's memory space. The sVMCS is constructed by L0 VMM according to VMCS for L2 Guest in the L1 VMM's memory space, which is also called vVMCS. In order to simplify the procedure of accessing vVMCS, a copy of vVMCS is kept in L0 VMM's memory and synchronized with L1 VMM. Fig.5 represents their relationships.

B. Trap-and-emulation of VMX Instructions

When L1 VMM issues a VMX instruction, it generates a VM Exit which is trapped by L0 VMM. A handler in L0 VMM will handle the VMX instructions on behalf of the L1 VMM. These handlers take advantages of the real VMX extension which makes the performance of L2 Guest close to L1 Guest.

Five VMCS maintenance instructions and five VMX management instructions are provided by VMX extension[11], and all of them has a corresponding handler in L0 VMM. Here we describe implementation details of some important instructions handlers.

1) Virtual VMPTRLD/VMPTRST Handling: VMPTRLD [11] loads the current VMCS region pointer from memory. The handler of VMPTRLD fetches the address of the new VMCS region by decoding the VM Exit reason, and synchronizes the L0 VMM's copy of vVMCS. For later reference, the address of the new VMCS region is also saved in L0 VMM. VMPTRST stores the current VMCS pointer into memory, and the handler is similar. The vVMCS in L1 VMM is synchronized with the copy in L0 VMM, and the saved address is returned.

2) Virtual VMCLEAR Handling: VMCLEAR ensures all fields of VMCS are copied to VMCS region[11]. The handler of this instruction just synchronizes the L0 VMM's cached copy with the vVMCS in L1 VMM's memory.

3) Virtual VMREAD/VMWRITE Handling: VMREAD reads a specified VMCS field[11]. The handler works as follows: (1) Decoding VMREAD information from the exit information of VM Exit. (2) Reading the specified field from the L0 VMM's vVMCS copy. (3) Saving the value to the specified register in the exit information. The handler of VMWRITE works similar. It does the writing on vVMCS copy instead of reading.

4) Virtual VMLAUNCH/VMRESUME Handling: These two instructions launch or resume a guest managed by current VMCS and then transfer control to the guest[11]. They are handled in the same way in nested virtualization environment. In Fig.3, "VMENTRY" and "Virtual VMENTRY" are examples of these two instructions. VMPTRST, VMPTRLD and VMCLEAR are preparations of these two instructions. The pVMCS differs before and after the VMRESUME. It points to iVMCS when L1 VMM is running, and points to sVMCS when L2 Guest is running. When L0 VMM handles VMRESUME, the pVMCS should be switched from iVMCS to sVMCS. After pVMCS switching, L0 VMM can enter L2 Guest by a real VMRESUME instruction.

V. Optimizations

Section IV introduces the implementation of NestCloud. In this section we describe the optimizations on NestCloud. The goal of optimizations is to eliminate the performance gap between L2 Guest and L1 Guest. We provide 3 optimizations including Guest Page Fault Bypassing, Virtual EPT and PV VMCS. The idea of these optimizations is to reduce the transitions between L0, L1 and L2, which are considered as one of the root causes of the overhead.

A. Guest Page Fault Bypassing

Page faults can occur for a variety of reasons. In some cases, page faults alert the VMM to an inconsistency between the page table and its shadow copy[25]. In other cases, the hierarchies are already consistent and the page fault should be handled by the guest OS. The formal cases are called shadow page faults and can only be handled by the VMM, while the latter cases do not need interceptions of VMM at all.

The optimization of guest page fault bypassing makes the L2 Guest handle its own page faults without causing a VM Exit to save transition time. It is implemented by a feature of VMX. VMX provides 2 registers in VMCS: PFEC_MASK and PFEC_MATCH. When the page fault error code (PFEC) matches these 2 registers (PFEC & PFEC_MASK = PFEC_MATCH), the page fault will be delivered through guest's IDT without causing a VM Exit[11]. In this optimization, PFEC_MASK and PFEC_MATCH are set to 1, so that page faults caused by non-present pages do not cause VM Exit at all. The key information to separate 2 page fault cases is that the reason of shadow page fault cannot be non-presented pages. In such a way, only page faults of L2 Guest are bypassed.

