

Lecture 06 - Safe Hardware Access with the Xen Virtual Machine Monitor

Isolating unmodified device drivers in their own domains, and sharing them safely across operating operating systems. Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Mark Williamson. University of Cambridge and Intel Research Cambridge, OASIS 2004.

Contents

01 Why drivers need isolation

02 The safe hardware model

03 I/O Spaces

04 Device channels

05 Sharing and restarting drivers

06 Evaluation

07 Takeaways

01

Why drivers need isolation

Xen's target: shared utility computing

Xen runs multiple operating systems concurrently on commodity x86 hardware. The paper targets two environments where machines are shared across parties who do not trust each other.

ORGANIZATIONAL DATA CENTERS

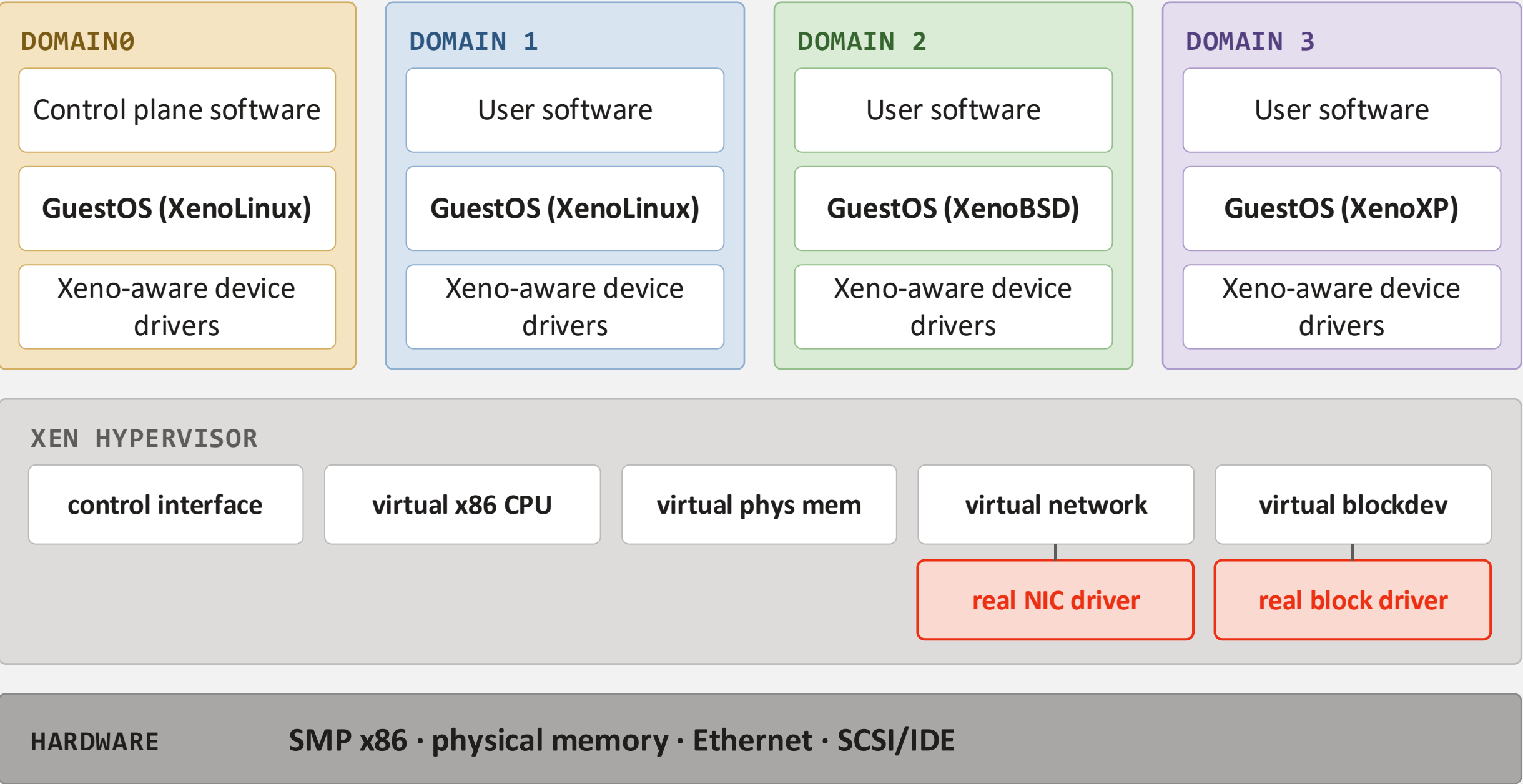
A large cluster of physical machines is shared across different administrative units of one organization.

GLOBAL COMPUTE UTILITIES

Unrelated customers lease distributed resources to deploy their own services, as in the XenoServers initiative.

