Nested Paging in bhyve

Neel Natu
The FreeBSD Project
neel@freebsd.org

Peter Grehan
The FreeBSD Project
grehan@freebsd.org

Abstract

Nested paging is a hardware technique used to reduce the overhead of memory virtualization. Specifically, this refers to Intel EPT (Extended Page Tables) and AMD NPT (Nested Page Tables). Nested paging support is available in bhyve starting from FreeBSD [1] 10.0 and provides useful features such as transparent superpages and overprovisioning of guest memory. This paper describes the design and implementation of nested paging in bhyve.

1 Introduction

Intel and AMD have introduced extensions to the x86 architecture that provide hardware support for virtual machines, viz.

- Intel Virtual-Machine Extensions (VMX) [2, 3]
- AMD Secure Virtual Machine (SVM) [4]

The first generation of VMX and SVM did not have any hardware-assist for virtualizing access to the memory management unit (MMU). Hypervisors had to make do with existing paging hardware to protect themselves and to provide isolation between virtual machines. This was typically done with a technique referred to as *shadow paging* [5].

A hypervisor would examine the guest paging structures and generate a corresponding set of paging structures in memory called *shadow page tables*. The shadow page tables would be used to translate a guest-virtual address to a host-physical address. The hypervisor would be responsible for keeping the shadow page tables synchronized with the guest page tables. This included tracking modifications to the guest page tables, handling page faults and reflecting accessed/dirty (A/D) bits from the shadow page tables to guest page tables. It was estimated that in certain workloads shadow paging could account for up to 75% of the overall hypervisor overhead [5].

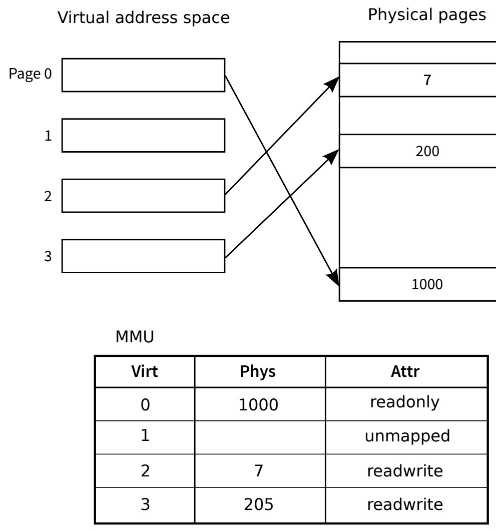
Nested page tables were introduced in the second generation of VMX and SVM to reduce the overhead in virtualizing the MMU. This feature has been shown to provide performance gains upwards of 40% for MMU-intensive benchmarks and upwards of 500% for micro-benchmarks [6].

With x86_64 page tables there are two types of addresses: virtual and physical. With nested page tables there are three types of addresses: guest-virtual, guest-physical and host-physical. The address used to access memory with x86_64 page tables is instead treated as a guest-physical address that must be translated to a host-physical address. The guest-physical to host-physical translation uses nested page tables which are similar in structure to x86_64 page tables.

bhyve has always relied on nested page tables to restrict guest access to memory, but until the nested paging work described in this paper it wasn't a well-behaved consumer of the virtual memory subsystem. All guest memory would be allocated upfront and not released until the guest was destroyed. Guest memory could not be swapped to stable storage nor was there a mechanism to track which pages had been accessed or modified[1].

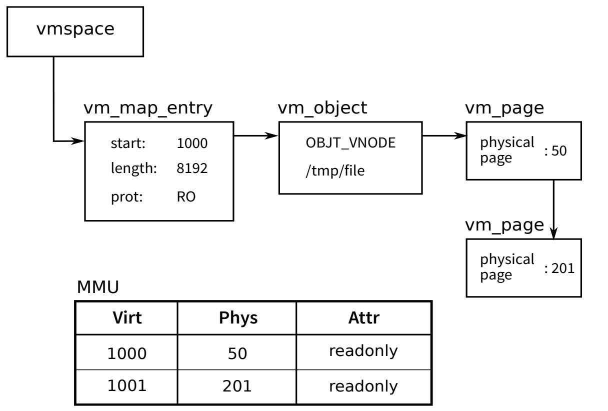
The nested paging work described in this paper allows bhyve to leverage the FreeBSD/amd64 *pmap* to maintain nested paging structures, track A/D bits and maintain TLB consistency. It also allows bhyve to represent guest memory as a FreeBSD *vmspace* and handle nested page faults in the context of this *vmspace*.