Not all page faults of L2 Guest are caused by non-presented pages. This optimization does not work for the page faults

caused by illegal access or other reasons. To judge the effectiveness of this optimization, we collect the count of page faults during a kernel building. KVMTrace[7] is a module in Linux kernel which can record the KVM event timestamps and event parameters. It is used to count the page faults of VM Exit from L2 Guest.

Page faults coming from L2 Guest are separated into 3 categories: (1) L0 shadow page fault, which is solved by L0 directly; (2) L1 shadow page fault, which is injected into and handled by L1 VMM; (3) L2 page fault, which is injected into L2 guest through L1 VMM. The expected effect of this optimization is reducing the count of L2 page faults we caught.

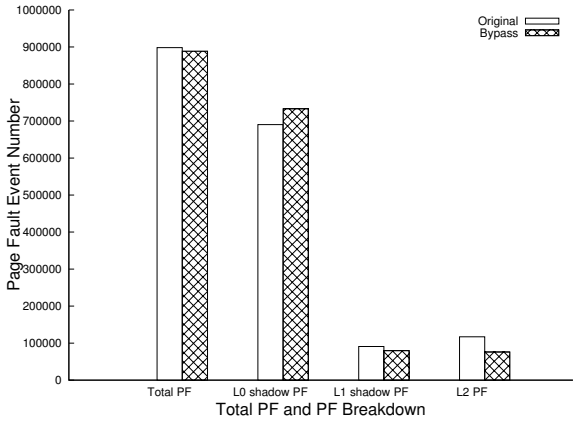

Fig. 6. Guest Page Fault Bypassing in Kernel Building

Fig.6 shows a 60 seconds sample of page fault count. In the meantime, we get a 5% performance gain during kernel building. The count of VM Exits caused by L2 page faults is reduced by 35% after the guest page fault bypassing. In the meanwhile, the L0 shadow page fault is increased by 6.2% due to the performance gain (L2 Guest did more during 60 seconds sample). Because only 13.13% of page faults are L2 page faults, the performance gain is not as good as we expected.

B. Virtual EPT Support

EPT can largely improve guest's performance. In this optimization, a concept of virtual EPT is proposed. Virtual EPT support is used in L1 VMM and works for L2 Guest's page table. Consequently, the EPT support provided by hardware is called host EPT.

Host EPT has already been supported by KVM as we described in Section II. It also creates a great performance gain on nested virtualization. But currently, EPT has not been supported in L1 VMM. Address translation of L2 Guest has to use the shadow page table mechanism and causes a lot of VM Exits.

We present a full EPT interface to L1 VMM by trapping all the EPT events from L1 VMM, and forward them directly to the real hardware. Meanwhile, the hardware EPT events are injected into L1 VMM by L0 VMM, such as EXIT_REASON_EPT_VIOLATION and EXIT_REASON_EPT_MISCONFIG. With virtual EPT, VM

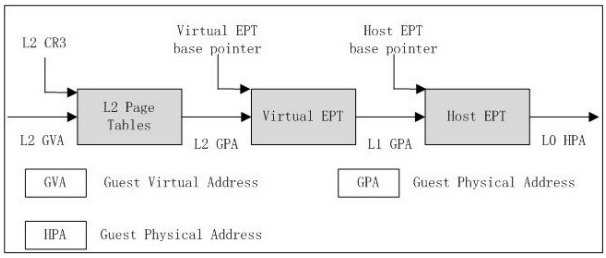

Fig. 7. Virtual EPT Support

Exit by shadow page table will be significantly reduced and the performance can get a boost. Notice that virtual EPT is supported only when the host EPT is enabled, because the virtual EPT is implemented by forwarding events to the host EPT. Fig.7 shows how the host EPT and virtual EPT work.

