X86-64 XenLinux: Architecture, Implementation, and Optimizations

Jun Nakajima, Asit Mallick
Intel Open Source Technology Center
jun.nakajima@intel.com, asit.k.mallick@intel.com

Ian Pratt, Keir Fraser
University of Cambridge
{first.last}@cl.cam.ac.uk

Abstract

Xen 3.0 has been officially released with x86-64 support added. In this paper, we discuss the architecture, design decisions, and various challenging issues we needed to solve when we para-virtualized x86-64 Linux.

Although we reused the para-virtualization techniques and code employed by x86(-32) XenLinux as much as possible, there are notable differences between x86 XenLinux and x86-64 XenLinux. Because of the limited segmentation with x86-64, for example, we needed to run both the guest kernel and applications in ring 3, raising the problem of protecting one from the other. This also complicated system calls handling, event handling, including exceptions such as page faults and interrupts. For example the native device drivers run in Ring 3 in x86-64 XenLinux today.

Xen itself was required to extend to support x86-64 XenLinux. To handle transitions between kernel and user mode securely, for example, Xen is aware of the mode of the guests controlling the page tables used for each mode. We also discuss other extensions to x86 XenLinux, in support of x86-64, including page table management, 4-level writable page tables, shadow page tables for live migration, new hypercalls, and DMA.

We also discuss the performance optimizations techniques used today, and also discuss how to overcome the overheads caused by the transitions between user and kernel mode.

1 Introduction

1.1 Full virtualization and para virtualization

x84-64 XenLinux is a para-virtualized version of x86-64 Linux ported to the x86-64 Xen. Although the Linux kernel is modified, no modifications are required to user space, i.e. existing binaries and operating system distributions work without modification.

Xen 3.0 supports both para-virtualization and hardware-based full virtualization. Para-virtualization on Xen was designed to achieve high performance, and it requires modifications

to the guest operating system to work with the platform interface provided by Xen. In other words, Xen requires the *porting* of guest operating systems to the Xen Interface, to exploit para-virtualization.

The alternative, full-virtualization, on contrary to para-virtualization, no modifications to the guest operating systems are required, but instead it requires to provide the guest operating systems with an illusion of a complete virtual platform seen within a virtual machine behavior same as a standard PC/server platform.

1.2 Para-virtualization of Linux

Para-virtualization of Linux means a task of modifying the Linux kernel code to run it on the virtual platform provided by Xen. The virtual platform is defined by the Xen Interface (See [3] for the detail).

Para-virtualized Linux, i.e. XenLinux does not need to run on a standard PC/server platform, but on a virtual platform with virtual CPU provided by Xen. The areas of such modifications are mostly low-level CPU-dependent code, initialization code, and platform specific code.

1.3 Para-virtualization using VMI

Compared to Xen Interface, VMware's VMI [1] is closer to the instruction level. The idea is "the closer the API resembles a native platform which the OS supports, the lower the cost of porting." However, this layer can be legacy when hardware-based virtualization is broadly available in the near future.

In addition, the current VMI does not have high-level interface API for the other virtualization-related resources, such as interrupt controllers (which Xen obviates), time, virtual block, network devices, and virtual TPM.

2 Xen Interface for x86-64

Xen Interface for x86-64 is mostly common with x86-32. In this section, we briefly explain the abstraction provided by Xen 3.0 to describe the scope of the modifications required for Linux.

Part of Xen Interface is provided by **hypercalls**. The hypercall interface allows domains to perform executive procedures in Xen running at privilege level 0. The other part is provided via the data structures available to domains.

2.1 Virtual CPU Architecture

- CPU state – The critical difference at initialization time between the native x86-64 Linux and x86-64 XenLinux is that the latter is set to run in **the 64-bit mode** (with paging enabled) at initialization when the virtual machine, i.e. domain in Xen is built. The CPU in guests run at privilege level 3. The virtual address preestablished for the guest kernel at initialization time is minimum, and guests need to extend or create new translation as necessary.

- Floating point registers – Xen allows guests to use the lazy save and restore techinique. The operation clear, set CR0.TS are simply replaced with `fpu_taskswitch`(0), `fpu_taskswitch`(1), respectively.

- Exceptions – The IDT is virtualized as a simple trap table, and the hypercall `set_trap_table` is used to register the set of the handlers with Xen upon exceptions, such as #PF (page fault).

- Interrupts and events – External interrupts are virtualized by mapping them

to event channels, which are delivered asynchronously to the target domain using a callback supplied via the `set_callbacks` hypercall.