[1]Modified and dirty are used interchangeably in this paper



Virt	Phys	Attr
0	1000	readonly
1		unmapped
2	7	readwrite
3	205	readwrite

Figure 1: Memory Management Unit



Virt	Phys	Attr
1000	50	readonly
1001	201	readonly

Figure 2: mmap(/tmp/file, 8192, readonly)

The rest of the paper is organized as follows: Section 2 provides an overview of virtual memory management in FreeBSD on x86_64 processors. Section 3 describes the virtualization extensions in Intel CPUs. Section 4 introduces Intel's implementation of nested page tables. Sections 5, 6 and 7 describe the design and implementation of nested paging in bhyve. Section 8 presents results of experimental evaluation of the overhead of nested paging. Section 9 looks at opportunities to leverage nested page tables for several useful features.

2 FreeBSD virtual memory management

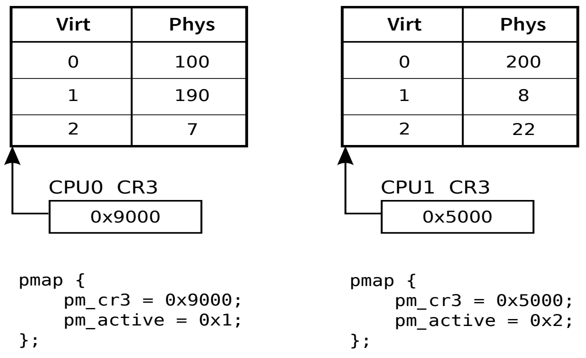
The FreeBSD virtual memory (VM) subsystem provides each process with a virtual address space. All memory references made by a process are interpreted in the context of its virtual address space. These virtual addresses are translated into physical addresses by the MMU as shown in Figure 1.

The MMU performs address translation in fixed-sized units called pages. The size of a page is machine-dependent and for the x86_64 architecture this can be 4KB, 2MB or 1GB. The MMU also protects physical pages belonging to an address space from being written to or read from a different address space. All MMU implementations allow a translation to be marked as readonly while some implementations can keep track of which pages have been read or written by the process.

The process address space in FreeBSD is represented by a *vmspace* [7]. The address space is divided into contiguous ranges such that all addresses in a range are mapped with the same protection (e.g., readonly) and source data from the same backing object (e.g., a file on disk). Each range is represented by a *vm_map_entry*. The physical memory pages provided by the backing object are mapped into the address range represented by the *vm_map_entry*. The backing object is represented by a *vm_object*. The physical memory pages associated with the backing object are represented by a *vm_page*. A *vm_page* contains the physical address of the page in system memory. This address is used by the MMU in its translation tables. Figure 2 shows the data structures involved in a readonly mapping of */tmp/file* into a process's address space.

The physical-mapping (pmap) subsystem provides machine-dependent functionality to the VM subsystem, such as:

- Creating virtual to physical mappings
- Invalidating a mapping to a physical page
- Modifying protection attributes of a mapping



Virt	Phys
0	100
1	190
2	7

Virt	Phys
0	200
1	8
2	22

```
pmap {
    pm_cr3 = 0x9000;
    pm_active = 0x1;
};
```

```
pmap {
    pm_cr3 = 0x5000;
    pm_active = 0x2;
};
```

Figure 3: pmap

- Tracking page access and modification

Each *vmspace* has an embedded *pmap*. The *pmap* contains machine-dependent information such as a pointer to the root of the page table hierarchy.

For the x86_64 architecture the pmap subsystem maintains mappings in hierarchical address-translation structures commonly referred to as *page tables*. The page tables are the machine-dependent representation of the *vmspace*. The processor's control register CR3 points to the root of the page tables.

It is important to note that multiple processors may have an address space active simultaneously. This is tracked by the *pm_active* bitmap. Figure 3 depicts a dual-processor system with a different address space active on each processor.

