Lecture 09 - A comparison of software and hardware techniques for x86 virtualization

Keith Adams and Ole Agesen, VMware · ASPLOS 2006. VMware’s binary-translating VMM against its first VT-x VMM: how each one works, why the hardware usually lost, and what hardware would have to add to win.

Contents

01 Classical virtualization

02 Ways to virtualize

03 The question

04 Binary translation

05 Hardware assist

06 Measurements

07 Looking ahead

08 Takeaways

01

Classical virtualization

Popek and Goldberg: what makes a VMM

Three requirements from 1974. Every approach in this deck is judged against them.

01

Fidelity

Software runs exactly as it would on the hardware, except for timing.

02

Performance

The overwhelming majority of guest instructions run directly on the CPU, without the VMM.

03

Safety

The VMM stays in control of all hardware resources.

CLASSICALLY VIRTUALIZABLE

An architecture where trap-and-emulate alone meets all three. x86 is not; it is still virtualizable by these criteria, using binary translation.

Two x86 behaviors that break trap-and-emulate

A de-privileged kernel must trap on everything privileged. On x86, some privileged behavior neither traps nor stays hidden.

PRIVILEGED STATE IS VISIBLE

```
mov  %cs, %ax      ; selector
and  $3, %ax       ; low bits = CPL
```

The current privilege level sits in the low two bits of %cs. A de-privileged kernel that reads it sees it is not at CPL 0, and nothing traps.

PRIVILEGED INSTRUCTIONS THAT DON’T TRAP

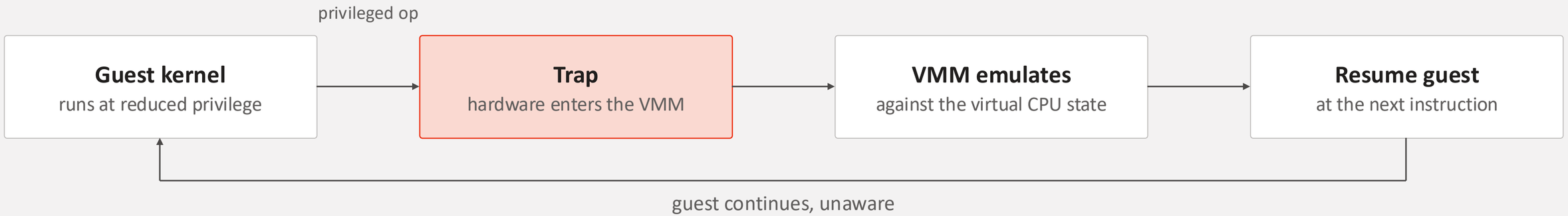
```
popf               ; CPL 0: may set IF
                   ; CPL 3: IF ignored
```

In the kernel, popf can change the interrupt flag. De-privileged, it drops that change silently, so the VMM never learns the guest wanted interrupts off.

- One such instruction is enough to rule out trap-and-emulate for binary-only guests like Windows.

Trap-and-emulate: run the guest kernel de-privileged

The guest kernel runs directly, but at reduced privilege. Anything privileged traps to the VMM, which emulates it on the virtual CPU.



TRAP BY INSTRUCTION TYPE

Some instructions trap whenever they run unprivileged, for example out to an I/O port.

TRAP BY PROTECTION

The VMM protects what the guest touches, for example the address range of a memory-mapped device.

02

Ways to virtualize

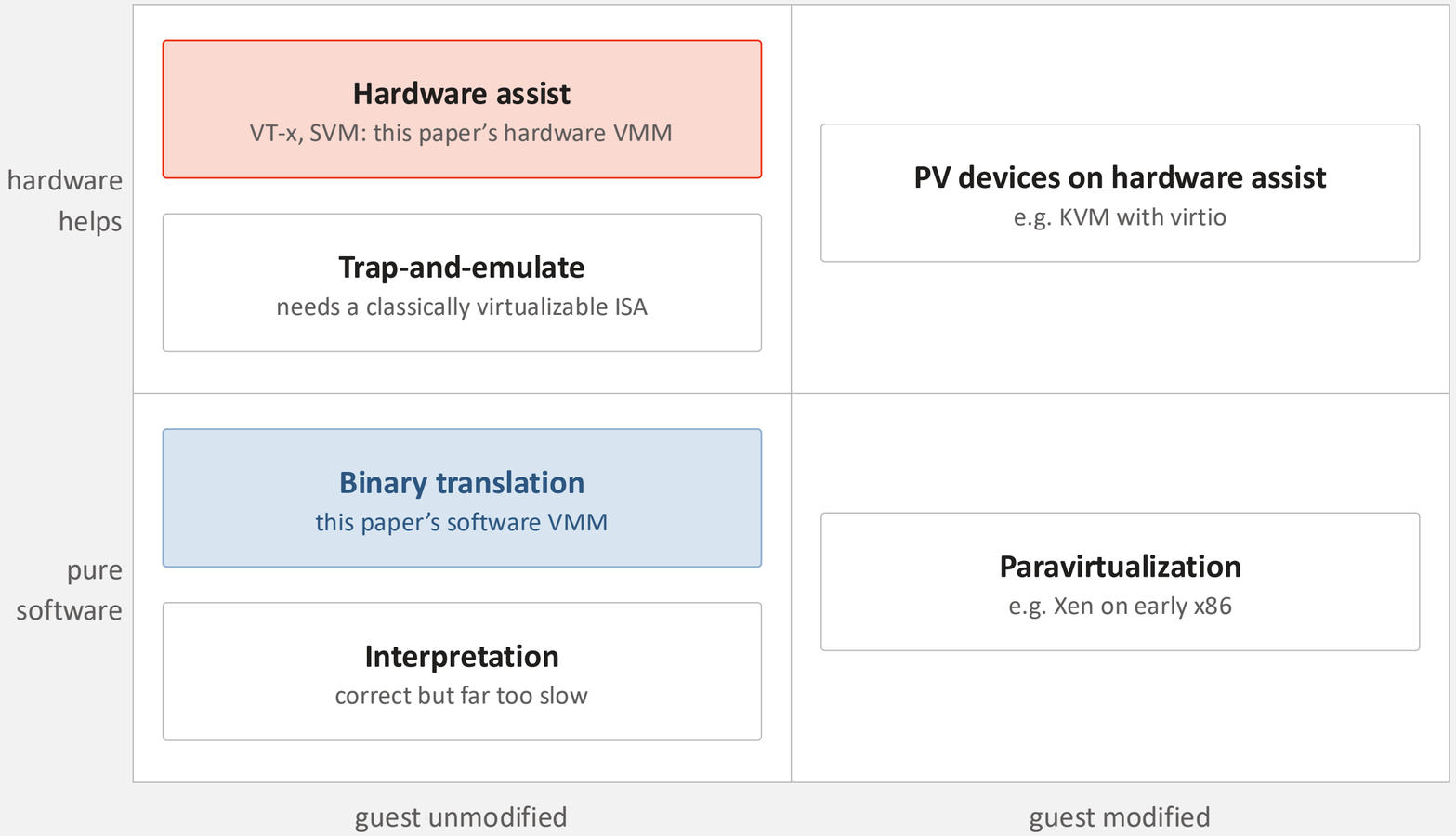
Classical VMMs were refined in two directions

In the first VM boom one company often built the hardware, the VMM and the guest OS, so it could change either interface. Both ideas have heirs today.

WHAT CHANGES	FIRST VM BOOM	PRESENT BOOM, 2006
The OS/VMM interface guest is modified	**Guest OS passes higher-level information to the VMM**	**Paravirtualization** Xen, Denali
The hardware/VMM interface CPU is extended	**IBM S/370 interpretive execution (SIE)**	**Hardware assist** Intel VT-x, AMD SVM
Neither guest code is rewritten	**Not needed: the hardware was virtualizable**	**Binary translation** VMware, Virtual PC

- Paper §2.5. Paravirtualization relaxes fidelity for speed; interpretive execution needs new hardware.

Two questions place every approach



THIS PAPER

Compares the two shaded boxes: unmodified guests, with and without hardware help.

MOVING RIGHT

A faster, simpler VMM, but closed-source guests like Windows can't be changed.

MIXING IS NORMAL

The top-right box is background, not the paper: CPU via hardware, devices paravirtualized.

How each approach catches privileged operations

Every VMM must notice when the guest touches privileged state. They differ in how, and in what that costs.

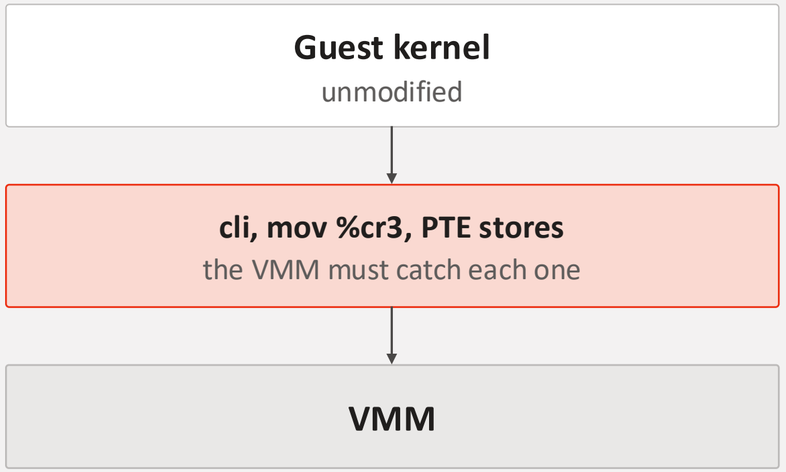
APPROACH	CATCHES PRIVILEGED OPS BY	GUEST CHANGES	MAIN COST
Interpretation	decoding every instruction in software	none	hundreds of host instructions per guest instruction
Trap-and-emulate	hardware traps from a de-privileged guest	none	a trap per privileged op; impossible on x86
Binary translation	rewriting kernel code before it runs	none	translation, indirect branches, MMU tracing
Paravirtualization	explicit hypercalls in a modified kernel	kernel source	no unmodified guests; a new interface to maintain
Hardware assist	exits from the CPU’s guest mode	none	costly exits; no MMU help in the first generation

- The paper measures the two shaded rows against each other. Both run unmodified guests.