2.1.1 Tickless in idle

Xen allows guests to implement 'tickless mode' on idle CPU. The hypercall `set_timer_op` is used to request that they receive a timer event sent at a specified system time.

2.2 Memory

Xen is responsible for managing the allocation of physical memory to domains, and the guest physical memory is virtualized as "pseudo-physical memory".

On a real system, E820 BIOS call typically reports the memory map, but the equivalent information is provided simply by "start info page" (`start_info.nr_pages`) on guests on Xen. The pointer to start info page is set by Xen (for domain 0) or the domain builder (otherwise) to the register `%rsi`. See Figure 1 for the fields in details.

The memory given to a domain is a single contiguous region of pseudo-physical memory. Each domain is supplied with a physical-to-machine table, and `start_info.mfn_list` points to the physical page number.

2.3 Writable Page Tables

In the default mode of operation, Xen provides "writable page tables", in which guests have the illusion that their page tables are directly writable.

At this point, the lowest level, i.e. page tables (L1) are handled this way. The higher levels, including PML4, page directory pointers, page directories are updated by the hyercall `mmu_update`. Updates to those entries are much less frequent compared to page tables.

3 The x86-64 XenLinux Architecture

The architecture of the x86-64 XenLinux should be same as the x86-32 XenLinux in general. See [2] for an overview of the Xen 3.0 architecture. In this section, we discuss x86-64 specific requirements and extensions.

3.1 x86-64 specific requirements

As described in Section 2.1, on x86-64 systems it is not architecturally possible to protect Xen from untrusted guest code running in privilege levels 1 and 2. Guests are therefore restricted to run in privilege level 3 only. The guest kernel is protected from its applications by context switching between the kernel and currently running application.

The other issue is the SWAPGS instruction. SWAPGS is intended for use with fast system calls when in 64-bit mode to allow immediate access to kernel structures on transition to kernel mode. The native x86-64 Linux uses PDA (Per processor data structure) to maintain critical data such as the pointer to the current process, the top of kernel stack for the current process, and user `%rsp` for system call, TLB state, and etc. The register `%gs` points to the area in the kernel mode, and the instruction SWAPGS is executed when the processor enters or exits from the kernel mode.

```
typedef struct start_info {
  char magic[32];            /* "xen-<version>-<platform>".          */
  unsigned long nr_pages;    /* Total pages allocated to this domain */
  ...
  unsigned long pt_base;     /* VIRTUAL address of page directory.   */
  unsigned long nr_pt_frames;/* Number of bootstrap p.t. frames.     */
  unsigned long mfn_list;    /* VIRTUAL address of page-frame list.  */
  unsigned long mod_start;   /* VIRTUAL address of pre-loaded module */
  unsigned long mod_len;     /* Size (bytes) of pre-loaded module.   */
  int8_t cmd_line[MAX_GUEST_CMDLINE];
} start_info_t;
```

Figure 1: start info page

The SWAPGS is only accessible at privilege level 0. Therefore it cannot be executed even in privilege level 1 or 2. Although we need to remove the instruction when para-vitalizing, we want to avoid to change the way the kernel uses PDA for no good reasons. This also justified the design to have the guest kernel run at privilege level 3.

We have two options for to protect the guest kernel from its applications:

1. Have two separate PML4 pages for the kernel and a user process. The one for the kernel has translation for the kernel and user, and the user one has just for the user.

2. Have a single PML4 page for both the kernel and a user process. When we switch to the user mode, we remove the translations for the kernel. When we switch back to the kernel mode, restore the kernel translations.

Since Xen must be OS agnostic and the kernel translations can be required for user processes (such as vsyscall), the first option is a cleaner option.

The current implementation uses the first one.

3.1.1 x86-64 Xen Address Space

Figure 2 shows the address map of the x86-64 Xen. As it shows, the kernel and user address spaced is separated by Xen. This is similar to the native x86-64 Linux, but the page offset of the native is 0xffff810000000000, and it is below the first address available for the guest, which is 0xffff880000000000. Thus, the page offset of x86-64 is set to 0xffff880000000000.

3.1.2 Unified system call and hypercall handling

Xen needs to intercept system calls and bounce them back to the guest kernel. The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used), and x86-64 Linux uses these. SYSCALL is, however, intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. This implies that x86-64 XenLinux cannot directly receive system calls from user processes. Despite such extra overheads, however, this framework allows to handle Xen hypercalls in the same fashion, and the hypercalls from the kernel are handled in the optimal fashion.