2.1 x86_64 address translation

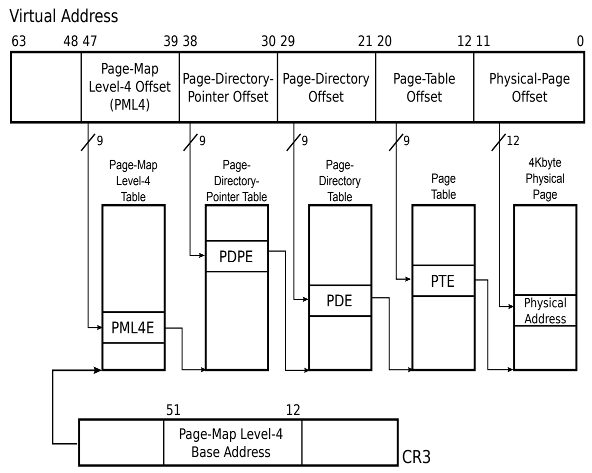


Figure 4: x86_64 address translation

A page-table-page is 4KB in size and contains 512 page-table-entries each of which is 64-bits wide. A page-table-entry (PTE) contains the physical address of the next level page-table-page or the page-frame. A page-table-entry also specifies the protection attributes, memory type information and A/D bits.

As shown in Figure 4, a 64-bit virtual address is divided into 4 fields with each field used to index into a page-table-page at different levels of the translation hierarchy:

- Bits 47:39 index into the page-map level4 table
- Bits 38:30 index into the page-directory pointer table
- Bits 29:21 index into the page-directory table
- Bits 20:12 index into the page table
- Bits 11:0 provide the offset into the page-frame

3 Intel Virtual-Machine Extensions

Intel Virtual-Machine Extensions (VMX) provide hardware support to simplify processor virtualization. This is done by introducing two new forms of processor operation: VMX root and VMX non-root.

A hypervisor runs in VMX root operation and has full control of the processor and platform resources. Guests run in VMX non-root operation which is a restricted environment.

A guest starts executing when the hypervisor executes the *vmlaunch* instruction to transition the processor into VMX non-root operation. The guest continues execution until a condition established by the hypervisor transitions the processor back into VMX root operation and resumes hypervisor execution. The hypervisor will examine the reason for the VM-exit, handle it appropriately, and resume guest execution.

The VMX transition from hypervisor to guest is a *VM-entry*. The VMX transition from guest to hypervisor is a *VM-exit*. VMX transitions and non-root operation are controlled by the *Virtual Machine Control Structure (VMCS)*. The VMCS is used to load guest processor state on VM-entry and save guest processor state on VM-exit. The VMCS also controls processor behavior in VMX non-root operation, for example to enable nested page tables. Of particular importance is the *Extended-Page-Table Pointer (EPTP)* field of the VMCS which holds the physical address of the root of the nested page tables.

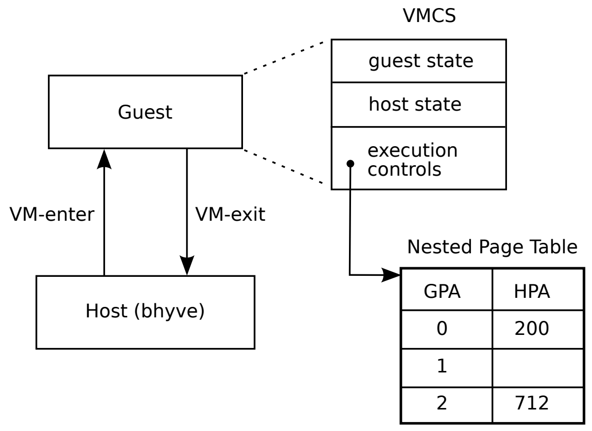


Figure 5: VMX operation

Figure 5 illustrates the VMX transitions and the nested page tables referenced from the VMCS.

4 Intel Extended Page Tables

The x86_64 page tables translate virtual addresses to physical addresses. This translation is done using page tables pointed to by CR3. In addition to mapping the virtual address to a physical address, the page tables also provide permissions and memory type associated with the mapping.

When the processor is operating in guest context and nested page tables are enabled, the physical address that is the output of x86_64 page tables is treated as a guest-physical-address. The nested page tables translate this guest-physical-address (GPA) to a host-physical-address (HPA). It is the HPA that is driven out on the processor's memory and I/O busses. This additional level of address translation allows the hypervisor to isolate the guest address space.

Note that with nested paging enabled there are two distinct page tables active simultaneously:

- x86_64 page tables pointed to by guest CR3
- nested page tables pointed to by the VMCS

Intel's implementation of nested page tables is called *Extended Page Tables (EPT)*. EPT is similar in structure and functionality to x86_64 page tables. It has same number of translation levels and it uses the the same bits to index into the page-table-pages. For example, bits 47:39 of the GPA index into the PML4 table. It also provides the same protection attributes as x86_64 page tables.

However, there are some differences.