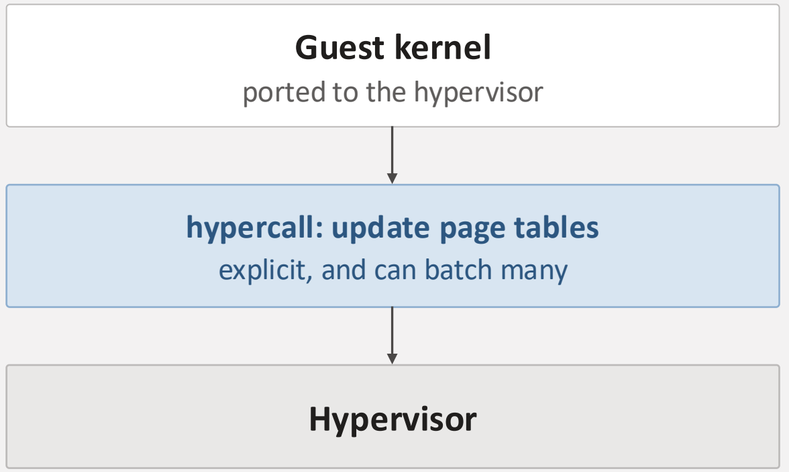
Paravirtualization: the guest stops pretending

Instead of executing privileged instructions and being caught, a modified guest kernel asks the hypervisor directly.

FULL VIRTUALIZATION



PARAVIRTUALIZATION



WHAT IT GAINS

Fewer, cheaper transitions and a simpler VMM.

WHAT IT GIVES UP

Unmodified and proprietary guests, plus a new interface to maintain.

YOU'VE SEEN IT

virtio (lecture 07) is paravirtualization for devices only.

03

The question

Two ways to run an unmodified x86 kernel

By 2006 VMware had shipped a software VMM for years. Intel VT-x and AMD SVM now offered a hardware route. The authors built both into VMware Player 1.0.1 and measured them on the same machine.

SOFTWARE VMM · BINARY TRANSLATION

Guest kernel code
unmodified x86 binary

↓

Binary translator
rewrites it on demand, block by block

↓

Translation cache
runs natively; privileged ops become cheap callouts

Privileged operations never reach the hardware as traps: the translator replaces them before they run.

HARDWARE VMM · VT-X / SVM

Guest kernel code
unmodified x86 binary

↓

CPU in guest mode
runs it directly, privileged code included

↓

Exit to the VMM
on the events the VMM chose to intercept

No translation at all. Cost is concentrated in exits: each one is a full switch out of guest mode and back.

The headline: the hardware VMM was usually slower

forkwait creates and destroys 40,000 processes. Time on a 3.8 GHz Pentium 4:

6.0 s
Native
the baseline

36.9 s
Software VMM
6.1× native

106.4 s
Hardware VMM
17.7× native

WHY, ACCORDING TO THE AUTHORS

1. The hardware offers no support for MMU virtualization, so page tables must still be shadowed in software.
2. It does not combine with the software techniques (adaptive translation) that made shadowing affordable.

04

Binary translation

Binary translation: an interpreter, made fast

Interpreting x86 meets fidelity and safety but burns hundreds of instructions per guest instruction. VMware's translator has six properties:

Binary

Input is x86 machine code, not source.

Dynamic

Translation is interleaved with running the result.

On demand

Code is translated just before it first runs, so code and data never need telling apart.

System level

No ABI assumptions: even a smashed return address behaves as on hardware.

Subsetting

Input is all of x86, privileged instructions included; output is a safe subset.

Adaptive

Translations are rewritten when the guest's behavior changes.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq                 ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

- being translated now
- already translated
- new in the cache
- patched: chained
- never translated
- continuation fires / never ran

TRANSLATION CACHE

empty

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

START

The guest calls isPrime(49)

Nothing is translated yet. The translator starts at the guest PC and reads raw code bytes from guest memory: binary in, not source.

STATE

%edi = a = 49

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq                 ; sign-extend
          idiv %esi           ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            jmp  [fallthrAddr]
```

STEP 1 OF 6

Translate unit 1

The unit stops at jge, a control-flow instruction. Three instructions copy IDENT. jge becomes two continuations, one per successor; each calls the translator the first time it runs.

- being translated now
- already translated
- new in the cache
- patched: chained
- never translated
- continuation fires / never ran

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STATE

i = 2: is 2 ≥ 49? No, so [fallthrAddr] fires.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq               ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            jmp  [fallthrAddr]           ; elided
nexti':    *mov  %eax, %ecx              ; IDENT
            cdq
            idiv %esi
            test %edx, %edx
            jz   [takenAddr2]            ; JCC
            jmp  [fallthrAddr2]
```

- being translated now
- already translated
- new in the cache
- patched: chained
- never translated
- continuation fires / never ran

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STEP 2 OF 6

Fall-through: translate unit 2

The new fragment is emitted right after fragment 1, where execution goes anyway. The jump to it is no longer needed, so fragment 1 now falls through into fragment 2.

STATE

49 % 2 = 1, so jz is not taken and [fallthrAddr2] fires.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq               ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            ; fall-thru into next CCF
nexti':    *mov  %eax, %ecx              ; IDENT
            cdq
            idiv %esi
            test %edx, %edx
            jz   [takenAddr2]            ; JCC
            jmp  [fallthrAddr2]          ; elided
           *inc  %esi                    ; IDENT
            cmp  %esi, %ecx
            jl   [takenAddr3]            ; JCC
            jmp  [fallthrAddr3]
```

being translated now
already translated
new in the cache
patched: chained
never translated
continuation fires / never ran

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STEP 3 OF 6

Fall through again: unit 3

Fragment 3 goes right after fragment 2, so fragment 2 falls into it too. It ends in jl nexti, and this time the branch is taken.

STATE

i = 3: is 3 < 49? Yes, so [takenAddr3] fires.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq               ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            ; fall-thru into next CCF
nexti':    *mov  %eax, %ecx              ; IDENT
            cdq
            idiv %esi
            test %edx, %edx
            jz   [takenAddr2]            ; JCC
            ; fall-thru into next CCF
           *inc  %esi                    ; IDENT
            cmp  %esi, %ecx
            jl   nexti'                  ; JCC
            jmp  [fallthrAddr3]
```

being translated now
already translated
new in the cache
patched: chained
never translated
continuation fires / never ran

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STEP 4 OF 6

Chaining: jl goes straight to nexti'

nexti is already translated. The translator finds nexti' in its guest-to-cache hash table and patches the continuation into a direct jump. The loop now runs entirely in the cache.

STATE

i = 3 … 6: no translator calls at all.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq               ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

being translated now
already translated
new in the cache
patched: chained
never translated
continuation fires / never ran

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            ; fall-thru into next CCF
nexti':    *mov  %eax, %ecx              ; IDENT
            cdq
            idiv %esi
            test %edx, %edx
            jz   notPrime'               ; JCC
            ; fall-thru into next CCF
           *inc  %esi                    ; IDENT
            cmp  %esi, %ecx
            jl   nexti'                  ; JCC
            jmp  [fallthrAddr3]
notPrime': *xor  %eax, %eax              ; IDENT
            pop  %r11                    ; RET
            mov  %gs:0xff39eb8(%rip), %rcx ; spill
            movzx %ecx, %r11b
            jmp  %gs:0xfc7dde0(8*%rcx)
```

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STEP 5 OF 6

49 % 7 = 0: translate notPrime

jz is taken for the first time. Its continuation translates notPrime, then is chained to the new fragment. ret is indirect, so it becomes the %gs table lookup.

STATE

i = 7: 49 % 7 = 0, so jz is taken.

Translating isPrime(49), step by step

GUEST CODE

```
isPrime:  mov  %ecx, %edi   ; %ecx = a
          mov  %esi, $2     ; i = 2
          cmp  %esi, %ecx   ; i >= a?
          jge  prime
nexti:    mov  %eax, %ecx   ; %eax = a
          cdq               ; sign-extend
          idiv %esi         ; a % i
          test %edx, %edx   ; remainder 0?
          jz   notPrime
          inc  %esi         ; i++
          cmp  %esi, %ecx   ; i >= a?
          jl   nexti
prime:    mov  %eax, $1     ; return 1
          ret
notPrime: xor  %eax, %eax   ; return 0
          ret
```

- being translated now
- already translated
- new in the cache
- patched: chained
- never translated
- continuation fires / never ran

TRANSLATION CACHE

```
isPrime':  *mov  %ecx, %edi              ; IDENT
            mov  %esi, $2
            cmp  %esi, %ecx
            jge  [takenAddr]             ; JCC
            ; fall-thru into next CCF
nexti':    *mov  %eax, %ecx              ; IDENT
            cdq
            idiv %esi
            test %edx, %edx
            jz   notPrime'               ; JCC
            ; fall-thru into next CCF
           *inc  %esi                    ; IDENT
            cmp  %esi, %ecx
            jl   nexti'                  ; JCC
            jmp  [fallthrAddr3]
notPrime': *xor  %eax, %eax              ; IDENT
            pop  %r11                    ; RET
            mov  %gs:0xff39eb8(%rip), %rcx ; spill
            movzx %ecx, %r11b
            jmp  %gs:0xfc7dde0(8*%rcx)
```

Operands are destination first, as in the paper. * marks the first instruction of each fragment.

STEP 6 OF 6

The cache after isPrime(49)

Four fragments; * marks each one's first instruction. Two jumps were elided, two branches chained, and two continuations never ran. The "return 1" block was never translated.

STATE

Next call: runs from the cache, mostly without the translator.

Translating ret: why the code reaches for %gs

A guest return address can’t be jumped to directly, so ret becomes a table lookup. Its working space lives in VMM memory, which only %gs can reach.

NOTPRIME’ · TRANSLATION OF RETURN 0