Both require **reliable execution, hard isolation, and accounting** for the physical resources underneath.

The original Xen architecture (SOSP 2003)



Redrawn from Barham et al., SOSP 2003, Figure 1. Red boxes added: the real drivers inside Xen.

DOMAIN0

Created at boot and allowed to use the control interface: it hosts the management software that creates domains and configures their devices.

IDEALIZED DEVICES

Guests run small Xeno-aware drivers for virtual network and block devices, moving data over shared-memory I/O rings.

THE CATCH

The drivers for the real hardware were built into Xen itself, so one driver bug could take down every domain.

Drivers are the weak link

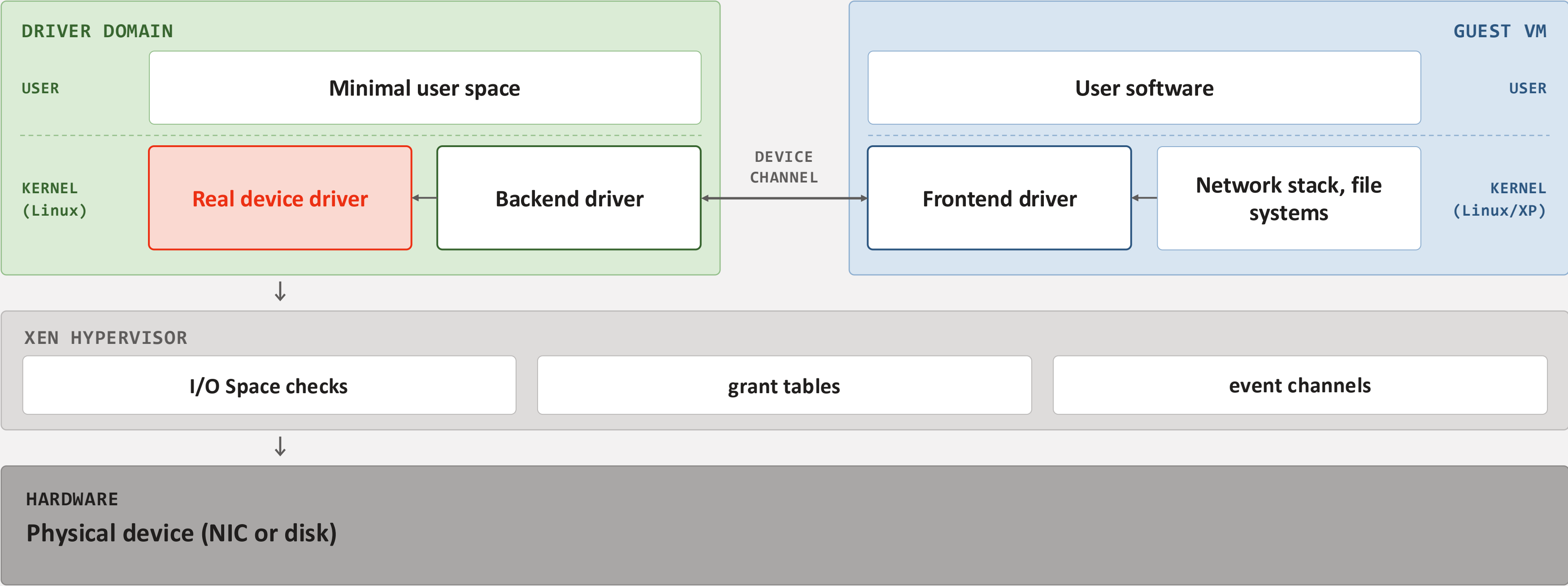
The original Xen design placed device-driver code inside the hypervisor and exported safe virtual devices. That made every driver bug a hypervisor bug. The problem compounds:

- device drivers are a well-known major source of bugs and system failures;
- sharing one device across many OS instances drastically raises the stakes of a single driver fault;
- the variety of PC hardware rules out rewriting drivers: existing, unmodified driver code must be reused.

The answer is to run unmodified drivers in isolated **driver domains**: in essence, driver-specific virtual machines.

Moving the driver out of the hypervisor

SPLIT DRIVER · FRASER ET AL., OASIS 2004



Two benefits of a driver in its own VM

A Safe Hardware Interface limits each driver to the specific memory, interrupts, and I/O ports its device needs. Placing drivers in separate virtual OS instances then yields two benefits.

ISOLATION

Drivers are fenced off from the rest of the system. They may crash, may crash, or be deliberately restarted, with minimal impact on impact on running OSes.

A UNIFIED INTERFACE

Each driver is reached through one interface, so it can be shared safely by many OSes at once, and one low-level driver serves every paravirtualized OS.

- Earlier partitioning systems depended on specific hardware support or ignored enterprise-class performance. This work targets commodity PC hardware without specialized support.