4.1 Page-table-entry

The most obvious difference between the page-table-entries in Table 4.1 is that different bit positions are used to express the same functionality. For example, the dirty flag is bit 6 in the x86_64 PTE versus bit 9 in the EPT PTE.

Some differences arise when the x86_64 PTE has functionality that does not exist in the EPT PTE. Bit 8 in the x86_64 PTE is used to represent mappings that are global and are not flushed on an address space change. There is no corresponding bit in the EPT PTE because this functionality is not relevant in extended page tables.

The EPT PTE and x86_64 PTE also differ in their *default settings*. The execute permission must be explicitly granted in an EPT PTE whereas it must be explicitly revoked in a x86_64 PTE.

Bit	**x86_64 PTE**	**EPT PTE**
0	Valid	Read permission
1	Write permission	Write permission
2	User/Supervisor	Execute permission
3	Write-through cache	Memory type[0]
4	Cache disable	Memory type[1]
5	Accessed	Memory type[2]
6	Dirty	Ignore guest PAT
7	Page Attribute Table index	Ignored
8	Global	Accessed
9	Ignored	Dirty
61	Execute disable	Suppress #VE

Table 1: Differences between x86_64 and EPT PTEs

4.2 Capabilities

Table 4.2 highlights differences in the capabilities of x86_64 page tables and EPT page tables.

Capability	**x86_64 PTE**	**EPT PTE**
2MB mapping	Yes	Optional
A/D bits	Yes	Optional
Execute-only mapping	No	Optional

5 Design of nested paging in bhyve

The address space of a typical guest is shown in Figure 6. This guest is configured with 2GB of system memory split across two memory segments: the first segment starts at 0x00000000 and the second segment starts at 0x100000000. The region of the address space between 1GB and 4GB is called the *PCI hole* and is used for addressing *Memory-Mapped I/O (MMIO)* devices. The guest's system firmware[2] is mapped readonly in the address space just below 4GB.

The nested paging implementation in bhyve is based on the observation that the guest address space is similar to process address space:

- Guest memory segments are backed by a *vm_object* that supplies zero-filled, anonymous memory.
- Guest firmware is backed by a *vm_object* that is associated with the firmware file and is mapped readonly.

[2]The guest BIOS or UEFI image

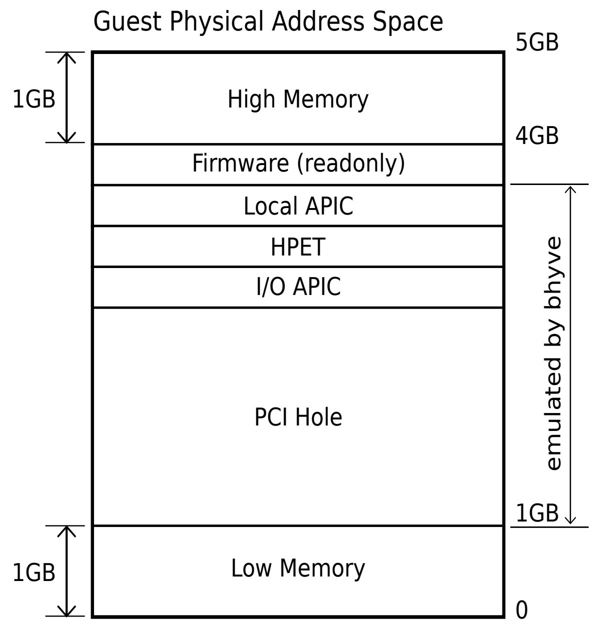


Figure 6: Guest address space

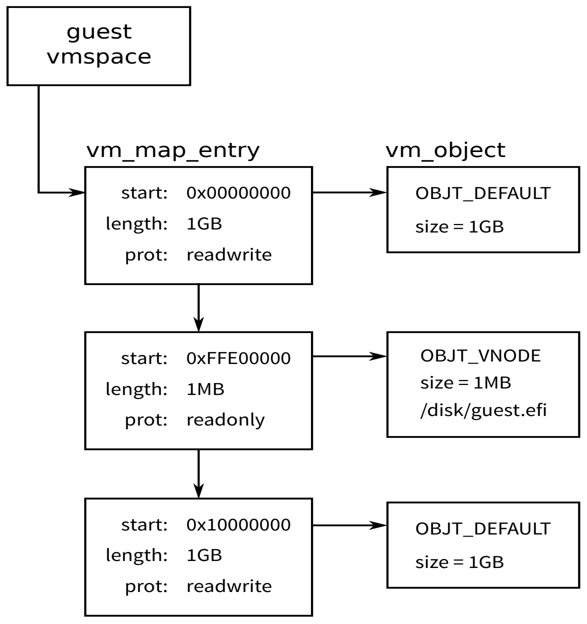


Figure 7: Guest vmspace

- The PCI hole is not mapped. Any access to it from the guest will cause a nested page fault.

