Nested VMX

Overview

On Intel processors, KVM uses Intel's VMX (Virtual-Machine eXtensions) to easily and efficiently run guest operating systems. Normally, these guests *cannot* themselves be hypervisors running their own guests, because in VMX, guests cannot use VMX instructions.

The "Nested VMX" feature adds this missing capability - of running guest hypervisors (which use VMX) with their own nested guests. It does so by allowing a guest to use VMX instructions, and correctly and efficiently emulating them using the single level of VMX available in the hardware.

We describe in much greater detail the theory behind the nested VMX feature, its implementation and its performance characteristics, in the OSDI 2010 paper "The Turtles Project: Design and Implementation of Nested Virtualization", available at:

https://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf

Terminology

Single-level virtualization has two levels - the host (KVM) and the guests. In nested virtualization, we have three levels: The host (KVM), which we call L0, the guest hypervisor, which we call L1, and its nested guest, which we call L2.

Running nested VMX

The nested VMX feature is enabled by default since Linux kernel v4.20. For older Linux kernel, it can be enabled by giving the "nested=1" option to the kvm-intel module.

No modifications are required to user space (qemu). However, qemu's default emulated CPU type (qemu64) does not list the "VMX" CPU feature, so it must be explicitly enabled, by giving qemu one of the following options:

- cpu host (emulated CPU has all features of the real CPU)
- cpu qemu64,+vmx (add just the vmx feature to a named CPU type)

ABIs

Nested VMX aims to present a standard and (eventually) fully-functional VMX implementation for the a guest hypervisor to use. As such, the official specification of the ABI that it provides is Intel's VMX specification, namely volume 3B of their "Intel 64 and IA-32 Architectures Software Developer's Manual". Not all of VMX's features are currently fully supported, but the goal is to eventually support them all, starting with the VMX features which are used in practice by popular hypervisors (KVM and others).

As a VMX implementation, nested VMX presents a VMCS structure to L1. As mandated by the spec, other than the two fields revision_id and abort, this structure is *opaque* to its user, who is not supposed to know or care about its internal structure. Rather, the structure is accessed through the VMREAD and VMWRITE instructions. Still, for debugging purposes, KVM developers might be interested to know the internals of this structure; This is struct vmcs12 from arch/x86/kvm/vmx.c.

The name "vmcs12" refers to the VMCS that L1 builds for L2. In the code we also have "vmcs01", the VMCS that L0 built for L1, and "vmcs02" is the VMCS which L0 builds to actually run L2 - how this is done is explained in the aforementioned paper.