Where earlier work falls short

APPROACH	EXAMPLES	GAP THE PAPER IDENTIFIES
Driver interposition	Swift et al. (Nooks)	Improves Linux driver reliability but sacrifices performance drastically; focused on one OS instance.
Extensible kernels	SPIN, VINO	Admit untrusted code into a single existing OS rather than isolating it.
Device-managing layers	Nemesis, Exokernel	Explored the idea, but not applied to existing systems.
Commercial VMMs	VMware ESX Server	Shares devices across OSs; not designed around driver dependability.
Standard driver interfaces	UDI	Drivers still run in the OS's protection domain; the interface becomes the union of all drivers.
Microkernel driver reuse	Fluke, L4, LeVasseur et al.	Wraps legacy drivers for one new OS rather than sharing isolated drivers.

02

The safe hardware model

Three parts, one path from OS to device

Figure 1 of the paper stacks operating systems over isolated device drivers, which sit on a safe hardware interface above the physical devices.

THE DATA PATH



I/O SPACES

Vertical isolation: each driver–device pair gets only its own registers, interrupts, and memory.

UNIFIED INTERFACES

One idealized interface per device class, so a single driver source serves every OS.

DEVICE MANAGER

A BIOS-like extension that boots driver domains and exports common control interfaces.

Three isolation requirements

REQUIREMENT	RESOURCE	WHAT MUST BE ENFORCED	ENFORCED BY
1 · Driver isolation	Memory	Driver executes in its own logical fault domain	VMM
	CPU	Scheduling prevents excessive consumption	VMM
	Privilege	Access to the instruction set is limited	VMM
2 · Driver → device	I/O registers	Access restricted to permitted ranges	Xen
	Interrupts	Driver may mask and receive only its device's interrupt	Xen
3 · Device isolation	Memory	No DMA to arbitrary host memory	IOMMU
	Other devices	No access to arbitrary other devices	Chipset

■ Requirement 3 cannot be met in software on 2004 x86 hardware. The paper builds everything else and leaves a clean hook for an IOMMU.

Why a VM rather than a fault domain?

Earlier driver isolation placed driver code in a separate **logical fault domain**: virtual-memory protection between the driver and the rest of the kernel.

That is only a partial solution. It primarily protects memory, but Requirement 1 also covers scheduling and access to privileged instructions.