```
xor   %eax, %eax                ; IDENT
pop   %r11                      ; guest return addr
mov   %gs:0xff39eb8(%rip), %rcx ; spill
movzx %ecx, %r11b               ; 8-bit hash
jmp   %gs:0xfc7dde0(8*%rcx)     ; via table
```

Destination first, as in the paper. Red: accesses to VMM memory.

THE HASH TABLE IS VMM DATA TOO

movzx keeps the low byte of the return address; jmp reads its target from a 256-entry table through %gs.

WHY RET ISN’T IDENT

The stacked address points into guest code. Jumping there would leave the translation cache.

IT NEEDS A SCRATCH REGISTER

Guest registers live in the real ones, so the guest’s %rcx is saved first and reloaded later.

ONLY %GS REACHES A SAFE PLACE

Guest memory is visible to the guest; guest segments stop below the VMM (more on this shortly). %gs is reserved for VMM data.

- This lookup is why call/ret costs 51 cycles under the software VMM, against 11 natively.

Why pop the return address first?

The address could stay on the stack. The paper doesn’t discuss this choice; the reasons below are our inference.

ALTERNATIVE · LEAVE IT ON THE STACK

```
mov   %gs:spill(%rip), %rcx  ; spill
movzx %ecx, byte (%rsp)      ; hash it
lea   %rsp, 8(%rsp)          ; drop it
jmp   %gs:table(8*%rcx)
```

WHAT THE TRANSLATOR DOES