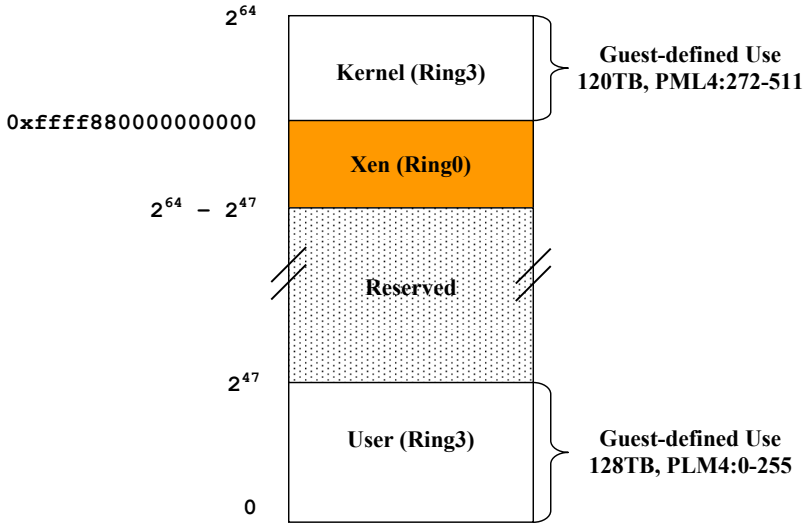

Figure 2: x86-64 Xen Address Space

- Xen must be aware in which mode the guest is running, kernel or user,
- SWAPGS is done by Xen so that the guest kernel can access PDA correctly without major modifications,
- the guest requsts Xen to switch to the user mode via a hypercall,
- The guests can modify the GS.base via a hypercall. The generic MSR handling is covered in Section 4.3.4 below.

4 Implementation

4.1 Memory Management

4.1.1 Memory Map

The memory given to a domain is a single contiguous region of pseudo-physical memory, and the total pages allocated is reported by "start info page" as described by Xen Interface above. This is handled as a very simple case of the E820 memory map, which is used by the native Linux.

4.1.2 2MB page and 1:1 Direct mapping

The native x86-64 Linux uses 2MB pages for the kernel and direct mapping for the physical memory, both of which are required for x86-64 XenLinux as well. At this point, Xen 3.0 does not support such super pages, and we needed to modify the initialization code to add one more paging level. Since this requires changes to the logic (e.g. extra page allocation for pte pages), a future version of Xen should support 2MB pages to minimize the changes to x86-64 Linux. In addition, the TLB efficiency can go down as we need to access more physical memory from the kernel.

4.2 Page Table Management

One of the notable changes required for Xen is to change the way the kernel allocate/deallocate the page table pages, including pgd, pmd, pud, and pte because those pages need to read-only.

The changes required are based on the well-established routines or the architecture-specific hooks, and thus they are cleanly replaced for x86-64 XenLinux.

4.2.1 `pgd_alloc` and `pgd_populate`

The `pgd_alloc` routine allocates to two back-to-back pgd pages for the kernel and user translations, and `pgd_populate` duplicates the modification to the kernel pgd into the user pgd.

Note that the `pgd_alloc`, for example, in the native x86-64 simply allocates a single page.

4.2.2 `pmd_alloc`, `pud_alloc`, `pte_alloc`

Since we pin the whole page tables at once when the pgd is pinned, we don't need to write-protect those pages at allocation time. However, we need to change the free routines. See Section 5.1.1 for "pinning".

4.2.3 `pmd_free`, `pud_free`, `pte_free`

When the page table pages we need to make sure that we remove write-protection from those pages. To that end, we use `update_va_mapping` to revert the page attribute (back to PAGE_KERNEL).

4.3 Process Management

4.3.1 Kernel and User Mode Transition in Xen

For each virtual CPU, Xen maintains a flag that indicates the guest kernel or user mode, and the x86-64 specific routine `toggle_guest_mode(structvcpu*v)` in Xen that toggles the kernel or user mode for the guest, switching the page tables accordingly.

4.3.2 Transition from the kernel to user mode by guest

When it needs to return to the user mode (after performing service for a system call, for example), x86-64 XenLinux needs to explicitly perform a hypercall `iret`, resulting in `toggle_guest_mode` from the kernel to the user mode in Xen.

The routine `toggle_guest_mode` is also called when switching from the user mode to the kernel, for example, upon exception or external interrupt in the user mode so that the kernel can handle the event notified by Xen.

4.3.3 Context Switching

The context code, especially, `switch_mm` (used for switching the address space) is simple and efficiently done by a multicall as shown in Figure 4.