TABLE I
L1 VMM EVENTS BREAKDOWN

Event	Percentage
VMREAD	67%
VMWRITE	19%
Exception	7%
VMRESUME	6%
Others	1%

C. PV VMCS

In order to uncover the performance bottleneck of L1 VMM, we collected statistic information on the VMX events during kernel building. Table I is the breakdown of all events in L1 Guest VM Exit reasons. 86% of VM Exits are due to VMREAD and VMWRITE. Before optimization, every time when L1 VMM accesses a vVMCS field, VMREAD or VMWRITE causes a transition from L1 VMM to L0 VMM, and L0 VMM will access the field in vVMCS copy. Actually, L1 VMM has its own copy of vVMCS, thus it has full knowledge to perform VMREAD and VMWRITE by itself.

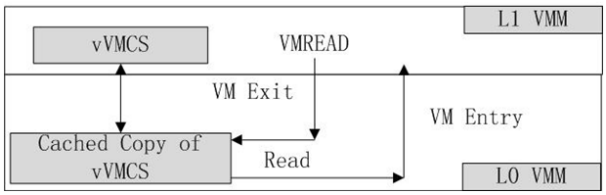

Fig. 8. Before PV VMCS Optimization

In order to enable vVMCS access in L1 VMM, we need to expose vVMCS layout and accessing method in L1 VMM. Besides, L0 VMM should be slightly modified too. As we mentioned in Section III, L0 VMM holds a vVMCS copy, which is synchronized with vVMCS in L1's memory. This copy should be updated explicitly in this optimization. Fig.8 and 9 shows the PV VMCS optimization of VMREAD.

The effect of PV VMCS varies according to different applications. The PV VMCS needs modifications on the L1

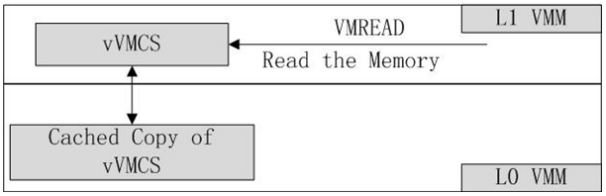

Fig. 9. After PV VMCS Optimization

VMM, which is not applicable in some situations such as commercial virtualization solutions.

VI. Evaluation

We have implemented NestCloud and the optimizations on KVM-84[7]. In this section, we evaluate the performance of NestCloud. We try to prove that: (1) NestCloud is better than the nested solution of QEMU on KVM (2) With optimizations, the performance of NestCloud is close to that of L1 Guest on CPU and memory.

Most evaluations have 7 situations: L1 (L1 Guest performance), QEMU (nested virtualization using QEMU emulation with host EPT), Basic (implementation of NestCloud with no optimization), Bypass (using both L1 VMM and L2 Guest page fault bypassing), PV VMCS (BASIC with PV VMCS), Host EPT (BASIC with host EPT), Host/Virtual EPT (BASIC with host and virtual EPT), Host/Virtual EPT + PV VMCS (BASIC with host EPT, virtual EPT, and PV VMCS). Our goal is to make the performance of L2 Guest close to a normal guest (performance of L1 Guest with host EPT), thus some results are normalized to L1.

A. Environment and benchmarks

We performed all experiments on a server with a VT-enabled Intel core i7-920 and 6 GB memory. The host/guest OS used in our tests is Ubuntu 9.04. The L0 VMM's kernel is KVM-84[7] with NestCloud; the L1 Guest's kernel is KVM-84 with no modification; and the L2 Guest uses original kernel of Ubuntu 9.04. To make the L2 Guest time accurate, KVM PV-TIMER module (CONFIG_KVM_CLOCK=y) is enabled in the L2 Guest kernel.

VMX extension is used for CPU virtualization, which is the focus of our tests. SPEC CPU 2006[26], [27] is an industry-standardized, CPU-intensive benchmark suite. It contains two test packages: CINT tests and CFP tests. Benchmarks in SPEC CPU 2006 are derived from real world applications. They spend at least 95% of its execution time in user space[27]. SysBench-CPU[28] uses calculation of prime numbers up to a specified value, and the result is valued in running time.

In addition, we use SysBench-Memory[28] to measure the memory performance. To get I/O performance, SysBench OLTP[28] is used. OLTP stands for On-Line Transaction Processing. SysBench OLTP keeps generating transactions for MySQL when it is running.