```
pop   %r11                   ; read + move %rsp
mov   %gs:0xff39eb8(%rip), %rcx
movzx %ecx, %r11b
jmp   %gs:0xfc7dde0(8*%rcx)
```

RET MOVES THE STACK

ret adds 8 to %rsp; jmp doesn’t. pop reads and adjusts in one step and, unlike add, leaves the flags alone.

HITS NEED CHECKING

Only 8 bits index the table, so the target must compare the full address. It’s in %r11, not a dead stack slot.

FAULT FIRST

If pop faults, nothing has changed yet: it looks like the guest’s own ret faulting. Later, the spill would need undoing.

OPEN QUESTION

pop overwrites %r11 without saving it. The paper doesn’t say why that is safe.

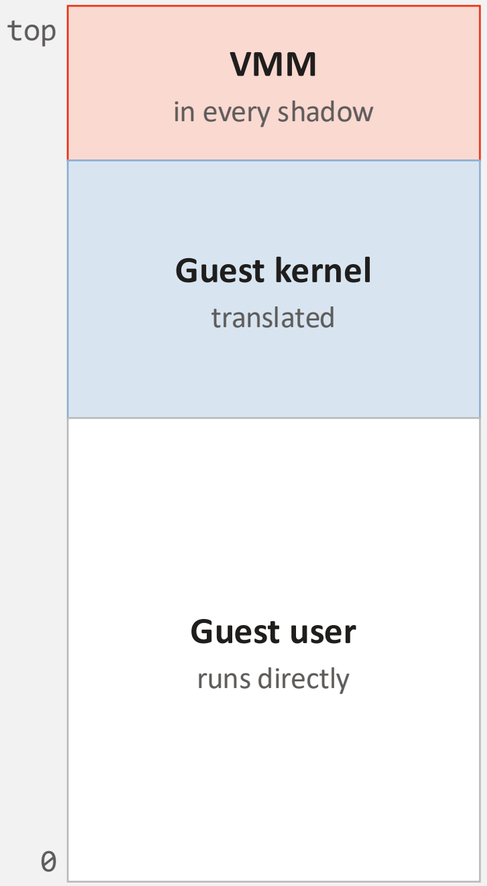
- Keeping the address on the stack would leave %r11 untouched: a deliberate trade-off, or a detail the listing omits.

What cannot be translated IDENT, and what it costs

CASE	WHY NOT IDENT	COST
PC-relative addressing	The translation lives at a different address	Compensation code: small growth and slowdown
Direct control flow	Code layout changes in translation	Mapped at translation time: insignificant
Indirect control flow	jmp, call and ret targets are known only at run time	A lookup, e.g. a hash table: usually single-digit %
Privileged instructions	They must act on the virtual CPU, not the real one	Simple ones run inline, often faster than native; complex ones call out

- Only guest kernel code needs translation. User code runs directly at native speed, switching modes with the guest.

One address space, two tenants: guest and VMM



WHO RUNS WHERE

CPL 0
the VMM

CPL 1
translated guest kernel

CPL 3
guest user code, direct

CPL assignment as described in VMware's later account (Bugnion et al., 2012).

THE VMM RIDES ALONG

It maps itself into the top of every shadow page table, so entering it is a jump, not a switch.

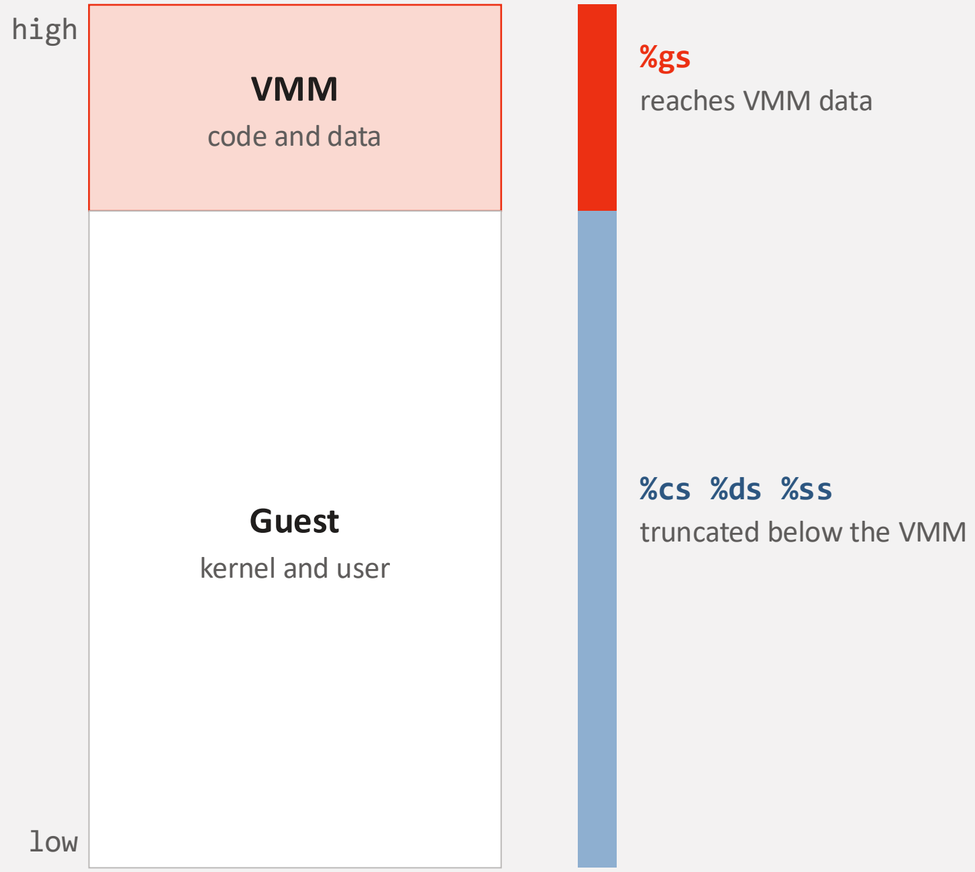
USER CODE: STOPPED BY PAGING

VMM pages are supervisor-only. A CPL 3 access to them faults.

KERNEL CODE: STOPPED BY SEGMENTS

Paging has one user/supervisor bit, and CPL 1 counts as supervisor. Only segment limits fence it off.

Keeping translated code out of the VMM



SEGMENTATION, NOT PAGING

Segment limits are checked by hardware on every access, so translated loads and stores stay IDENT.

%GS IS THE ESCAPE HATCH

The translator adds %gs prefixes when it needs VMM memory, e.g. to spill a register.

THE CATCH

Intel’s 64-bit mode has no segment limits, so on the paper’s Intel machine the software VMM is not safe.

64-bit mode: the fence is gone, the door remains

32-BIT PROTECTED MODE

SEGMENTS	BASE	LIMIT
`%cs %ds %es %ss`	used	checked
`%fs %gs`	used	checked

- Limits work, so truncated segments fence off the VMM.

64-BIT LONG MODE

SEGMENTS	BASE	LIMIT
`%cs %ds %es %ss`	forced to 0	ignored
`%fs %gs`	used	ignored

- No limits at all. Only %fs and %gs keep a base.

THE FENCE IS GONE