The Figure 4 shows:

- Call `mm_pin()` if the next mm is not pinned. See Section 5.1.1 for this.
- Switch the kernel PML4 page,

```
static inline void pud_free(pud_t *pud)
{
  pte_t *ptep = virt_to_ptep(pud);

  if (!pte_write(*ptep)) {
      BUG_ON(HYPERVISOR_update_va_mapping(
          (unsigned long)pud,
          pfn_pte(virt_to_phys(pud) >> PAGE_SHIFT, PAGE_KERNEL),
          0));
  }
  free_page((unsigned long)pud);
}
```

Figure 3: pud_free

- Switch the user PML4 page,
- Switch LDT if needed, and
- Perform the three operations aboved by a single multicall hypercall.

4.3.4 MSR Handling

x86-64 Linux needs to access several MSRs at initialization and runtime as well.

- STAR, LSTAR, CSTAR, and SFMASK – These must be set to handle SYSCALL/SYSRET. As described in Section 3.1.2, they are initialized by Xen, not by guests on x86-64 XenLinux.
- EFER – This is read by Linux to check if NX is available at initialization time.
- GS.base, KernelGSbase, and FS.base – Access to GS.base is not frequent, but access to KernelGSbase and FS.base can be frequent.

To minimize changes to the original Linux, we emulate MSR access if rare. For example, access to EFER is emulated upon #GP in Xen, and the code in Figure 5 does not need any modification in x86-64 XenLinux.

However, FS.base is for the base address of TLS (Thread Local Storage), and thus can be modified frequently at context witch time. We use the `set_segment_base` hypercall if frequent.

4.4 DMA

Since the memory allocated to guests is "pseudo-physical" and can be anywhere in the system, e.g. >4GB or not physically contiguous, we need to convert guest physical to machine physical when specifying address for DMA. We reuse the swiotlb code in Linux, which was originally developed for IA-64. The code is shared by x86-32 and x86-64 XenLinux.

4.5 ACPI

The ACPI (Advanced Configuration and Power Interface) driver is a critical and complex component when configuring the I/O devices as well, and it is configured for x86-64 Linux distributions by default. The domain 0 has the

```
static inline void switch_mm(struct mm_struct *prev,
                             struct mm_struct *next,
                             struct task_struct *tsk)
{
  unsigned cpu = smp_processor_id();
  struct mmuext_op _op[3], *op = _op;

  if (likely(prev != next)) {
      if (!next->context.pinned)
      mm_pin(next);

      /* stop flush ipis for the previous mm */
      clear_bit(cpu, &prev->cpu_vm_mask);

      set_bit(cpu, &next->cpu_vm_mask);

      /* load_cr3(next->pgd) */
      op->cmd = MMUEXT_NEW_BASEPTR;
      op->arg1.mfn = pfn_to_mfn(__pa(next->pgd) >> PAGE_SHIFT);
      op++;

      /* xen_new_user_pt(__pa(__user_pgd(next->pgd))) */
      op->cmd = MMUEXT_NEW_USER_BASEPTR;
      op->arg1.mfn =
            pfn_to_mfn(__pa(__user_pgd(next->pgd)) >> PAGE_SHIFT);
      op++;

      if (unlikely(next->context.ldt != prev->context.ldt)) {
          /* load_LDT_nolock(&next->context, cpu) */
          op->cmd = MMUEXT_SET_LDT;
          op->arg1.linear_addr = (unsigned long)next->context.ldt;
          op->arg2.nr_ents     = next->context.size;
          op++;
      }

      BUG_ON(HYPERVISOR_mmuext_op(_op, op-_op, NULL, DOMID_SELF));
  }
```

Figure 4: switch_mm

```
arch/x86_64/kernel/setup64.c:

void __cpuinit check_efer(void)
{
    unsigned long efer;

    rdmsrl(MSR_EFER, efer);
    if (!(efer & EFER_NX) || do_not_nx) {
        __supported_pte_mask &= ~_PAGE_NX;
    }
}
```

Figure 5: check_efer in x86-64 XenLinux – unmodified

identical ACPI driver except a one-line change required to point to the RSDP because the physical address in the ACPI table needs to be comprehended as "machine physical" as opposed to "guest physical".

4.6 Local/IO APIC

The virtual CPU abstracted by Xen does not need to access the local APIC. IPI (Inter-Processor Interrupt), for example, is handled by local APIC on the native, but it is done simply by `event_channel_op` with `EVTCHNOP_send` on XenLinux.

The ACPI tables, such as MADT, is parsed by Xen, and the interrupt controllers, such as I/O APIC(s) is owned by Xen. However, the PCI interrupt routing information is provided by ACPI, and thus, the domain 0 needs to communicate the information to Xen. The following operations were added for the paravirtualization purpose, and they are handled by the hypercall `physdev_op`.