TABLE II
SysBench-CPU Results

	Results(s)
L1	36.0535
Basic	38.2076
Bypass	38.7977
Host EPT	40.7520
Host EPT + Virtual EPT	38.4142
PV VMCS	37.8735
PV VMCS, Host EPT + Virtual EPT	37.9351
QEMU	785.7888

B. CPU Performance

The results of SysBench-CPU is presented in Table II. Differences between Basic situation and situations with optimizations are quite small, and they are about 21 times faster than QEMU. In the situation of Host/virtual EPT and PV VMCS, L2 Guest introduces 5.22% overhead compare to L1 Guest.

The VMX interface of NestCloud enables the L2 Guest's instruction to execute on the physical CPU directly. In a CPU-intensive benchmark like SysBench-CPU, the overhead of an additional level is quite small.

SPEC CPU 2006 on QEMU nested environment has very low performance, and some benchmarks fail to get a result. Here we only provide bzip2 and gcc results in Table III, which shows that the QEMU nested virtualization can only get about 5% of a L1 Guest's performance.

TABLE III
QEMU Nested SPEC CPU 2006 Results

	L1	QEMU
bzip2	756	11872
gcc	420	8109

Fig.10 shows 12 results of CINT benchmarks, and Fig.11 shows the results of CFP benchmarks. These results are normalized to L1 Guest's results. Compare to SysBench-CPU, SPEC CPU 2006 is a mixed benchmark, which consists of CPU workload, memory workload and a little bit of I/O workload. The effects of optimizations varies between different tests.

1) Effect of virtual EPT: Virtual EPT works extremely well in some of the benchmarks, including gcc in CINT, soplex and tonto in CFP. After an investigation on these benchmarks, we figure out that these benchmarks perform many memory allocations and freeings[29]. These activities lead to page table changes, and therefore provide bad results with shadow page table. In the following subsection, we will discuss performance of shadow page table in detail.

Also, virtual EPT does not work in some cases, including sjeng, xalancbmk in CINT and bwaves, zeusmp and lbm in CFP. The performance of Intel EPT has lower performance under: (1) little MMU activity (2) high TLB miss rate[30]. And, all these benchmarks have relatively higher TLB miss rate[31],