Truncated segments can’t stop a store reaching the VMM. On Intel, the paper’s BT VMM fails Safety.

THE DOOR REMAINS

%gs still has a base, so %gs-relative access to VMM data, as in ret, still works.

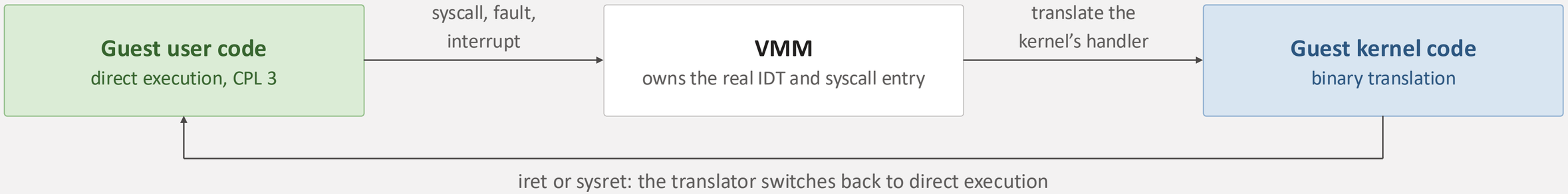
WHAT VMWARE DID

AMD added an optional long-mode limit check (EFER.LMSLE). On Intel, 64-bit guests used VT-x.

- From the paper (§7.3): BT on Intel would need a VMM protection scheme without segments. The third column is background, not from the paper.

User code runs directly; only the kernel is translated

The VMM switches engines at the guest's own kernel/user boundary. At CPL 3, x86 behaves as if it were classically virtualizable.



THE CPL IT SEES IS REAL

User code reads CPL 3 from %cs, and CPL 3 is where it really is.

POPF BEHAVES NATIVELY

At user level, popf ignores IF changes on real hardware too. Nothing is lost.

FORBIDDEN OPS STILL FAULT

cli, out or a %cr3 load faults as on hardware; the VMM hands it to the guest kernel.

- The cost moves to the boundary: a system call takes about 2,300 cycles under the software VMM, against 242 natively.

Why only “most user code on most guests”

The paper leaves the details out of scope. These are examples of what would break direct execution; they are ours, not the paper’s.

01

User I/O privilege

If a guest sets IOPL 3, e.g. for an X server, cli and port I/O run at user level without trapping. cli would disable real interrupts.

02

Leaky instructions

sgdt, sidt, sldt and smsw didn’t trap at CPL 3 on CPUs of the time. They show the VMM’s tables, not the guest’s.

03

Address-space conflicts

The VMM occupies the top of the address space. A guest whose processes use that range collides with it.

THE FALLBACK

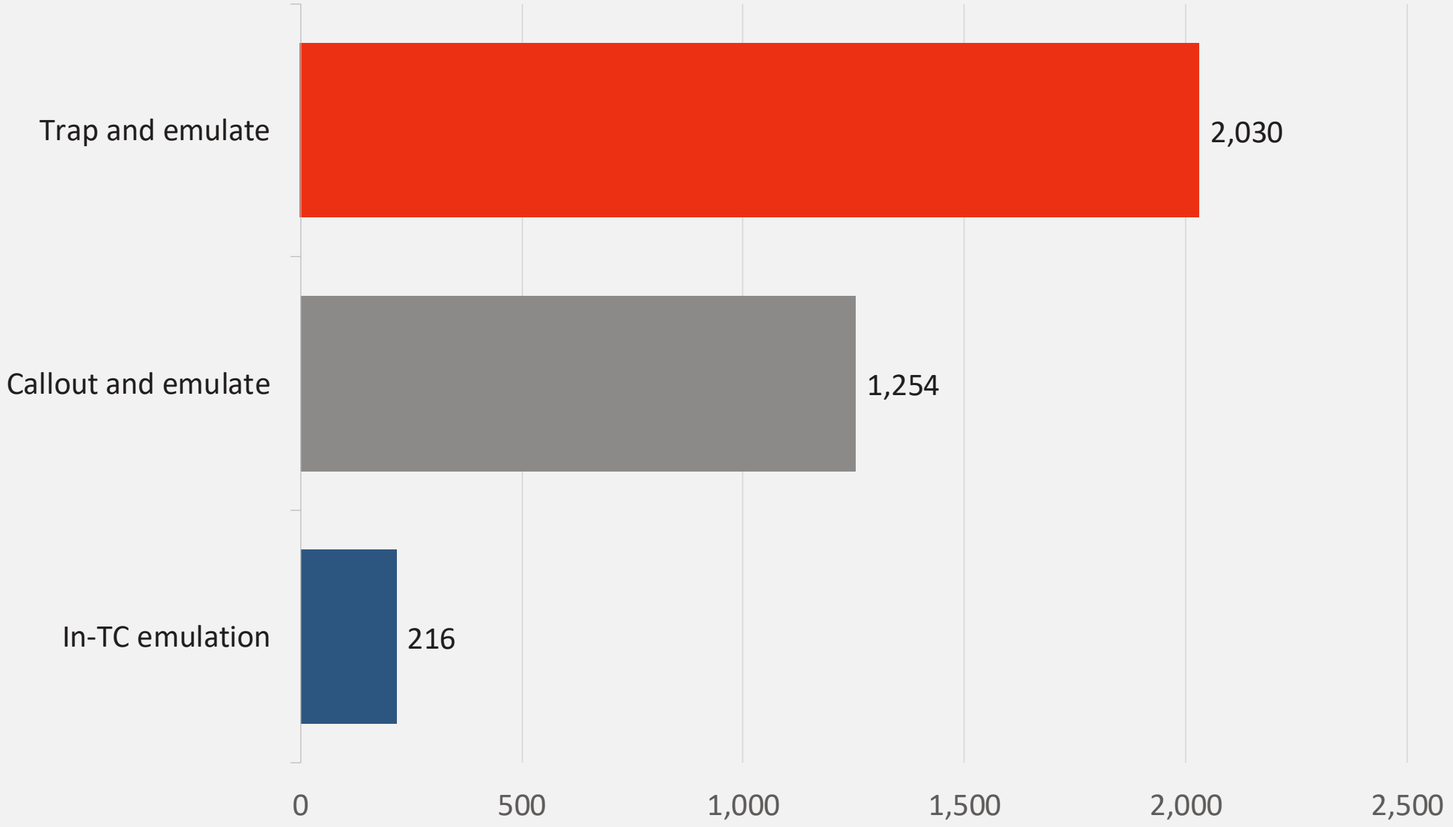
Translate the user code too. Direct execution is the fast path, not a requirement.

WHY IT MATTERS FOR THE RESULTS

SPECint and SPECjbb run near native because they live on this fast path.

Traps are expensive; translation avoids them

The same privileged instruction, rdtsc, emulated three ways on a Pentium 4:



TRAP AND EMULATE

The classical path: a hardware fault into the VMM.

CALLOUT

A call from translated code into the VMM. No fault.

IN-TC

A few instructions inline in the translation cache.

Three kinds of address, one MMU

The guest maps virtual to guest-physical; the VMM maps guest-physical to host-physical. Before nested paging, the MMU could walk only one table.

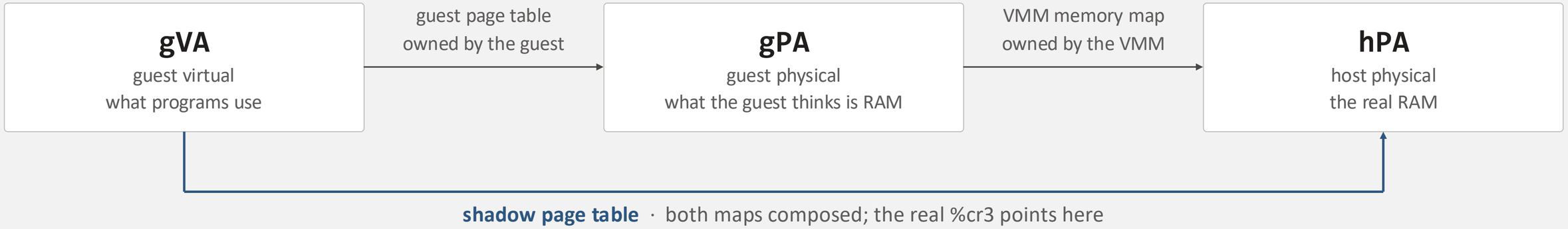
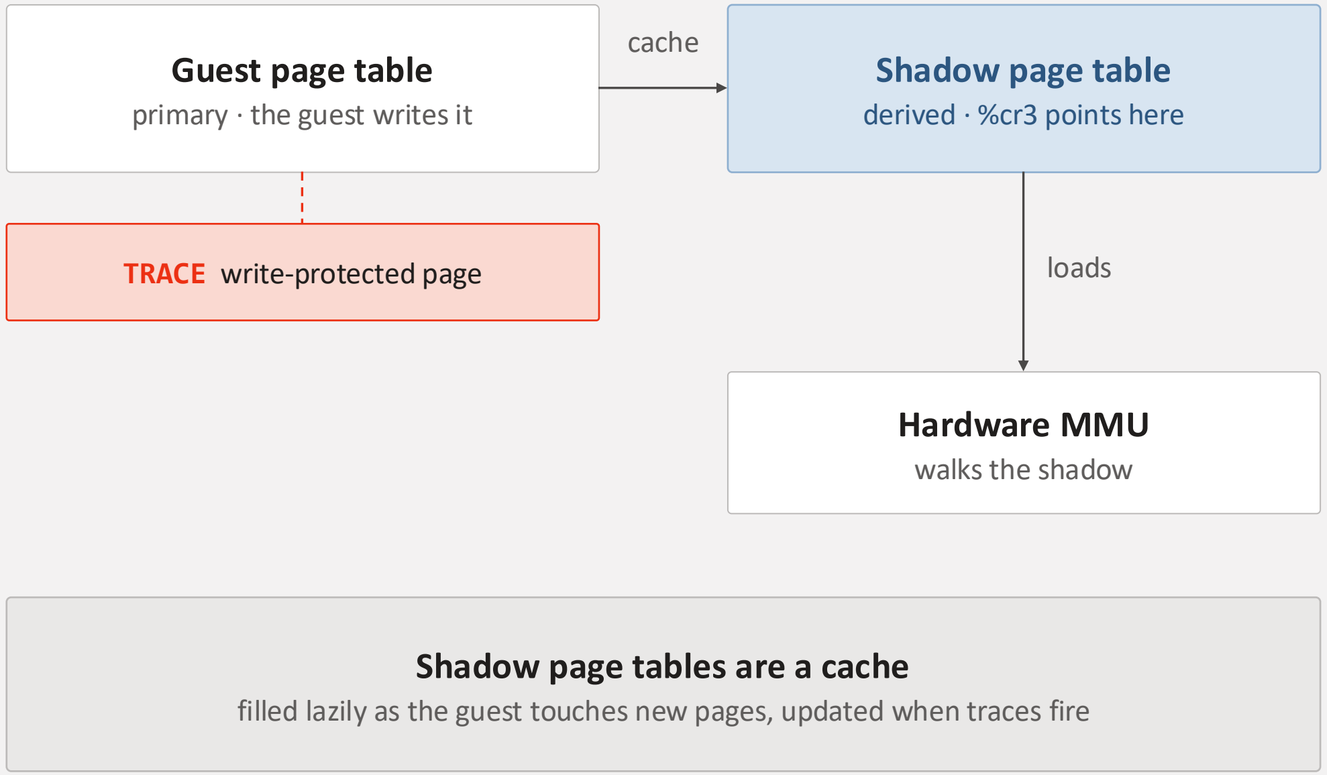


TABLE	MAPS	EXAMPLE
Guest page table	gVA → gPA	0x00400000 → 0x01234000
VMM memory map	gPA → hPA	0x01234000 → 0x09876000
Shadow page table	gVA → hPA	0x00400000 → 0x09876000

Shadow page tables, kept coherent with traces



Shadow page tables are a cache

filled lazily as the guest touches new pages, updated when traces fire

HIDDEN PAGE FAULT

A miss in the shadow only. The VMM fills the entry and resumes; the guest never sees it.