Figure 7 shows the guest address space represented as a *vmspace*. The VM subsystem already had the primitives needed to represent the guest address space in this manner. However, the pmap needed modifications to support nested paging.

6 pmap modifications

The pmap subsystem is responsible for maintaining the page tables in a machine-dependent format. Given the differences between the x86_64 page tables and EPT, modifications were required to make the pmap EPT-aware.

6.1 pmap initialization

The *pmap* was identified as an x86_64 or EPT pmap by adding an enumeration for its type.

```
enum pmap_type {
    PT_X86,     /* regular x86 page tables */
    PT_EPT,     /* Intel's nested page tables */
    PT_RVI,     /* AMD's nested page tables */
};
struct pmap {
    ...
    enum pmap_type pm_type;
};
```

Prior to the nested paging changes *vmspace_alloc()* called *pmap_pinit()* to initialize the *pmap*. *vmspace_alloc()* was modified to accept a pointer to the pmap initialization function.

```
struct vmspace *
vmspace_alloc(min, max, pmap_pinit_t pinit)
{
    /* Use pmap_pinit() unless overridden by the caller */
    if (pinit == NULL)
            pinit = &pmap_pinit;
}
```

A new function *pmap_pinit_type* was added to initialize a *pmap* based on its type. In particular the pmap type is used to ensure that the kernel address space is not mapped into nested page tables.

```
int
pmap_pinit_type(pmap_t pmap, enum pmap_type type, int flags)
{
    pmap->pm_type = type;
    if (type == PT_EPT) {
        /* Initialize extended page tables */
    } else {
        /* Initialize x86_64 page tables */
    }
}

int
pmap_pinit(pmap_t pmap)
{
    return pmap_pinit_type(pmap, PT_X86, flags);
}
```

Finally the EPT pmap is created as follows.

```
int
ept_pinit(pmap_t pmap)
{
    return pmap_pinit_type(pmap, PT_EPT, flags);
}

struct vmspace *
ept_vmspace_alloc(vm_offset min, vm_offset max)
{
    return vmspace_alloc(min, max, ept_pinit);
}
```

6.2 EPT page-table-entries

Section 4.1 highlighted the differences between EPT PTEs and x86_64 PTEs. The pmap code was written to support the x86_64 page tables and used preprocessor macros to represent bit fields in the PTE.

```
#define PG_M    0x040   /* Dirty bit */
```

This would not work for nested page tables because the dirty flag is represented by bit 9 in the EPT PTE.

The bitmask is now computed at runtime depending on the pmap type.

```
#undef  PG_M
#define X86_PG_M  0x040
#define EPT_PG_M  0x200
pt_entry_t
pmap_modified_bit(pmap_t pmap)
{
    switch (pmap->pm_type) {
    case PT_X86:
        return (X86_PG_M);
    case PT_EPT:
        return (EPT_PG_M);
    }
}
```

Note that **PG_M** is now undefined to force compilation errors if used inadvertently. Functions that used **PG_M** were modified as follows:

```
void
some_pmap_func(pmap_t pmap)
{
    pt_entry_t PG_M = pmap_modified_bit(pmap);
    /* Rest of the function does not change */
}
```

The same technique was used for all fields in the EPT PTE that are different from the x86_64 PTE with the exception of **PG_U**. Section 6.3 discusses the special treatment given to **PG_U**.

6.3 EPT execute permission

bhyve has no visibility into how the guest uses its address space and therefore needs to map all guest memory with execute permission. An EPT mapping is executable if the **EPT_PG_EXECUTE** field at bit 2 is set in the PTE.

PG_U in the x86_64 PTE represents whether the mapping can be accessed in user-mode. **PG_U** is at bit 2 in the x86_64 PTE. The pmap sets **PG_U** if the address mapped by the PTE is in the range [0, VM_MAXUSER_-ADDRESS).

The guest address space is in the same numerical range as the user address space i.e., both address spaces start at 0 and grow upwards [3]. From pmap's perspective, mappings in the guest address space are considered user mappings and **PG_U** is set. However, bit 2 is interpreted as **EPT_PG_EXECUTE** in the EPT context. This has the desired effect of mapping guest memory with execute permission.

