Paravirtualization

Twinkle Choudhary

11/09/2022
Contents

- Introduction of Paravirtualization
- CPU Virtualization
- Memory Virtualization
- I/O Virtualization
- Summary Discussion
Introduction
Recap - Virtualization Principles

Popek and Goldberg’s virtualization principles in 1974:

- Fidelity
- Performance
- Safety
Recap - Privileged and Sensitive Instructions

Goldberg (1974) two classes of instructions

- privileged instructions: those that trap when in user mode
- sensitive instructions: those that modify or depends on hardware configs
Recap - x86 difficulties (cpu)

- Not all sensitive instructions are privileged in x86.
- Hardware managed TLB gives Hypervisor no chance to intercept on TLB misses
- X86 has non-tagged TLB
  - Why this might be an obstacle for virtualization?
Recap - VMware Full Virtualization Solution

“VMware’s ESX Server dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention might be required”

- CPU: binary translation
Reasons against Full Virtualization

“These problems can be solved, but only at the cost of increased complexity and reduced performance.”

- Complexity
  - binary translation is very complicated
- Reduced performance
  - Management of shadow page table and pmap is expensive
  - Dynamic binary translation is expensive
ParaVirtualization

“We avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware.”

- Tradeoff between improved performance with slight modifications to the guest operating system.
- Expose the existence of hypervisor to guest OS, rather than fool them.
ParaVirtualization Design Principles

1. Virtualize all architectural features required by existing standard ABIs. *Why is this important?*
   - ABI: application binary interface
2. Supporting full multi-application modern operating systems is important.
3. Paravirtualization is necessary to work on uncooperative machine architectures such as x86.
Xen

- First public version released in 2003
- Backing major cloud service e.g. AWS for many years
- Natively supported by Linux after kernel version 3.0 in 2011
  - KVM ported in 2006
- Not widely used now
Xen chooses optimization while VMware chooses transparency
Structure of XEN

- **Control Plane Software**
- **User Software**
- **User Software**
- **User Software**

- **GuestOS (XenoLinux)** with Xeno-Aware Device Drivers
- **GuestOS (XenoLinux)** with Xeno-Aware Device Drivers
- **GuestOS (XenoBSD)** with Xeno-Aware Device Drivers
- **GuestOS (XenoXP)** with Xeno-Aware Device Drivers

- **Domain0 control interface**
- virtual x86 CPU
- virtual phy mem
- virtual network
- virtual blockdev

- **H/W (SMP x86, phy mem, enet, SCSI/IDE)**
Control Transfer: Hypercalls and Events

- Hypercall allows guest OS to perform a software trap to hypervisor for privileged operations
  - Similar to regular system call from user process to kernel

- Async events replaces device interrupts to perform notification from Xen to a domain
  - E.g. new data has been received from network
## Cost of Porting an OS to Xen

<table>
<thead>
<tr>
<th>OS subsection</th>
<th># lines</th>
<th>Linux</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture-independent</td>
<td>78</td>
<td>1299</td>
<td></td>
</tr>
<tr>
<td>Virtual network driver</td>
<td>484</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Virtual block-device driver</td>
<td>1070</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Xen-specific (non-driver)</td>
<td>1363</td>
<td>3321</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2995</td>
<td>4620</td>
<td></td>
</tr>
<tr>
<td>(Portion of total x86 code base)</td>
<td>1.36%</td>
<td>0.04%</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2:** The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).
Paravirtualized x86 Interface

<table>
<thead>
<tr>
<th>Memory Management</th>
<th>Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.</td>
</tr>
<tr>
<td>Paging</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Guest OS must run at a lower privilege level than Xen.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.</td>
</tr>
<tr>
<td>System Calls</td>
<td>Guest OS may install a ‘fast’ handler for system calls, allowing direct calls from an application into its guest OS and avoiding indireciting through Xen on every call.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Hardware interrupts are replaced with a lightweight event system.</td>
</tr>
<tr>
<td>Time</td>
<td>Each guest OS has a timer interface and is aware of both ‘real’ and ‘virtual’ time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device I/O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Network, Disk, etc.</td>
<td>Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.</td>
</tr>
</tbody>
</table>

Details will be discussed in following sections
CPU Virtualization
## CPU Virtualization

| Memory Management |  |
|-------------------|  |
| Segmentation      | Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space. |
| Paging            | Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages. |

| CPU               |  |
|-------------------|  |
| Protection        | Guest OS must run at a lower privilege level than Xen. |
| Exceptions        | Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same. |
| System Calls      | Guest OS may install a ‘fast’ handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call. |
| Interrupts        | Hardware interrupts are replaced with a lightweight event system. |
| Time              | Each guest OS has a timer interface and is aware of both ‘real’ and ‘virtual’ time. |

| Device I/O        |  |
|-------------------|  |
| Network, Disk, etc. | Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications. |
Protection (Privileged Instruction)

- Any guest OS attempt to directly execute a privileged instruction is failed by processor.
- All sensitive instructions are paravirtualized to be hypercall to Xen.

Graph credit to https://www.geeksforgeeks.org/virtualization-xen-paravirtualization/
Protection (Privileged Instruction)

Compare with VMware full virtualization on handling of non privileged sensitive instructions in x86

<table>
<thead>
<tr>
<th>Method</th>
<th>Xen (ParaVirtualization)</th>
<th>VMware ESX (Full Virtualization)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Modification on guest OS to issue Hypercall for sensitive instructions</td>
<td>Dynamically rewrites portions of the hosted machine code to insert traps (binary translation)</td>
</tr>
<tr>
<td>Pros</td>
<td>Lightweight and efficient</td>
<td>No need to port the guest OS</td>
</tr>
<tr>
<td>Cons</td>
<td>Need to port OS to support Xen</td>
<td>Slow</td>
</tr>
</tbody>
</table>

Guest OS must run at a lower privilege level than Xen.
### Exceptions and System Calls

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Calls</td>
<td>Guest OS may install a ‘fast’ handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.</td>
</tr>
</tbody>
</table>

- Each guest OS registers an exception handler table with Xen for validation (no execution in ring 0)
- Some exceptions (e.g. system call) can be handled by guest OS directly through a installed fast handler
- Page fault must be handled by Xen for ring 0 privilege
Exceptions and System Calls

The diagram illustrates the flow of system calls and exceptions in both native and paravirtualized systems. It highlights the different rings (Ring0 to Ring3) and the interaction between the application, kernel, and hypervisor.

1. Native System Call: The system call process in a native environment.
2. Paravirtualized System Call: The system call process in a paravirtualized environment, where the kernel is replaced by a hypervisor.
4. Hypercall: A method of invoking a system call that can bypass the kernel.
Interrupts

- Hardware interrupts notify domains in the form of events
- Pending events stored in a per-domain bitmask
- Each guest OS specifies an event-callback handler which
  - Resets pending events set
  - Responds to events properly
- Event handling can be deferred by domain by set a flag
  - Similar to disabling interrupts
- What's the benefit of using asynchronous event?
When could knowing real time be important?

When could knowing virtual time be important?
Memory Virtualization
Recap: Software-based Memory Virtualization

- Hypervisor will maintain a pmap to record the mapping from PPN to MPN
- When Guest OS tries to change page table, it will be trapped to hypervisor, and hypervisor will dynamically generate a Shadow Page Table
Recap: Hardware-assisted Memory Virtualization

- EPT is used to translate from PPN to MPN
- Hardware will first translate from VPN to PPN using guest page table, then use EPT to translate to MPN
Direct paging

- Guest OSes are responsible for the translation from PPN to MPN
  - Guest OS directly manages page tables
  - Page tables are mappings of VPN to MPN
- The top 64MB of each address space is reserved for Xen
  - This avoids the overhead of TLB flushes when entering or leaving hypervisors
Update page tables

- Page tables are read-only for Guest OSes
  - This is to prevent Guest OSes from making unacceptable changes
- Updates should be passed to Xen through a hypercall
  - Xen will validate this request before applied
    - No mapping to other guests’ pages
    - No write mapping to page table pages
  - Updates can be batched to reduce the number of hypercalls
    - This is particularly beneficial when creating new address spaces
- Guest OSes also can change the machine-to-physical table maintained by Xen through a hypercall
Page Faults

- Hypervisor will first check if this page fault happens in its own area
  - If so, hypervisor will handle it by itself
  - If not, hypervisor will transfer to page fault handler registered by Guest OS
- Can we use an unmodified page fault handler?
Page Faults

- Hypervisor will first check if this page fault happens in its own area
  - If so, hypervisor will handle it by itself
  - If not, hypervisor will transfer to page fault handler registered by Guest OS

- Can we use an unmodified page fault handler?
  - CR2 register can only be read in ring 0!

- Xen will create a copy of the exception stack frame on the guest OS stack, including the value of CR2 register
Full process of a page fault

1. Application (Guest OS) requests a page fault.
2. Page Fault signal is sent to the Hypervisor Page Fault Handler.
3. The Hypervisor Page Fault Handler handles the request.
4. A Hypercall is made by the Hypervisor to access physical memory.
5. The Hypercall is handled by the Hypercall Handler.
6. The Hypercall Handler returns the page to the application (Guest OS).
7. The application (Guest OS) resumes execution with the updated page.
Discussion: SPT vs EPT vs Direct Paging
I/O Virtualization
Xen handle I/O virtualization using I/O ring and asynchronized event delivery mechanism

- Xen does not emulate hardware devices
- Exposes device abstractions for simplicity and performance.
- I/O data transferred to/from guest via Xen using shared-memory buffers
- Virtualized interrupts: lightweight event delivery mechanism from Xen-guest to xen
I/O ring mechanism introduction, with 4 descriptor pointers

- Ring is circular queue of descriptors
- Descriptors are allocated by guest OS
- Descriptors don’t directly contain I/O data
- Two pairs of producer/consumer pointers
- Domains place request
- Domain Advances request producer pointer
- Xen removes and handles them
- Xen advances request consumer pointer
- Zero copy transfer
Step 1, Domains place request
Domain Advances request producer pointer

Request Consumer
Private pointer in Xen

Request Producer
Shared pointer updated by guest OS

Response Producer
Shared pointer updated by Xen

Response Consumer
Private pointer in guest OS

- **Request queue** - Descriptors queued by the VM but not yet accepted by Xen
- **Outstanding descriptors** - Descriptor slots awaiting a response from Xen
- **Response queue** - Descriptors returned by Xen in response to serviced requests
- **Unused descriptors**
Step 2, Xen removes and handles them, asynchronously.
Step 3, Xen advances request consumer pointer.

- **Request Consumer**
  - Private pointer in Xen

- **Request Producer**
  - Shared pointer updated by guest OS

- **Response Producer**
  - Shared pointer updated by Xen

- **Response Consumer**
  - Private pointer in guest OS

- **Request queue** - Descriptors queued by the VM but not yet accepted by Xen
- **Outstanding descriptors** - Descriptor slots awaiting a response from Xen
- **Response queue** - Descriptors returned by Xen in response to serviced requests
- **Unused descriptors**
Step 4, Zero Copy to the domains

- **Request Consumer**: Private pointer in Xen
- **Request Producer**: Shared pointer updated by guest OS
- **Response Producer**: Shared pointer updated by Xen
- **Response Consumer**: Private pointer in guest OS

- **Request queue**: Descriptors queued by the VM but not yet accepted by Xen
- **Outstanding descriptors**: Descriptor slots awaiting a response from Xen
- **Response queue**: Descriptors returned by Xen in response to serviced requests
- **Unused descriptors**
Discussion: Pros and Cons of IO ring mechanism ?
Advantages

Security
  More secure than grant direct access to the hardware

Performance improvement
  Since we use asynchronous event delivery mechanism and zero copy, the performance is significantly better than other IO virtualization technique.
Discussion: Pros and Cons of IO ring mechanism

Advantages

Security
More secure than grant direct access to the hardware

Performance improvement
Since we use asynchronous event delivery mechanism and zero copy, the performance is significantly better than other IO virtualization technique.

Disadvantages

Violate the fidelity rule.

No guarantee delivery
“If no frame is available, the packet is dropped.” If the descriptor buffer is filled up, there could be a chance we may lose the packets.
Performance

- Close to native setting (L-UP/L-SMP)
- Much better than VMW and UML

<table>
<thead>
<tr>
<th>Config</th>
<th>null call</th>
<th>l/O</th>
<th>stat</th>
<th>openslct</th>
<th>closeTCP</th>
<th>sig inst</th>
<th>sig hndl</th>
<th>fork proc</th>
<th>exec sh proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-SMP</td>
<td>0.53</td>
<td>0.81</td>
<td>2.10</td>
<td>3.51</td>
<td>23.2</td>
<td>0.83</td>
<td>2.94</td>
<td>143</td>
<td>601</td>
</tr>
<tr>
<td>L-UP</td>
<td>0.45</td>
<td>0.50</td>
<td>1.28</td>
<td>1.92</td>
<td>5.70</td>
<td>0.68</td>
<td>2.49</td>
<td>110</td>
<td>530</td>
</tr>
<tr>
<td>Xen</td>
<td>0.46</td>
<td>0.50</td>
<td>1.22</td>
<td>1.88</td>
<td>5.69</td>
<td>0.69</td>
<td>1.75</td>
<td></td>
<td>198</td>
</tr>
<tr>
<td>VMW</td>
<td>0.73</td>
<td>0.83</td>
<td>1.88</td>
<td>2.99</td>
<td>11.1</td>
<td>1.02</td>
<td>4.63</td>
<td>874</td>
<td>2k3</td>
</tr>
<tr>
<td>UML</td>
<td>24.7</td>
<td>25.1</td>
<td>36.1</td>
<td>62.8</td>
<td>39.9</td>
<td>26.0</td>
<td>46.0</td>
<td>21k</td>
<td>33k</td>
</tr>
</tbody>
</table>

Table 3: lmbench: Processes - times in μs

<table>
<thead>
<tr>
<th>Config</th>
<th>2p 0K</th>
<th>2p 16K</th>
<th>2p 64K</th>
<th>8p 16K</th>
<th>8p 64K</th>
<th>16p 16K</th>
<th>16p 64K</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-SMP</td>
<td>1.69</td>
<td>1.88</td>
<td>2.03</td>
<td>2.36</td>
<td>26.8</td>
<td>4.79</td>
<td>38.4</td>
</tr>
<tr>
<td>L-UP</td>
<td>0.77</td>
<td>0.91</td>
<td>1.06</td>
<td>1.03</td>
<td>24.3</td>
<td>3.61</td>
<td>37.6</td>
</tr>
<tr>
<td>Xen</td>
<td>1.97</td>
<td>2.22</td>
<td>2.67</td>
<td>3.07</td>
<td>28.7</td>
<td>7.08</td>
<td>39.4</td>
</tr>
<tr>
<td>VMW</td>
<td>18.1</td>
<td>17.6</td>
<td>21.3</td>
<td>22.4</td>
<td>51.6</td>
<td>41.7</td>
<td>72.2</td>
</tr>
<tr>
<td>UML</td>
<td>15.5</td>
<td>14.6</td>
<td>14.4</td>
<td>16.3</td>
<td>36.8</td>
<td>23.6</td>
<td>52.0</td>
</tr>
</tbody>
</table>

Table 4: lmbench: Context switching times in μs
Performance Benchmark

The file create and file delete time does not show significant improvement over other methods.

two transitions into Xen per page

<table>
<thead>
<tr>
<th>Config</th>
<th>0K File create</th>
<th>0K File delete</th>
<th>10K File create</th>
<th>10K File delete</th>
<th>Mmap lat</th>
<th>Prot fault</th>
<th>Page fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-SMP</td>
<td>44.9</td>
<td>24.2</td>
<td>123</td>
<td>45.2</td>
<td>99.0</td>
<td>1.33</td>
<td>1.88</td>
</tr>
<tr>
<td>L-UP</td>
<td>32.1</td>
<td>6.08</td>
<td>66.0</td>
<td>12.5</td>
<td>68.0</td>
<td>1.06</td>
<td>1.42</td>
</tr>
<tr>
<td>Xen</td>
<td>32.5</td>
<td>5.86</td>
<td>68.2</td>
<td>13.6</td>
<td>139</td>
<td>1.40</td>
<td>2.73</td>
</tr>
<tr>
<td>VMW</td>
<td>35.3</td>
<td>9.3</td>
<td>85.6</td>
<td>21.4</td>
<td>620</td>
<td>7.53</td>
<td>12.4</td>
</tr>
<tr>
<td>UML</td>
<td>130</td>
<td>65.7</td>
<td>250</td>
<td>113</td>
<td>1k4</td>
<td>21.8</td>
<td>26.3</td>
</tr>
</tbody>
</table>

Table 5: **imbench**: File & VM system latencies in $\mu s$
Performance Benchmark

Network benchmark shows significant improvements.

Smaller MTU -> More packets -> more interrupts -> more performance penalty.

<table>
<thead>
<tr>
<th></th>
<th>TCP MTU 1500</th>
<th></th>
<th>TCP MTU 500</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TX</td>
<td>RX</td>
<td>TX</td>
<td>RX</td>
</tr>
<tr>
<td>Linux</td>
<td>897</td>
<td>897</td>
<td>602</td>
<td>544</td>
</tr>
<tr>
<td>Xen</td>
<td>897 (-0%)</td>
<td>897 (-0%)</td>
<td>516 (-14%)</td>
<td>467 (-14%)</td>
</tr>
<tr>
<td>VMW</td>
<td>291 (-68%)</td>
<td>615 (-31%)</td>
<td>101 (-83%)</td>
<td>137 (-75%)</td>
</tr>
<tr>
<td>UML</td>
<td>165 (-82%)</td>
<td>203 (-77%)</td>
<td>61.1 (-90%)</td>
<td>91.4 (-83%)</td>
</tr>
</tbody>
</table>

Table 6: ttcp: Bandwidth in Mb/s
Summary Discussion
What goals of Xen are not valid or less valid in today's cloud environments? Or, why Xen is not that popular now?

Use case of ParaVirtualization now?

https://dl.acm.org/authorize?N47257