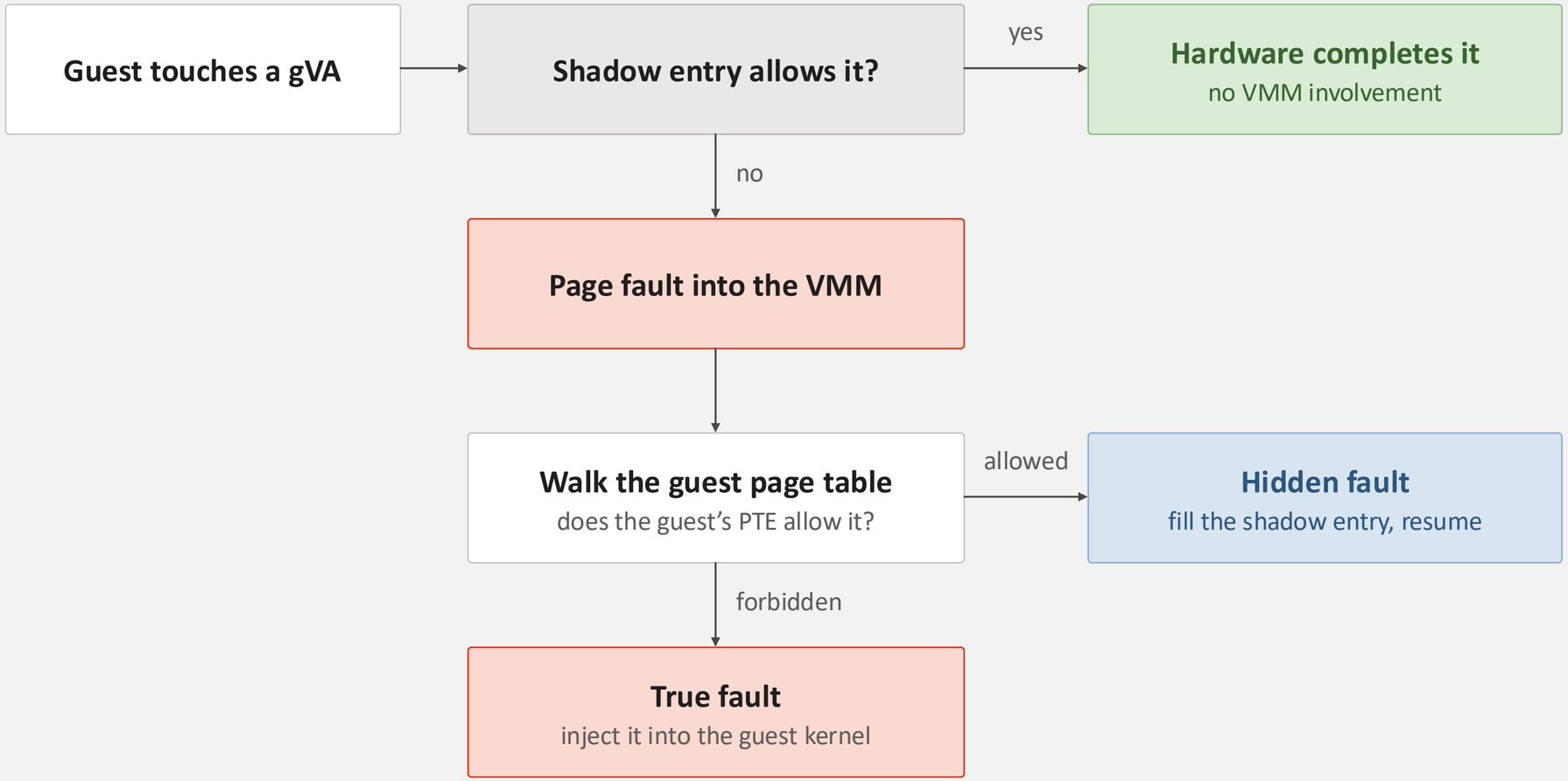
TRUE PAGE FAULT

The guest’s own entry forbids the access. The VMM forwards it to the guest.

TRACE FAULT

The guest wrote a shadowed page-table page. The VMM emulates the store and updates the shadow.

One guest access, followed through the shadow



%CR3 WRITE

Load the cached shadow for this guest root, or start an empty one.

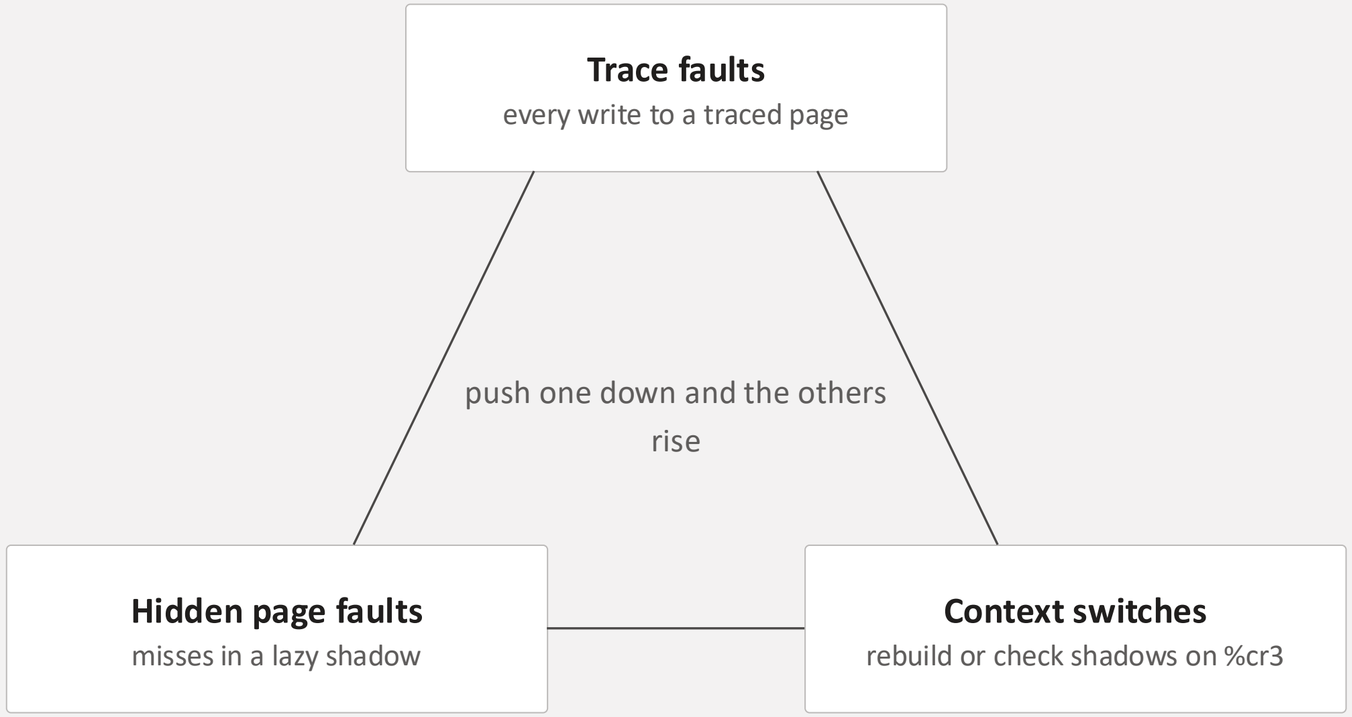
GUEST PTE WRITE

Traced: fault and update.
Untraced: wait for the TLB flush.

A/D BITS

Copy hardware-set bits back into the guest's PTEs.

Every shadow-paging design trades off three costs



TRACE A LOT

Shadows stay coherent and fill eagerly, but every guest PTE write faults.

TRACE LITTLE

PTE writes are free, but the VMM takes more hidden faults or revalidates on each switch.

THE PAPER'S POINT

Hardware assist makes all three dearer: each one becomes a VM exit.

Eager vs lazy shadows: when does the VMM find out?

	① Guest maps a page PTE: not present → present	② Guest touches it first access to the page	③ Guest unmaps it PTE write, then invlpg	④ Switch process guest loads a new %cr3
EAGER trace guest page tables	**Trace fault** VMM fills the shadow now	**Shadow hit** no fault	**Trace fault** VMM drops the entry	**Reload shadow** cached copy still coherent
LAZY no traces; act on flushes	**No fault** the shadow goes stale	**Hidden fault** VMM fills it on first touch	**Flush intercepted** VMM drops it at invlpg	**Revalidate** or rebuild the shadow

Red: the VMM gets involved. **Green**: the hardware handles it alone.

EAGER PAYS WHEN TRACES ARE CHEAP

With adaptive translation (next), a traced write is a callout of about 391 cycles. VMware traced liberally.

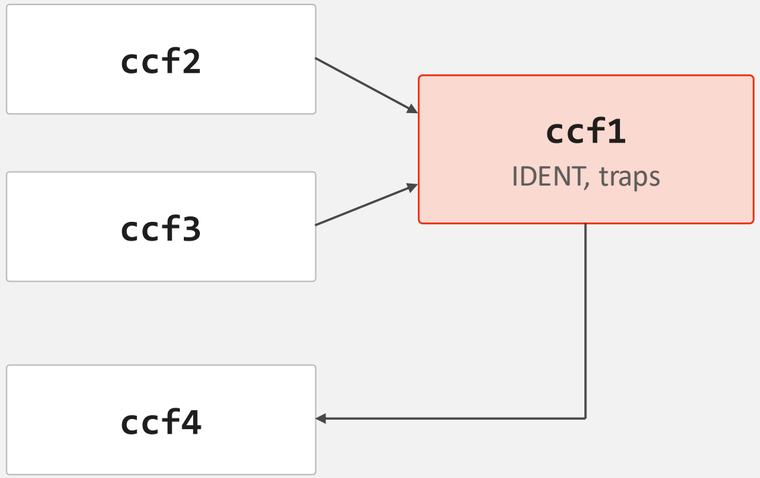
LAZY PAYS WHEN TRACES ARE DEAR

On the hardware VMM that write is a 12,700-cycle exit. VMware began exploring lighter tracing.

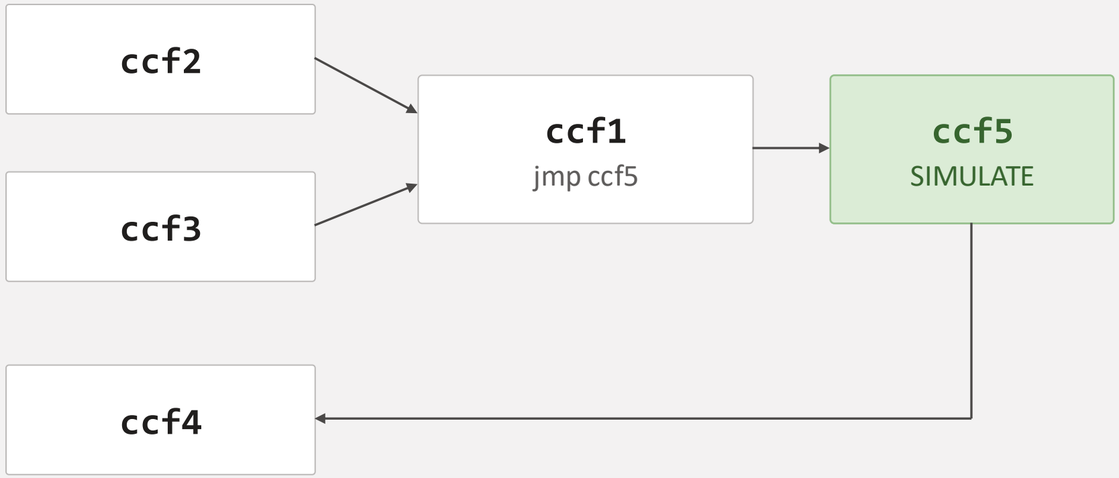
Adaptive translation: innocent until proven guilty

Plain loads and stores can still trap when they touch traced page tables. The translator watches for instructions that trap often and retranslates them.

BEFORE: CCF1 TRAPS ON EVERY RUN



AFTER: CCF1 FORWARDS TO CCF5

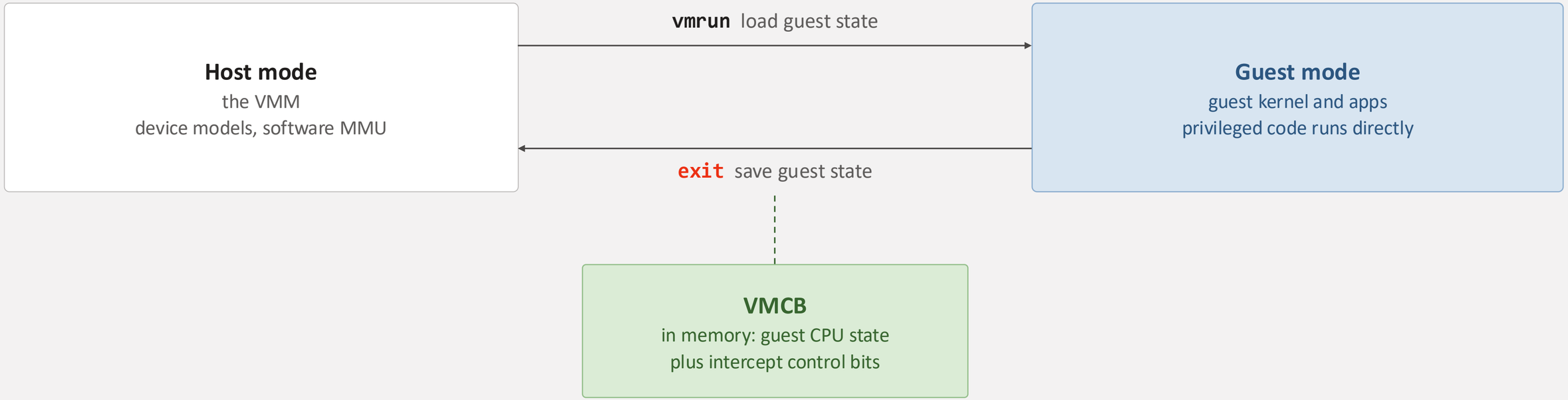


- Patching ccf1 is constant-time: no predecessor needs redirecting. It reverses if the instruction turns innocent again.

05

Hardware assist

VT-x and SVM: guest mode, the VMCB, vmrun and exit



VMRUN

Loads the guest's state from the VMCB and runs it in guest mode.

EXIT

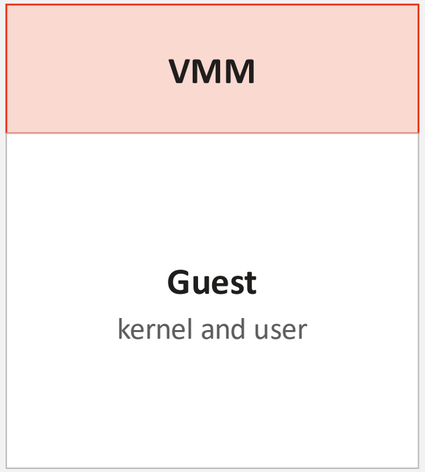
On a condition set in the VMCB. Fields say why, e.g. the I/O port.

SHADOWED STATE

Some state, like %eflags, lives in the VMCB, so popf needn't exit.

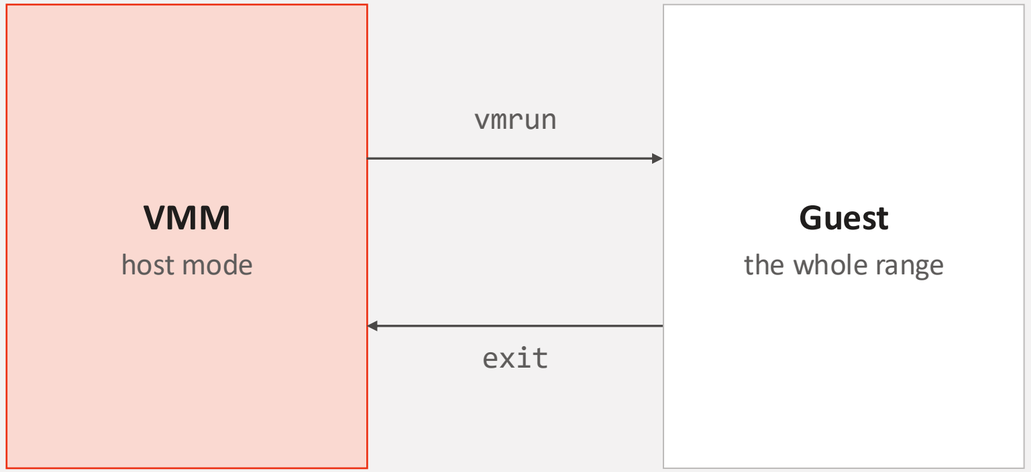
With hardware assist, the VMM leaves the guest’s space

SOFTWARE VMM · ONE SHARED SPACE



one %cr3: a shadow with the VMM mapped in