Note that the guest still retains the ability to make its mappings not executable by setting the **PG_NX** bit in its PTE.

6.4 EPT capabilities

The original pmap implementation assumed MMU support for 2MB superpages and A/D bits in the PTE. However these features are optional in an EPT implementation.

The *pm_flags* field was added to the *pmap* to record capabilities of the EPT implementation.

```
#define PMAP_PDE_SUPERPAGE      (1 << 0)
#define PMAP_EMULATE_AD_BITS    (1 << 1)
#define PMAP_SUPPORTS_EXEC_ONLY (1 << 2)
struct pmap {
    ...
    int    pm_flags;
}
```

A **PT_X86** pmap sets *pm_flags* to **PMAP_PDE_-SUPERPAGE** unconditionally. A **PT_EPT** pmap sets *pm_flags* based on EPT capabilities advertised by the processor in a model specific register.

The pmap already had code to disable superpage promotion globally and it was trivial to extend it to check for **PMAP_PDE_SUPERPAGE** in *pm_flags*.

6.5 EPT A/D bit emulation

The x86_64 page tables keep track of whether a mapping has been accessed or modified using the **PG_A** and **PG_M** bits in the PTE.

[3] VM_MAXUSER_ADDRESS implies an upper limit of 128TB on guest physical memory

The VM subsystem uses the accessed bit to maintain the activity count for the page. The dirty bit is used to determine whether the page needs to be committed to stable storage. It is important to faithfully emulate the A/D bits in EPT implementations that don't support them[4].

A straightforward approach would be to assign unused bits in the EPT PTE to represent the A/D bits. Dirty bit emulation was done by making the mapping read-only and setting the emulated **PG_M** bit on a write fault. Accessed bit emulation was done by removing the mapping and setting the emulated **PG_A** bit on a read fault.

Accessed bit emulation required the mapping to be entirely removed from the page tables with it being reinstated through *vm_fault()*. Dirty bit emulation required differentiating between true-readonly mappings versus pseudo-readonly mappings used to trigger write faults. The code to implement this scheme required extensive modifications to the pmap subsystem [8].

A simpler solution is to interpret the relevant bits in the EPT PTE as follows [9]: **PG_V** and **PG_RW** are now assigned to unused bits in the EPT PTE. On the other hand **PG_A** maps to **EPT_PG_READ** and **PG_M** maps to **EPT_PG_WRITE** which are interpreted by the MMU as permission bits.

	PTE bit	Interpreted by
PG_V	52	A/D emulation handler
PG_RW	53	A/D emulation handler
PG_A	0	MMU as **EPT_PG_READ**
PG_M	1	MMU as **EPT_PG_WRITE**

Clearing the accessed bit removes read permission to the page in hardware. Similarly, clearing the modified bit removes write permission to the page in hardware. In both cases the rest of the PTE remains intact. Thus, the A/D bit emulation handler can inspect **PG_V** and **PG_RW** in the PTE and handle the fault accordingly.

The A/D bit emulation handler can resolve the following types of faults:

- Read fault on 4KB and 2MB mappings
- Write fault on 4KB mappings

The handler will attempt to promote a 4KB mapping to a 2MB mapping. It does not handle write faults on 2MB mappings because the pmap enforces that if a superpage is writeable then its **PG_M** bit must also be set [10].

[4]Hardware support for A/D bits in EPT first appeared in the Haswell microarchitecture

6.5.1 EPT PTE restrictions:

There is an additional issue with clearing the emulated **PG_A**. Recall that clearing the emulated **PG_A** actually clears **EPT_PG_READ** and makes the mapping not readable.

The MMU requires that if the PTE is not readable then:

- it cannot be writeable
- it cannot be executable unless the MMU supports execute-only mappings

These restrictions cause pessimistic side-effects when the emulated **PG_A** is cleared. Writeable mappings will be removed entirely after superpage demotion if appropriate. Executable mappings suffer the same fate unless execute-only mappings are allowed.

6.6 EPT TLB invalidation

The *Translation Lookaside Buffer (TLB)* is used to cache frequently used address translations. The pmap subsystem is responsible for invalidating the TLB when mappings in the page tables are modified or removed.

To facilitate this a new field was added to *pmap* called *pm_eptgen*. This field is incremented for every TLB invalidation request. A copy of the generation number is also cached in the virtual machine context as *eptgen*. Just prior to entering the guest, *eptgen* is compared to *pm_eptgen*, and if they are not equal the EPT mappings are invalidated from the TLB.