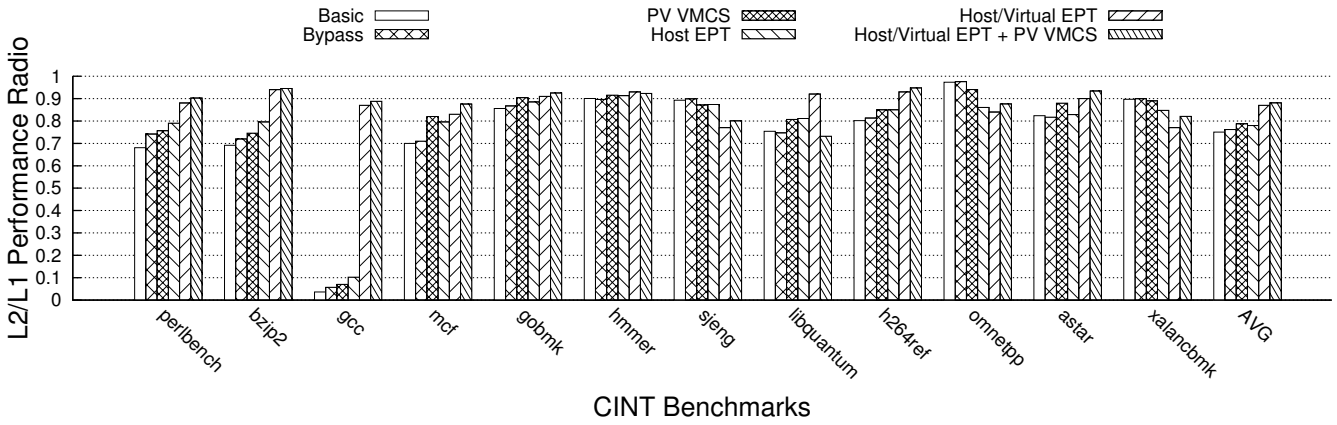

Fig. 10. SPEC CPU 2006 CINT Results

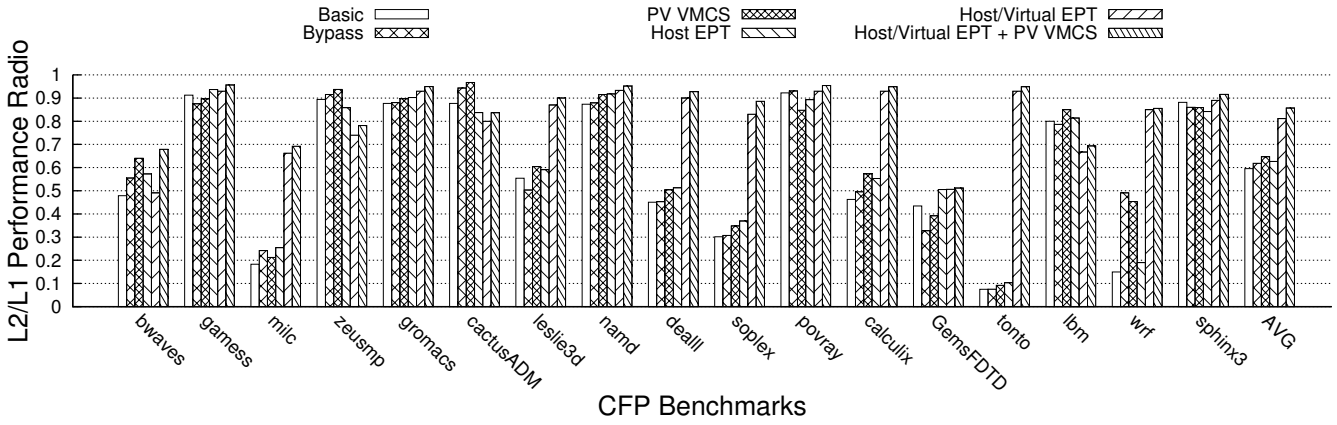

Fig. 11. SPEC CPU 2006 CFP Results

together with few memory allocation/freeing activities[29].

2) Effect of PV VMCS: Actually, PV VMCS is a trade-off that works only when the frequency of VMREAD and VMWRITE is high enough. In a rare case, the synchronization cost of vVMCS is larger than the performance gain, and this optimization will get worse result. The test of libquantum in CINT is an example. PV VMCS works for it, but does not work when virtual EPT is also applied. The reason is that virtual EPT will significantly reduce the VMREAD/VMWRITE caused by page faults, and PV VMCS will not work as good as before. Similar results can be found in the test of PF-Bench following.

In conclusion, L2 Guest with optimizations can achieve 88.08% of L1 Guest in CINT benchmarks and 85.68% of L2 Guest in CFP benchmarks, which means 13.53% and 16.71% overhead.

C. Memory Performance

Table IV shows the result of SysBench-Memory. Similar to SysBench-CPU results, Basic situation and optimized situation vary slightly. Also, they are about 11 times faster than QEMU because of the VMX interface. The best result of SysBench-Memory presents 5.69% overhead compare to L1 Guest.

TABLE IV
SYSBENCH-MEMORY RESULTS

	Results(s)
L1	54.1131
Basic	57.6744
Bypass	57.3680
Host EPT	57.3903
Host EPT + Virtual EPT	57.3920
PV VMCS	56.6564
PV VMCS, Host EPT + Virtual EPT	56.5042
QEMU	647.9132

In order to measure our optimization effort on page faults, we design a micro-benchmark called PF-Bench, which keeps generating page faults when it is running. Page faults in L2 Guest without any optimization are heavy. Each of them triggers several VM Exits and VM Entries, and lets the CPU go back-and-forth between L0 VMM and L1 VMM. When L2 Guest is handling page faults, it modifies the page table, and triggers a L1 shadow page fault. When the memory pages of L2 Guest page faults are also absent from L1 VMM's page table, they trigger another page faults of L1 VMM. Furthermore, L1 VMM can also trigger L0 shadow page faults

TABLE V
PF-BENCH RESULTS

	Results(s)
L0 Performance	1.37
L1	23.85
Basic	501.01
Bypass	470.25
Host EPT	358.98
Host EPT + Virtual EPT	2.39
PV VMCS	71.01
PV VMCS, Host EPT + Virtual EPT	5.6
QEMU	35.90

when it is modifying its page table. Every page fault from L2 Guest triggers a page fault chain, which cost much CPU time.

The results are given as running time in Table V. Bypass works for page faults of L2 Guest. It eliminates the back-and-forth of L2 Guest page fault, and has a 6.54% performance gain. Host EPT works for L0 shadow page faults, and it has an acceleration of 39.56%. PV VMCS largely reduces the cost of VM Entry and VM Exit between L1 VMM and L2 Guest, and has a speedup of 605.55%. The best optimization is virtual EPT, it is 150+ times faster than Basic. The result of QEMU is better than Basic, and even better than several optimized situations such as Bypass and Host EPT. This is because QEMU does not use shadow page table, and avoids the heavy work of back-and-forth between levels.

D. I/O Performance

TABLE VI
SYSBENCH-OLTP RESULTS

	Results(t/s)
L1	535
Basic	13.92
Bypass	16.34
Host EPT	16.19
Host EPT + Virtual EPT	44.38
PV VMCS	19.12
PV VMCS, Host EPT + Virtual EPT	48.96
QEMU	13.23

Table VI is the test results of SysBench OLTP benchmark. The performance of L2 Guest is only 10% of the L1 Guest's. The low performance of I/O in L2 is understandable, since all the I/O operations needs back-and-forth between 3 levels just like the situation of page fault. However, the best optimization result is 3.7 times better than the QEMU nested.

In this paper, we do not explicitly optimize the I/O performance. The OLTP test uses emulated I/O, which depends on IRQ injection and foreign memory accessing. They are heavy in L1 VMM, because they all need interception of L0 VMM. Optimizations on them are listed as future work.

VII. RELATED WORK

Nested virtualization (A.K.A recursive virtualization) has a history of more than 30 years. In 1976, the Kernelized VM/370 was able to run a VMM recursively in a virtual machine but suffered from performance[28]. A study by Hugh et al.[12] proposes a computer system with recursive virtual machine architecture, whose central idea is the ability of any process to define a new virtual memory within its own virtual memory. Based on this idea, Bryan et al.[20] use the micro-kernel to propose a novel approach to develop a software-based virtualizable architecture called Fluke. Fluke allows recursive virtual machine, and can easily deploy arbitrary level of nested virtual machines.

Blue Pill[32] is targeted for security in Windows. It is a thin VMM to control the OS and is responsible for controlling "interesting" events inside the guest OS. Nested virtualization is one of the features it supports, and is implemented on AMD SVM. IBM z/VM[33] VMM also supports running a nested z/VM OS, but is intended only for testing purposes, and do not care much about the performance[34].

The turtles project[35] is a recent solution for nested virtualization. It has a different idea from us. It multiplexes multiple levels of virtualization into one level on CPU virtualization. On memory virtualization, it uses an idea of multi-dimensional page table. Compare to their evaluation, NestCloud get a similar performance overhead.

To make virtualization much easier and faster, lots of studies have been performed in both software fields[17], [18], [36], [37], [38] and hardware fields[11], [15], [39], but they do not address efficiency of nested virtualization.

VIII. CONCLUSIONS AND FUTURE WORK

Nested virtualization can be used in several usage models such as debugging and live migration. In this paper we present the design, implementation and evaluation of NestCloud, a three-level nested virtualization architecture for practical high performance nested virtualization. We have minimized the overhead caused by the additional level by three optimizations. The evaluation demonstrates that the implementation of NestCloud introduces 5.22% overhead on CPU and 5.69% overhead on memory, and is close to a conventional one.

The I/O performance of NestCloud is relatively low compared to a conventional guest, and optimizing it is the most relevant future work. I/O virtualization bypassing which bypasses an I/O device in L1 VMM to L0 VMM is a potential optimization. Direct access to I/O devices for L2 Guests can also be a solution. In addition, the support of SMP is another future work, which needs to deal with problems such as vCPU migration. The live migration of L2 Guest to other L1 VMM and L0 VMM on the same physical machine is also an interesting future work.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grant No. 61170050).

REFERENCES

[1] R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, 1974.
[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[3] ——, "Above the clouds: A berkeley view of cloud computing," Tech. Rep., Feb. 2009.

[4] "Vmware," www.vmware.com.

[5] "Microsoft hyper-v," http://www.microsoft.com/hyper-v-server/.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, October 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[7] "Kvm," http://www.linux-kvm.org/.

[8] K. Avi, "Kvm : The linux virtual machine monitor," *Proceedings of the Ottawa Linux Symposium*, 2007. [Online]. Available: http://ci.nii.ac.jp/naid/10024661906/en/

[9] "Virtualbox," http://www.virtualbox.org/.

[10] R. Russell, "lguest: Implementing the little linux hypervisor," *Proceedings of the Ottawa Linux Symposium*, pp. 173–177, 2007.

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

[12] H. C. Lauer and D. Wyeth, "A recursive virtual machine architecture," in *Proceedings of the workshop on virtual computer systems*. New York, NY, USA: ACM, 1973, pp. 113–116. [Online]. Available: http://doi.acm.org/10.1145/800122.803951

[13] "Windows xp mode and windows virtual pc," http://www.microsoft.com/windows/virtual-pc/default.aspx.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251203.1251223

[15] AMD, *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005. [Online]. Available: http://www.mimuw.edu.pl/ vincent/lecture6/sources/amd-pacifica-specification.pdf

[16] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 24–24. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247360.1247384

[17] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006, pp. 2–2. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267359.1267361

[18] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://portal.acm.org/citation.cfm?id=1404014.1404017

[19] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168860

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernel meet recursive virtual machines," in *USENIX 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996. [Online]. Available: http://eprints.kfupm.edu.sa/50691/

[21] "Kvm nested virtualization in the works," http://www.linux-kvm.com/content/kvm-nested-virtualization-works.

[22] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247360.1247401

[23] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[24] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao, "Quantitative comparison of Xen and KVM," in *Xen summit*. Berkeley, CA, USA: USENIX association, Jun. 2008.

[25] "Virtualization system including a virtual machine monitor for a computer with a segmented architecture," U.S. Patent US 6,397,242 B1, 2002.

[26] "Spec cpu 2006," http://www.spec.org/cpu2006/.

[27] D. Ye, J. Ray, and D. Kaeli, "Characterization of file i/o activity for spec cpu2006," *SIGARCH Comput. Archit. News*, vol. 35, pp. 112–117, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241622

[28] "Sysbench," http://sysbench.sourceforge.net/docs/.

[29] J. L. Henning, "Spec cpu2006 memory footprint," *SIGARCH Comput. Archit. News*, vol. 35, pp. 84–89, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241618

[30] "Performance evaluation of intel ept hardware assist," http://www.vmware.com/resources/techresources/10006.

[31] J. L. Henning, "Performance counters and development of spec cpu2006," *SIGARCH Comput. Archit. News*, vol. 35, pp. 118–121, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241623

[32] J. Rutkowska, "Introducing blue pill," Tech. Rep., 2006. [Online]. Available: http://theinvisiblethings.blogspot.com/2006/06/introducingblue-pill.html

[33] "Ibm z/vm," http://www.vm.ibm.com/pubs/hcsf8b22.pdf.

[34] "Ibm virtual machine facility /370: Release 2 planning guide," IBM Corporation, Tech. Rep. GC20-1814-0, 1973.

[35] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: design and implementation of nested virtualization," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://portal.acm.org/citation.cfm?id=1924943.1924973

[36] G. Liao, D. Guo, L. Bhuyan, and S. R. King, "Software techniques to improve virtualized i/o performance on multi-core systems," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: ACM, 2008, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477971

[37] D. Guo, G. Liao, and L. N. Bhuyan, "Performance characterization and cache-aware core scheduling in a virtualized multi-core server under 10gbe," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 168–177. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306784

[38] Y. Dong, X. Zheng, X. Zhang, J. Dai, J. Li, X. Li, G. Zhai, and H. Guan, "Improving virtualization performance and scalability with advanced hardware accelerations," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, dec. 2010, pp. 1 –10.

[39] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, jan. 2010, pp. 1 –10.