HARDWARE VMM · SEPARATE SPACES



host %cr3

guest %cr3, from the VMCB

NO FENCE NEEDED

Guest and VMM never share a space, so lost 64-bit segment limits don’t matter.

SHADOWS REMAIN

Without nested paging, the guest’s %cr3 still points at a VMM-built shadow.

ENTRY IS AN EXIT

Reaching the VMM is a full context swap, not a call.

What the hardware VMM still has to intercept

The hardware gives no help with the MMU, so the VMM inherits the software VMM’s shadow page tables, and every MMU event becomes an exit.

GUEST EVENT	EXITS?	WHY
System call	**No**	The guest’s trap vectors are loaded on the hardware
popf, %eflags changes	**No**	The VMCB keeps a shadow %eflags
%cr8 writes	**Rarely**	Only when lowered below a pending virtual interrupt
Page faults	**Yes**	Fill the shadow on hidden faults; vet true faults
%cr3 loads, TLB flushes	**Yes**	Switch or rebuild the shadow page table
Writes to traced page tables	**Yes**	Keep the shadow coherent
I/O instructions and MMIO	**Yes**	Devices are emulated in software

One fork() on the hardware VMM

A UNIX guest creates a process. Only the first step stays inside guest mode.

1	**fork() system call** CPL 3 to 0 happens on the hardware	No exit
2	**Write-protect for copy-on-write** each guest PTE write hits a trace	Exit per write
3	**Switch to the child** loading %cr3; VMM builds a new shadow table	Exit
4	**Child touches its pages** hidden page faults fill the shadow	Exit per page
5	**Writes to shared pages** true faults, vetted then forwarded to the guest	Exit, then inject

Where each approach wins, before measuring anything

Binary translation wins

Trap elimination
adaptive BT turns most traps into callouts

Emulation speed
a callout gets an already-decoded instruction

Callout avoidance
common cases are emulated inline in the cache

Hardware assist wins

Code density
no translation, so no code growth

Precise exceptions
no guest state to rebuild on a fault

System calls
run with no VMM involvement at all

IN SHORT

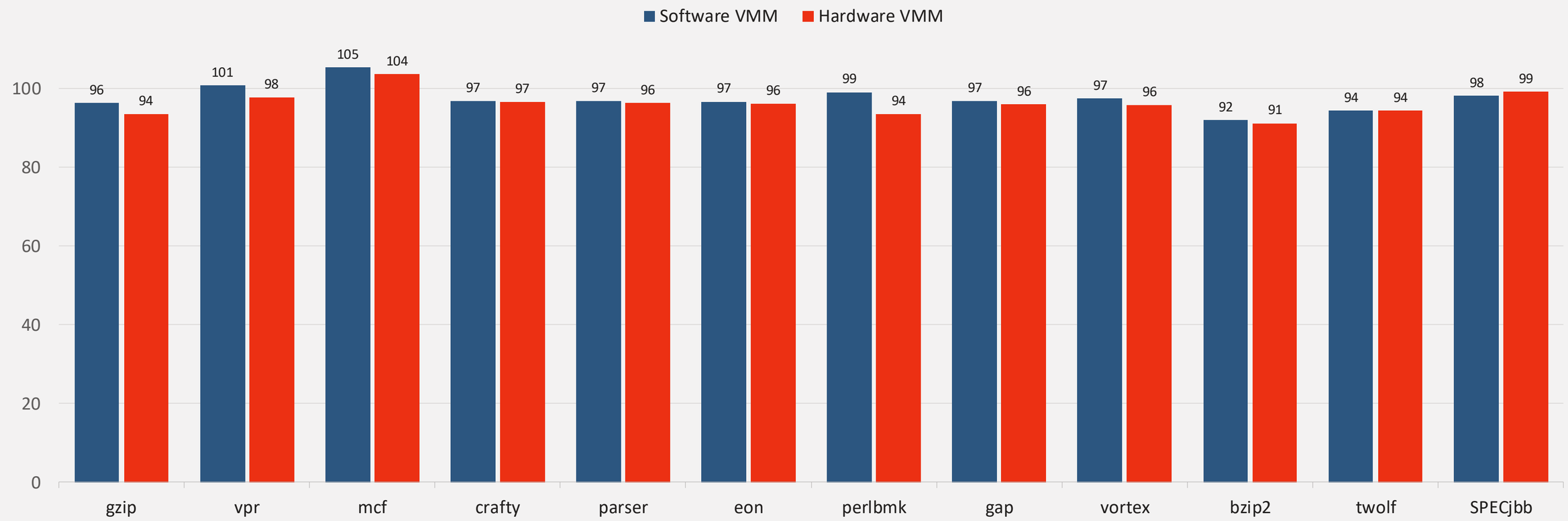
Hardware runs at native speed until it exits, and each exit is costly. Software pays on every kernel instruction, but makes most traps cheap.

06

Measurements

Compute-bound code runs near native on both

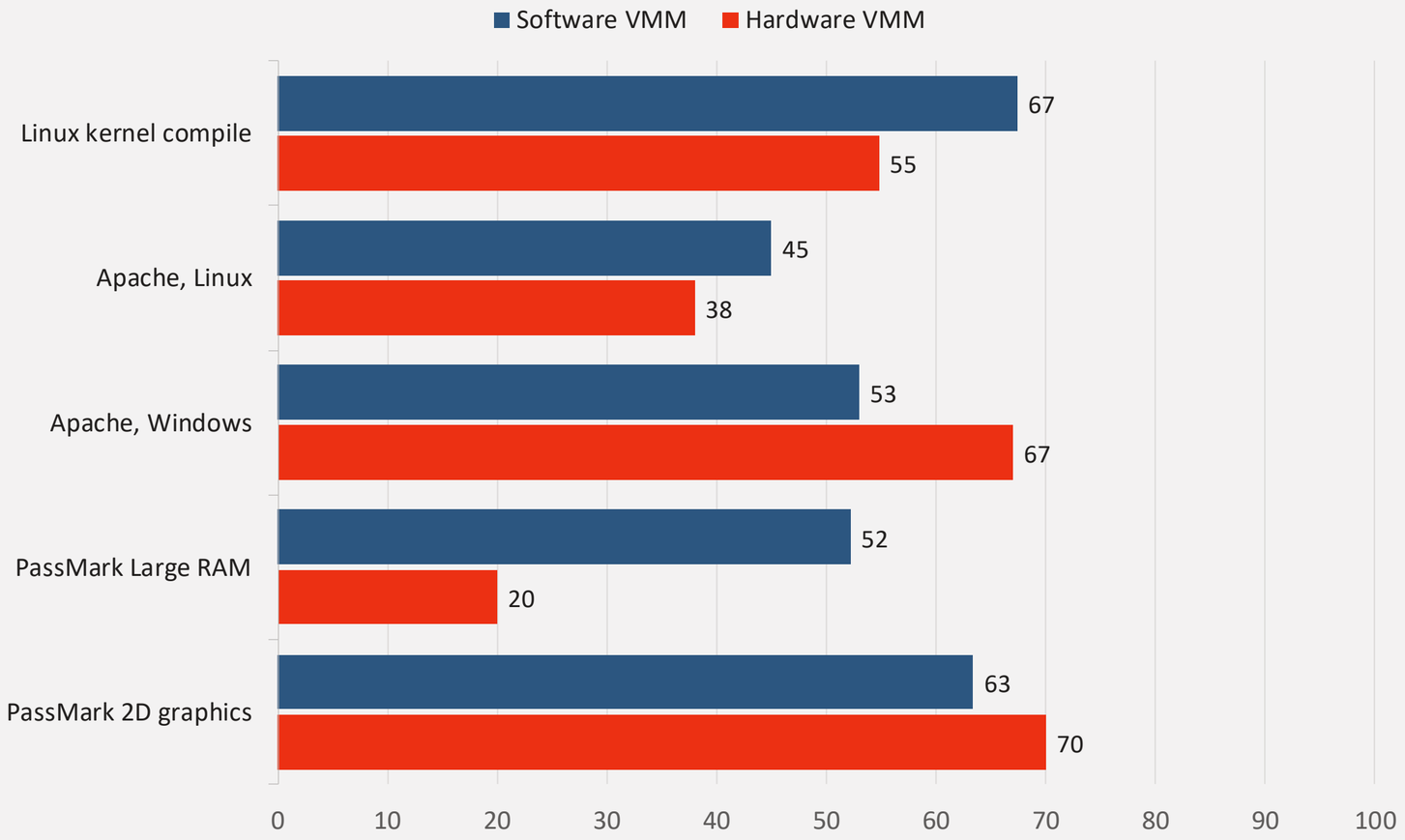
SPECint 2000 on Linux and SPECjbb 2005 on Windows, as % of native speed (higher is better).



- Average slowdown: about 4% for the software VMM, 5% for the hardware VMM. mcf ran faster than native on both.

Real workloads: mostly a software win

% of native speed (higher is better).



SOFTWARE WINS

Compiling, paging, multi-process Apache: lots of address-space work.

HARDWARE WINS

One address space, many system calls, few I/O exits.

KERNEL COMPILE

Overhead: 128 s software vs 219 s hardware.

forkwait magnifies the difference

To isolate process creation, the authors fork and wait 40,000 times.