The *pm_active* bitmap is used to track the cpus on which the guest address space is active. The bit corresponding to the physical cpu is set by bhyve on a VM-entry and cleared on a VM-exit. If the *pm_active* field indicates that the nested pmap is in use on other cpus, an *Inter-Processor Interrupt (IPI)* is issued to those cpus. The IPI will trigger a VM-exit and the next VM-entry will invalidate the TLB as previously described.

```
struct pmap {
    ...
    long pm_eptgen; /* EPT pmap generation */
};

struct vmx {
    ...
    long eptgen[MAXCPU]; /* cached pm_eptgen */
};
```

7 bhyve modifications

bhyve has always relied on nested page tables to assign memory to a guest. Prior to this work, guest memory was allocated when the virtual machine was created and released when it was destroyed. Guest memory could be accessed at all times without triggering a fault.

Representing guest memory as a *vm_object* meant that guest memory pages could be paged out or mapped read-only by the VM subsystem. This required bhyve to handle nested page faults. Additionally guest memory could be accessed only after ensuring that the underlying *vm_page* was resident.

7.1 Guest memory segments

A guest memory segment corresponds to a *vm_map_entry* backed by a *vm_object* of type **OBJT_DEFAULT** as depicted in Figure 7. Each memory segment is backed by zero-filled, anonymous memory that is allocated on-demand and can be paged out.

7.2 EPT-violation VM-exit

If the translation for a guest physical address is not present or has insufficient privileges then it triggers an EPT-violation VM-exit. The VMCS provides collateral information such as the GPA and the access type (e.g., read or write) associated with the EPT-violation.

If the GPA is contained within the virtual machine's memory segments then the VM-exit is a *nested page fault*, otherwise it is an *instruction emulation fault*.

7.2.1 Nested page fault

The nested page fault handler first calls into the pmap to do A/D bit emulation. If the fault was not triggered by A/D bit emulation, it is resolved by *vm_fault()* in the context of the guest vmspace.

Event re-injection A hypervisor can inject events (e.g., interrupts) into the guest using a VM-entry control in the VMCS. It is possible that the MMU could encounter a nested page fault when it is trying to inject the event. For example, the guest physical page containing the interrupt descriptor table (IDT) might be swapped out.

The hypervisor needs to recognize that a nested page fault occurred during event injection and re-inject the event on the subsequent VM entry.

It is now trivial to verify correct behavior of bhyve in this scenario by calling *pmap_remove()* to invalidate all guest physical mappings from the EPT.

7.2.2 Instruction emulation fault

An instruction emulation fault is triggered when the guest accesses a virtual MMIO device such as the local APIC. To handle this type of fault bhyve has to fetch the instruction that triggered the fault before it can be emulated. Fetching the instruction requires walking the guest's page tables. Thus bhyve needs to be able to access guest memory without triggering a page fault in kernel mode.

This requirement was satisfied by using an existing VM function: *vm_fault_quick_hold_pages()*. This function returns the *vm_page* associated with the GPA and also prevents the *vm_page* from being freed by the page daemon. *vm_gpa_hold()* and *vm_gpa_release()* in bhyve are the convenience wrappers on top of this.

vm_fault_hold() and superpages The original implementation of *vm_gpa_hold()* called *vm_fault_hold()*.

vm_fault_hold() resolved the GPA to a *vm_page* and called *pmap_enter()* to install it in the page tables. If there was already a superpage mapping for the GPA then it would get demoted, the *new* mapping would be installed and then get promoted *immediately*. This resulted in an inordinate number of superpage promotions and demotions.

7.3 PCI passthrough

bhyve supports PCI passthrough so a guest can directly access a physical PCI device. There were two memory-related issues that needed to be addressed with PCI passthrough.

7.3.1 MMIO BARs

Most PCI devices implement a set of registers to control operation and monitor status. These registers are mapped into MMIO space by programming the device's Base Address Register (BAR). The PCI device only responds to accesses that fall within the address range programmed in its BAR(s).

For the guest to get direct access to a PCI device the physical BAR has to be mapped into the guest's address space. This mapping is represented by a memory segment that is backed by a *vm_object* of type **OBJT_SG**. These mappings are *unmanaged* and they do not get paged out or promoted to superpages.

7.3.2 Direct memory access

A PCI device has the ability to access system memory independently of the processor. This is commonly referred to as Direct Memory Access (DMA). A PCI passthrough device is assigned to a guest and is programmed with guest physical addresses.

This implies that bhyve needs to install the GPA to HPA translation not just in the EPT but also in the I/O Memory Management Unit (IOMMU).

Additionally bhyve needs to ensure that the memory backing the guest address space is never paged out because the current generation of platforms cannot handle I/O page faults. This is implemented by calling *vm_map_wire()* on all memory segments.

7.4 Tunables, sysctls and counters

The following tunables can be used to influence the EPT features used by bhyve:

- hw.vmm.ept.use_superpages: 0 disables superpages
- hw.vmm.ept.use_hw_ad_bits: 0 forces A/D bit emulation
- hw.vmm.ept.use_exec_only: 0 disables execute-only mappings

The following sysctls provide nested pmap information:

- hw.vmm.ipinum: IPI vector used to trigger EPT TLB invalidation
- hw.vmm.ept_pmap_flags: *pm_flags* field in the pmap
- vm.pmap.num_dirty_emulations: count of dirty bit emulations
- vm.pmap.num_accessed_emulations: count of accessed bit emulations
- vm.pmap.num_superpage_accessed_emulations: count of accessed bit emulations for superpages
- vm.pmap.ad_emulation_superpage_promotions: superpage promotions attempted by the A/D bit emulation handler

The following bhyvectl counters[5] are available per vpcu:

- Number of nested pages faults
- Number of instruction emulation faults

8 Performance

The experiments described in this section were performed on a system with 32GB RAM and a Xeon E3-1220 v3 CPU at 3.10GHz. The host and the guest were both running FreeBSD/amd64 10.0-RELEASE.

[5]/usr/sbin/bhyvectl –get-stats

8.1 Nested paging overhead

In this experiment the guest was assigned 2 vcpus and 8GB memory. The vcpus were pinned to minimize scheduling artifacts. The host memory was overprovisioned to eliminate paging artifacts.

The user, system and wall-clock times for *make -j4 buildworld* in the guest were measured. The results are summarized in Table 2.

Guest memory	**User**	**System**	**Wall-clock**
Wired	3696	389	2207
Not wired	3704	409	2225
Not wired, A/D bit emulation	3784	432	2276

Table 2: Guest buildworld times in seconds

The buildworld time with guest memory wired established the baseline of 2207 seconds (i.e., nested paging disabled).

The buildworld took 18 seconds longer when guest memory was not wired and an additional 51 seconds with A/D bits emulated in software.

8.2 GUPS

Giga-updates per second (GUPS) is a measurement of how frequently a computer can issue updates to randomly generated memory locations [11].

In this experiment the guest was assigned 1 vcpu and 24GB memory. GUPS was configured with a 12GB working set and CPU time was measured.

Guest superpages	**Host superpages**	**CPU time in seconds**
Disabled	Disabled	500
Disabled	Enabled	258
Enabled	Disabled	267
Enabled	Enabled	102

Table 3: Effect of superpages on GUPS CPU time

Table 3 demonstrates the benefit of transparent superpage support in nested paging.

9 Future work

Nested paging is a foundational technology that will influence the design of upcoming features in bhyve.

The *vm_page* activity count and modified state may be used in live migration to compute the order in which guest memory is migrated.

Guest memory could be backed by a file which would be useful when suspending a virtual machine to disk.

10 Acknowledgements

Alan Cox and Konstantin Belousov provided guidance on nested page table support in pmap. John Baldwin reviewed the use of scatter-gather VM objects for PCI passthrough. Peter Holm tested the nested paging patch before it was committed to FreeBSD. Leon Dang did the artwork in this paper. We wish to thank them all.

References

[1] *The FreeBSD Project* `http://www.freebsd.org`

[2] *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization* Intel Technology Journal, Volume 10, Issue 3

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual*

[4] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*

[5] *AMD-V Nested Paging White Paper* `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`

[6] *Performance Evaluation of AMD RVI Hardware Assist* `http://www.vmware.com/pdf/RVI_performance.pdf`

[7] *The Design and Implementation of the FreeBSD Operating System* Marshall Kirk McKusick, George V. Neville-Neil

[8] *FreeBSD svn revision 254317*

[9] *FreeBSD svn revision 255960*

[10] *Practical, transparent operating system support for superpages* Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox `http://www.usenix.org/events/osdi02/tech/full_papers/navarro/navarro.pdf`

[11] *GUPS* `http://en.wikipedia.org/wiki/Giga-updates_per_second`