For convenience, we repeat the content of struct vmcs12 here. If the internals of this structure changes, this can break live migration across KVM versions. VMCS12_REVISION (from vmx.c) should be changed if struct vmcs12 or its inner struct shadow_vmcs is ever changed.

```
typedef u64 natural_width;
struct __packed vmcs12 {
        /* According to the Intel spec, a VMCS region must start with
         * these two user-visible fields */
        u32 revision_id;
        u32 abort;

        u32 launch_state; /* set to 0 by VMCLEAR, to 1 by VMLAUNCH */
        u32 padding[7]; /* room for future expansion */

        u64 io_bitmap_a;
        u64 io_bitmap_b;
        u64 msr_bitmap;
        u64 vm_exit_msr_store_addr;
        u64 vm_exit_msr_load_addr;
        u64 vm_entry_msr_load_addr;
        u64 tsc_offset;
        u64 virtual_apic_page_addr;
        u64 apic_access_addr;
```

```
u64 ept_pointer;
u64 guest_physical_address;
u64 vmcs_link_pointer;
u64 guest_ia32_debugctl;
u64 guest_ia32_pat;
u64 guest_ia32_efer;
u64 guest_pdptr0;
u64 guest_pdptr1;
u64 guest_pdptr2;
u64 guest_pdptr3;
u64 host_ia32_pat;
u64 host_ia32_efer;
u64 padding64[8]; /* room for future expansion */
natural_width cr0_guest_host_mask;
natural_width cr4_guest_host_mask;
natural_width cr0_read_shadow;
natural_width cr4_read_shadow;
natural_width dead_space[4]; /* Last remnants of cr3_target_value[0-3]. */
natural_width exit_qualification;
natural_width guest_linear_address;
natural_width guest_cr0;
natural_width guest_cr3;
natural_width guest_cr4;
natural_width guest_es_base;
natural_width guest_cs_base;
natural_width guest_ss_base;
natural_width guest_ds_base;
natural_width guest_fs_base;
natural_width guest_gs_base;
natural_width guest_ldtr_base;
natural_width guest_tr_base;
natural_width guest_gdtr_base;
natural_width guest_idtr_base;
natural_width guest_dr7;
natural_width guest_rsp;
natural_width guest_rip;
natural_width guest_rflags;
natural_width guest_pending_dbg_exceptions;
natural_width guest_sysenter_esp;
natural_width guest_sysenter_eip;
natural_width host_cr0;
natural_width host_cr3;
natural_width host_cr4;
natural_width host_fs_base;
natural_width host_gs_base;
natural_width host_tr_base;
natural_width host_gdtr_base;
natural_width host_idtr_base;
natural_width host_ia32_sysenter_esp;
natural_width host_ia32_sysenter_eip;
natural_width host_rsp;
natural_width host_rip;
natural_width paddingl[8]; /* room for future expansion */
u32 pin_based_vm_exec_control;
u32 cpu_based_vm_exec_control;
u32 exception_bitmap;
u32 page_fault_error_code_mask;
u32 page_fault_error_code_match;
u32 cr3_target_count;
u32 vm_exit_controls;
u32 vm_exit_msr_store_count;
u32 vm_exit_msr_load_count;
u32 vm_entry_controls;
u32 vm_entry_msr_load_count;
u32 vm_entry_intr_info_field;
u32 vm_entry_exception_error_code;
u32 vm_entry_instruction_len;
u32 tpr_threshold;
u32 secondary_vm_exec_control;
u32 vm_instruction_error;
u32 vm_exit_reason;
u32 vm_exit_intr_info;
u32 vm_exit_intr_error_code;
u32 idt_vectoring_info_field;
u32 idt_vectoring_error_code;
u32 vm_exit_instruction_len;
u32 vmx_instruction_info;
u32 guest_es_limit;
u32 guest_cs_limit;
u32 guest_ss_limit;
u32 guest_ds_limit;
```

```
        u32 guest_fs_limit;
        u32 guest_gs_limit;
        u32 guest_ldtr_limit;
        u32 guest_tr_limit;
        u32 guest_gdtr_limit;
        u32 guest_idtr_limit;
        u32 guest_es_ar_bytes;
        u32 guest_cs_ar_bytes;
        u32 guest_ss_ar_bytes;
        u32 guest_ds_ar_bytes;
        u32 guest_fs_ar_bytes;
        u32 guest_gs_ar_bytes;
        u32 guest_ldtr_ar_bytes;
        u32 guest_tr_ar_bytes;
        u32 guest_interruptibility_info;
        u32 guest_activity_state;
        u32 guest_sysenter_cs;
        u32 host_ia32_sysenter_cs;
        u32 padding32[8]; /* room for future expansion */
        u16 virtual_processor_id;
        u16 guest_es_selector;
        u16 guest_cs_selector;
        u16 guest_ss_selector;
        u16 guest_ds_selector;
        u16 guest_fs_selector;
        u16 guest_gs_selector;
        u16 guest_ldtr_selector;
        u16 guest_tr_selector;
        u16 host_es_selector;
        u16 host_cs_selector;
        u16 host_ss_selector;
        u16 host_ds_selector;
        u16 host_fs_selector;
        u16 host_gs_selector;
        u16 host_tr_selector;
};
```

Authors

These patches were written by:

- Abel Gordon, abelg <at> il.ibm.com
- Nadav Har'El, nyh <at> il.ibm.com
- Orit Wasserman, oritw <at> il.ibm.com
- Ben-Ami Yassor, benami <at> il.ibm.com
- Muli Ben-Yehuda, muli <at> il.ibm.com

With contributions by:

- Anthony Liguori, aliguori <at> us.ibm.com
- Mike Day, mdday <at> us.ibm.com
- Michael Factor, factor <at> il.ibm.com
- Zvi Dubitzky, dubi <at> il.ibm.com

And valuable reviews by:

- Avi Kivity, avi <at> redhat.com
- Gleb Natapov, gleb <at> redhat.com
- Marcelo Tosatti, mtosatti <at> redhat.com
- Kevin Tian, kevin.tian <at> intel.com
- and others.