```
int main(int argc, char *argv[]) {
  for (int i = 0; i < 40000; i++) {
    int pid = fork();
    if (pid < 0) return -1;
    if (pid == 0) return 0;
    waitpid(pid);
  }
  return 0;
}
```

6.0 s	Native
36.9 s	Software VMM · 6.1× native
106.4 s	Hardware VMM · 17.7× native

- The hardware VMM adds about 4.4× as much overhead as the software VMM: 100 s against 31 s.

Nanobenchmarks: one sensitive operation at a time

CPU cycles on the Pentium 4 (lower is better). The faster VMM is shaded.

OPERATION	NATIVE	SOFTWARE VMM	HARDWARE VMM	WHY
syscall	242	2,308	**246**	BT adds a layer and a privilege switch
in (port 0x80)	3,209	**216**	15,826	BT: a few instructions; HW: a full exit
%cr8 write	150	**35**	177	HW exits only when unmasking an interrupt
call / ret	11	51	**11**	BT pays for indirect control flow
page fault	1,093	**3,927**	11,242	reaching the VMM is cheaper than an exit
divide by zero	889	3,223	**1,014**	a fault with no MMU work
PTE write	1	**391**	12,733	callout vs an exit on every traced write

Why PTE writes decide the bottom line

Overheads during one 64-bit Windows XP boot and halt.

> 1 million

PTE writes in one boot

system calls were about as frequent

12,733 vs 391

cycles per traced PTE write

hardware VMM vs software VMM

≈ 3.3 s vs 0.1 s

of PTE overhead at 3.8 GHz

our estimate from the cycle counts

SYSTEM CALLS DON’T MAKE UP FOR IT

Software pays about 2,000 extra cycles per system call; hardware pays about 12,700 per PTE write.

TRANSLATION ITSELF IS CHEAP

About 2,300 cycles per guest instruction, and the cache captures the working set, even for a cold boot.

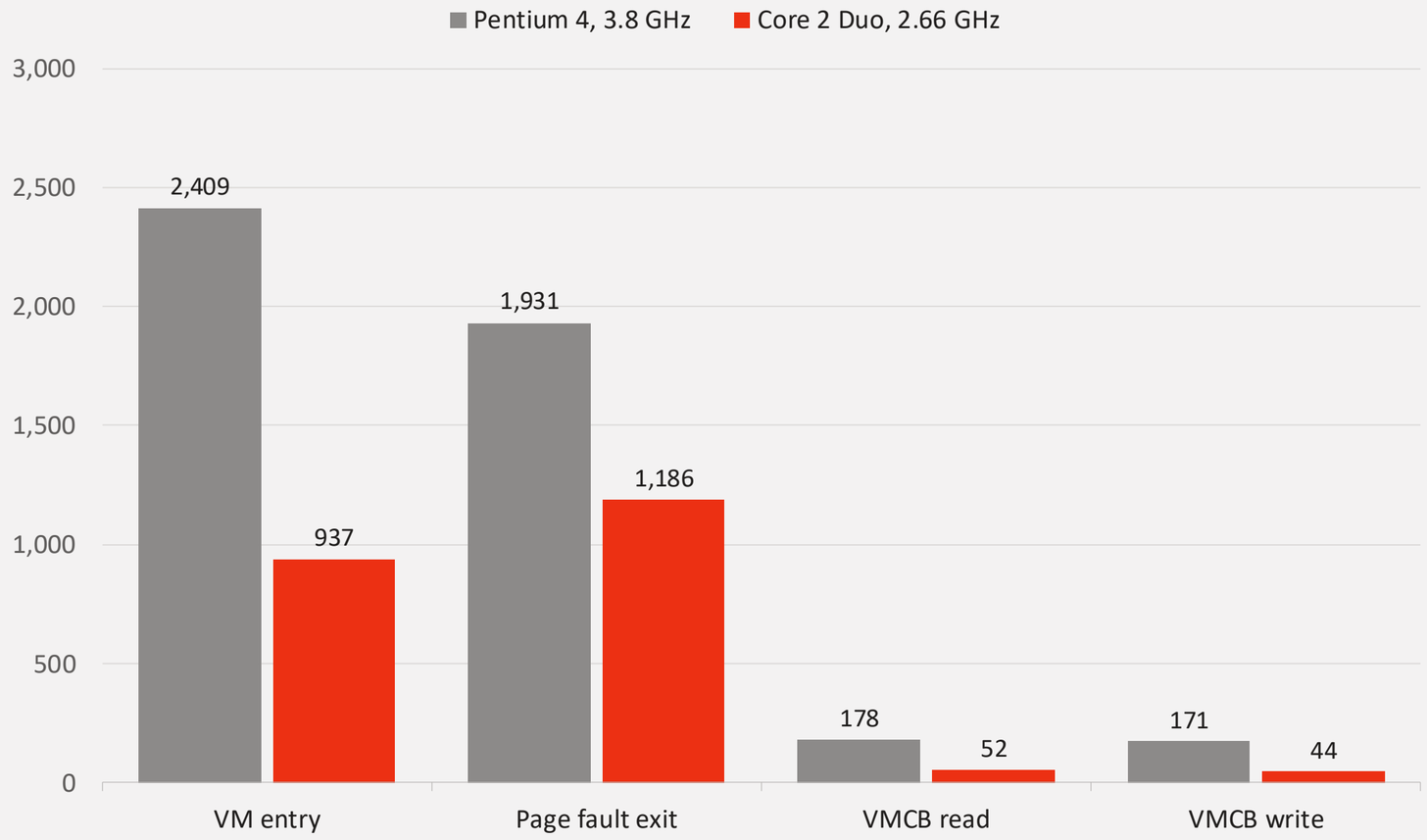
- The hardware wins only where the guest enters the kernel often but rarely does privileged work there.

07

Looking ahead

Faster exits help, but not enough

Pentium 4 against a pre-release Core 2 Duo (cycles, lower is better).



FORKWAIT VS NATIVE

Hardware VMM

17.7× → 14.8×

Software VMM

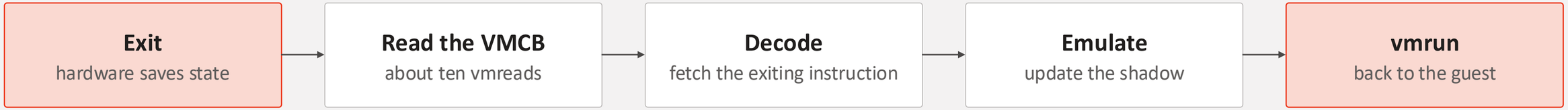
6.1× → 7.0×

- P4 on the left of each arrow, Core on the right.

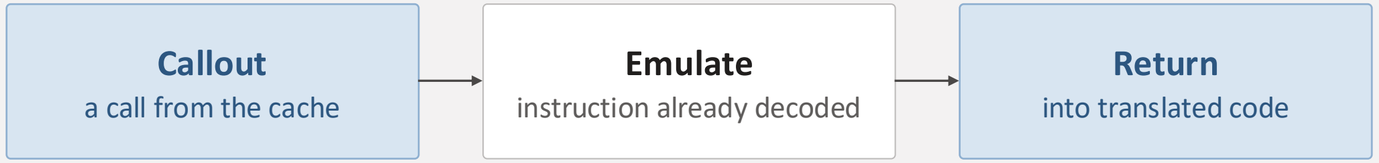
Even a free exit leaves a gap: the VMM starts from nothing

After an exit the hardware VMM must rebuild what the guest was doing. A BT callout already knows.

HARDWARE VMM, AFTER AN EXIT



SOFTWARE VMM, AFTER A CALLOUT



Two steps shorter, and a plain call instead of a guest-to-host round trip.

Three ways to close the gap

01

Trace less

Let shadows go briefly stale, as Workstation 2.0 did, and catch up on %cr3 reload. Narrows the gap; doesn’t close it.

02

Hybrid VMM

Pick BT or hardware by guest behavior: hardware for system calls, BT for the MMU. Heuristics still immature.

03

MMU hardware

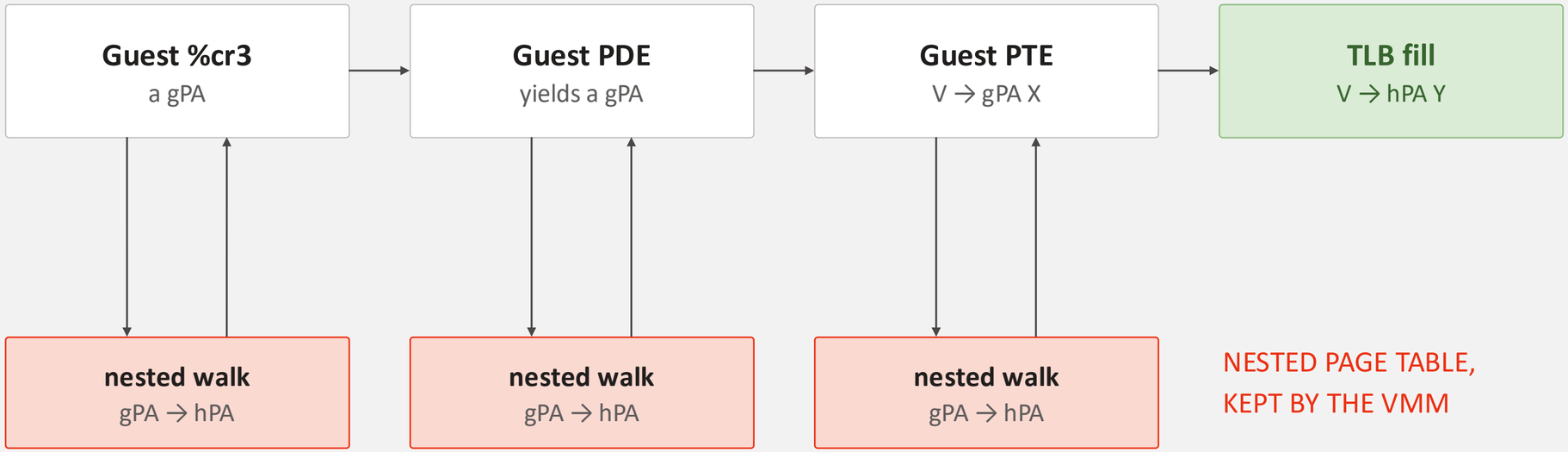
Nested paging (AMD) and EPT (Intel): let hardware walk the guest’s page tables itself. The authors’ bet.

WHY THE AUTHORS FAVOR MMU HARDWARE

IBM’s System/370 solved the same three-way trade-off in hardware, and both the AMD and Intel proposals resemble that design.

Nested paging: the hardware walks both tables

GUEST PAGE TABLE, WALKED BY HARDWARE



Every guest-physical address the walk meets goes through the nested table first.

NESTED PAGE TABLE, KEPT BY THE VMM

WHAT DISAPPEARS

Traces, hidden page faults and exits on %cr3: the three-way trade-off is gone.

WHAT IT COSTS

A TLB miss can take up to M×N memory references for M- and N-level tables.

WHY THAT'S ACCEPTABLE

Large TLBs and caches should absorb it, as they do for ordinary paging.

08

Takeaways

Four things to remember

01

Hardware assist isn’t free speed

First-generation VT-x and SVM rarely beat a mature binary translator, and lost badly on process creation and paging.

02

Count exits, not instructions

A hardware VMM is native until it exits, and each exit costs thousands of cycles.

03

The MMU was the missing piece

Shadow paging turned PTE writes and %cr3 loads into exits. Nested paging targets exactly that.

04

Leave room for software

The authors ask for hardware primitives that complement software techniques, not replace them.

For discussion

01 The software VMM is not safe on 64-bit Intel. How much does that weaken the comparison?

---

02 The hardware VMM reused an MMU design tuned for BT, where traces are cheap. Is this a fair test of the hardware?

---

03 Nested paging makes every TLB miss longer. Which workloads might end up slower than with shadow paging?

---

04 The paper sets aside paravirtualization because it needs guest changes. virtio is a guest change. Why did it win for I/O?