1. PHYSDEVOP_APIC_READ – Used to read an APIC register
2. PHYSDEVOP_APIC_WRITE – Used to write an APIC register
3. PHYSDEVOP_ASSIGN_VECTOR – This is used to for the domain 0 to communicate the interrupt routing information to Xen as mentioned above.

4.7 PCI

The PCI driver is also identical to the native except two lines, both of which are related to converting physical address to virtual address as the case with the ACPI driver.

4.8 Shadow Page Table for Live Migration

We extended the shadow code to support x86-64 XenLinux. A shadow page table is the effective page table fully controlled by Xen, whereas the guest page table is not active in terms of address translations but is managed and updated by the guest as the page table were effective. The page frame numbers in the guest page tables specify in "guest physical", thus they can continue to be same even if the underlying mapping from "guest physical" to "machine physical" is changed. This attribute is required for live migration, thus the shadow page support is required for XenLinux as well as HVM (Hardware-based Virtual Machine) guests.

The challenge with supporting XenLinux was to switch from the writable page table to shadow log-dirty mode at runtime. Since there are still some PTEs with write-protected, the shadow page needs to comprehend such special conditions. The "Log-dirty mode" is used to identify the pages modified in the guest memory to minimize the amount of the pages to transfer for live migration.

5 Optimizations

Performance of x86-64 XenLinux has been improved by various optimizations so far, leveraging the same techniques used for x86-32 XenLinux. In this section, we describe the most effective ones for x86-64 XenLinux.

5.1 Optimizations Techniques Used Today

In this section we discuss the performance optimizations techniques used today.

5.1.1 Pinning and Unpinning Page Tables

The most effective ones was "late pin, early unpin" because of the deeper levels of page tables for x86-64. Xen needs to check the page tables provided by guests to insure secure isolation, and Xen performs such checking **once** upon a request "pinning" from the guest. The page tables populated later are not pinned, and are modified by `update_va_mapping`.

`mm_pin`

At context switch time, especially in `switch_mm`, the new routine `mm_pin(structmm_struct*next)` is called if the page table for `next` is not pinned. A new field `pinned` was added to indicate the status.

The `mm_pin(next)` performs the following:

1. Change the page attribute to read-only by walking through the page table for `next`.
2. Change the page attribute of the kernel pgd
3. Change the page attribute of the user pgd
4. Set `next->context.pinned`

`arch_exit_mmap` and `mm_unpin`

XenLinux uses the standard hook `arch_exit_mmap` in `exit_map()` to unpin the defunct page table aggressively. The `mm_unpin(mm)` basically performs the reverse operation of `mm_pin()` above.

5.1.2 Writable Page Table

We extended the writable page table support for x86-64 in Xen. The writable page table requires fewer changes to guests and it is no slower for the batched interface that was used by the old version of Xen. In addition, the batched interface has problems with SMP guests, as the updates may be expected to be individually atomic.

5.2 Experiment

We made some experiment to overcome the overheads caused by the transitions between user and kernel mode.

Minimizing TLB Flush using a single PML4 page

As we discussed, today we flush TLB every time the guest switches between the kernel and user mode. The following steps basically reduces TLB flush at `toggle_guest_mode`.

1. When switching from the user to the kennel, just add the kernel translations, and don't flush TLBs. Since the number of PML4 entries used for the kernel is typically very small (typically only 3 on Linux), the cost is low.

2. When switching from the kernel to the user, remove the kernel translations from the PML4 page, and flush TLB.

Our experiment showed overall improvements with lmbench relative to the current Xen 3.0 (x86-64 XenLinux is based on 2.6.16). However, that did not improve other benchmarks, such as kernel build. Since this method still flushes the TLBs for the user process and more TLBs are used for the user mode in general, it may not make visible performance differences. We continue to investigate how we can improve performance in this area.

6 Conclusion

In this paper, we have presented a brief overview of Xen interface, the issues/areas required to be resolved when para-virtualizing x86-64 Linux, and the areas modified in that process, and the techniques used for performance optimizations.

Acknowledgment

A lot of developers in the Xen community contributed to x86-64 XenLinux in various areas, including stability, performance, bug fixes, SMP support, cleanups, and upgrades. We would like to thank especially the following people for their contributions: Christian Limpach, Chris Wright, Jan Beulich, and Li Xin.

References

[1] Virtual machine interface (vmi) specifications. `http://www.vmware.com/interfaces/vmi_specs.html`.

[2] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheirmer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Preedings of the Linux Symposium*, July 2005.

[3] University of Cambridge. *Interface Manual Xen v3.0 x86*. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`.

Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin