A Comparison of Software and Hardware Techniques for x86 Virtualization

Introduction:

- What is the motivation for writing the paper?

Section 2 (Classical Virtualization):

- What is classical virtualization, according to the paper? (And P&G)
  Why is x86 not classically virtualizable?

- What are shadow structures and why they are necessary?
  How does the type of shadow state affect the difficulty of maintaining it?
  (e.g., CPU state vs. in-memory state)

- What differentiates "true" vs. "hidden" page faults?
  Can you think of an analogous phenomenon in Xen's MMU virtualization? If so, compare & contrast.

- What mappings are stored in the shadow page tables? Why is this?
  How are shadow page tables kept coherent with guest page tables?

- What is the "interpretive execution" model described in Section 2.5? How can this be helpful?

Section 3 (Software Virtualization)

- What are the motivations for the six properties of VMWare's BT described in section 3.2? Discuss the relative importance and desirability of each.

- The first example in section 3.2 has no privileged operations, so why
  does it need any translation at all?

- How is x86's segmentation mechanism useful for VMWare's software
  VMM?

- What are the four major types of non-IDENT (not-identical) translatable instructions, and why are they that way?

- What does the "adaptive binary translation" adapt to? How is this helpful?

Section 4 and 5 (HW virtualization)

- What virtualization support mechanism(s) does the HW provide? How does this help?

- What is the VMCB? How does it assist in virtualization, and what sort of fields it might contain?

- Give some examples of design decisions in the three components mentioned in Section 4.4 that could affect VM exit rate.

- What are comparative advantages and disadvantages of BT and HW VMM?

Section 6 (Evaluation)

- Why is the HW VMM superior at handling in-guest user/kernel transitions, as illustrated in the 2D graphics test (Section 6.1)?

- Why does HW do so poorly on forkwait compared to the SW VMM?

- Section 6.3 mentions that syscalls are costly in the SW VMM. What was Xen's approach to reducing syscall overheads? How does that compare with the HW VMM's approach?

- Why is it that some instruction like 'in' can be even faster on the SW VMM than native execution? Why is the HW VMM is so slow in handling the same 'in' instructions?

- Consider the seven "nanobenchmarks" in sections 6.3. Which of the benchmarks do you think are likely to be important in terms of impact on real world performance? Which are unimportant?

- pgfault nanobenchmark: HW VMM spends much more time in faults, but why? Both are using shadow page tables, and trap overhead doesn't seem too heavy (see divzero benchmark), so what may be taking up all the extra time?

Section 7 (Future improvements)

- Why does the HW VMM incur some extra overheads relative to the SW VMM even if the cost of VMM exits was zero?

- How would HW MMU virtualization support help? What overheads can it avoid or reduce?

Introduction:

- What is the motivation for writing the paper?

Section 2 (Classical Virtualization):

- What is classical virtualization, according to the paper? (And P&G)
  Why is x86 not classically virtualizable?

- What are shadow structures and why they are necessary?
  How does the type of shadow state affect the difficulty of maintaining it?
  (e.g., CPU state vs. in-memory state)

- What differentiates "true" vs. "hidden" page faults?
  Can you think of an analogous phenomenon in Xen's MMU virtualization? If so, compare & contrast.

- What mappings are stored in the shadow page tables? Why is this?
  How are shadow page tables kept coherent with guest page tables?

- What is the "interpretive execution" model described in Section 2.5? How can this be helpful?

Section 3 (Software Virtualization)

- What are the motivations for the six properties of VMWare's BT described in section 3.2? Discuss the relative importance and desirability of each.

- The first example in section 3.2 has no privileged operations, so why
  does it need any translation at all?

- How is x86's segmentation mechanism useful for VMWare's software
  VMM?

- What are the four major types of non-IDENT (not-identical) translatable instructions, and why are they that way?

- What does the "adaptive binary translation" adapt to? How is this helpful?

Section 4 and 5 (HW virtualization)

- What virtualization support mechanism(s) does the HW provide? How does this help?

- What is the VMCB? How does it assist in virtualization, and what sort of fields it might contain?

- Give some examples of design decisions in the three components mentioned in Section 4.4 that could affect VM exit rate.

- What are comparative advantages and disadvantages of BT and HW VMM?

Section 6 (Evaluation)

- Why is the HW VMM superior at handling in-guest user/kernel transitions, as illustrated in the 2D graphics test (Section 6.1)?

- Why does HW do so poorly on forkwait compared to the SW VMM?

- Section 6.3 mentions that syscalls are costly in the SW VMM. What was Xen's approach to reducing syscall overheads? How does that compare with the HW VMM's approach?

- Why is it that some instruction like 'in' can be even faster on the SW VMM than native execution? Why is the HW VMM is so slow in handling the same 'in' instructions?

- Consider the seven "nanobenchmarks" in sections 6.3. Which of the benchmarks do you think are likely to be important in terms of impact on real world performance? Which are unimportant?

- pgfault nanobenchmark: HW VMM spends much more time in faults, but why? Both are using shadow page tables, and trap overhead doesn't seem too heavy (see divzero benchmark), so what may be taking up all the extra time?

Section 7 (Future improvements)

- Why does the HW VMM incur some extra overheads relative to the SW VMM even if the cost of VMM exits was zero?

- How would HW MMU virtualization support help? What overheads can it avoid or reduce?