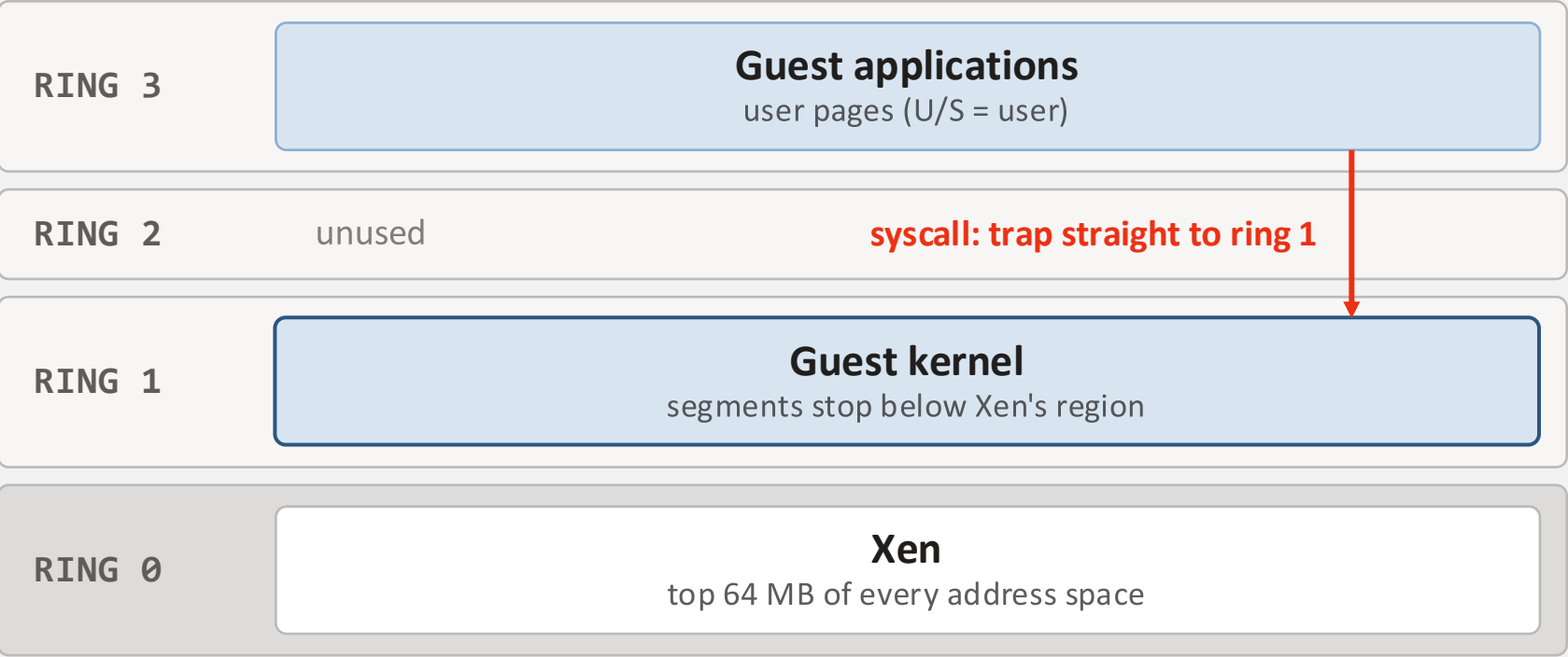
WHAT THE VMM ADDS

Xen tracks and retains full control of each driver's CPU and memory use: the same guarantee an OS gives its processes.

A driver that writes outside its heap damages only its own VM, and a misbehaving driver can be restarted with minimal impact on the rest of the system.

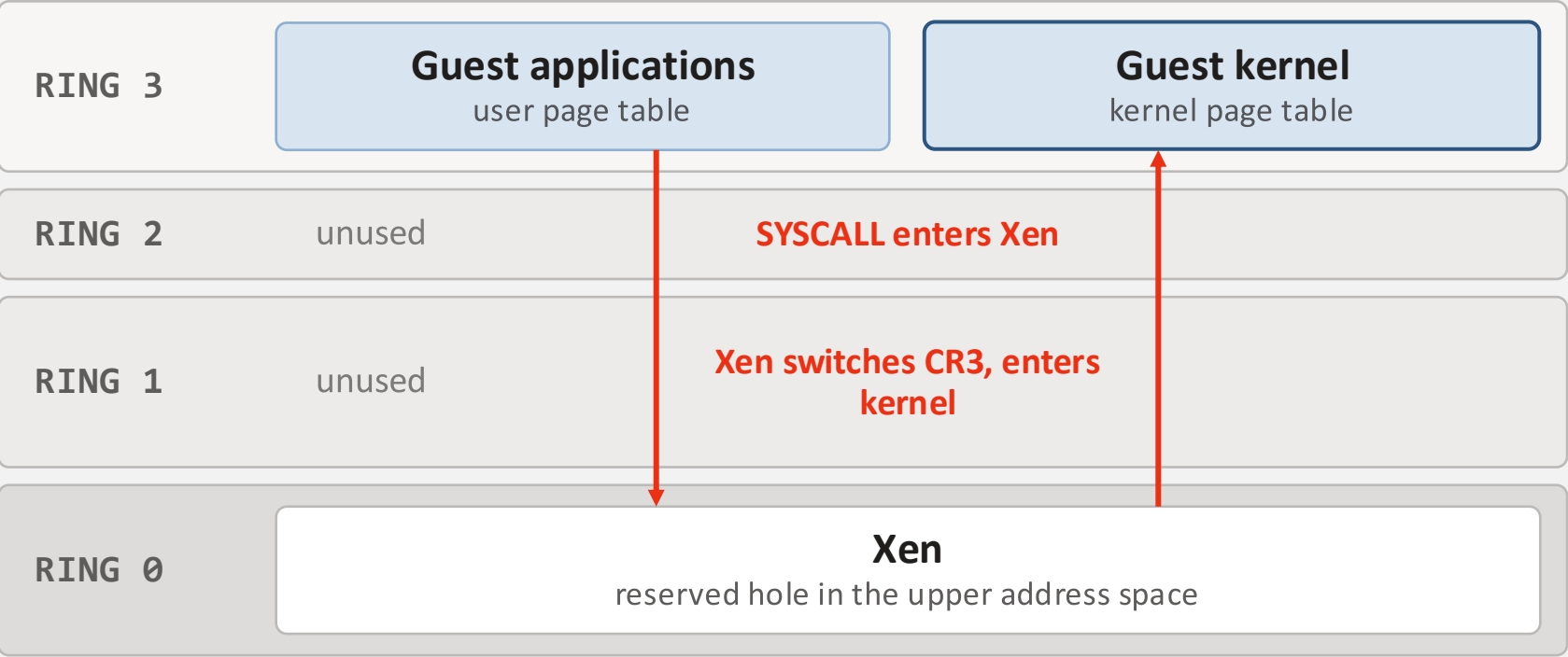
Rings: how PV Xen ran guests on x86-32 and x86-64

X86-32 · KERNEL IN RING 1



Apps vs kernel: the page U/S bit.
Kernel vs Xen: segment limits, since paging counts ring 1 as supervisor.

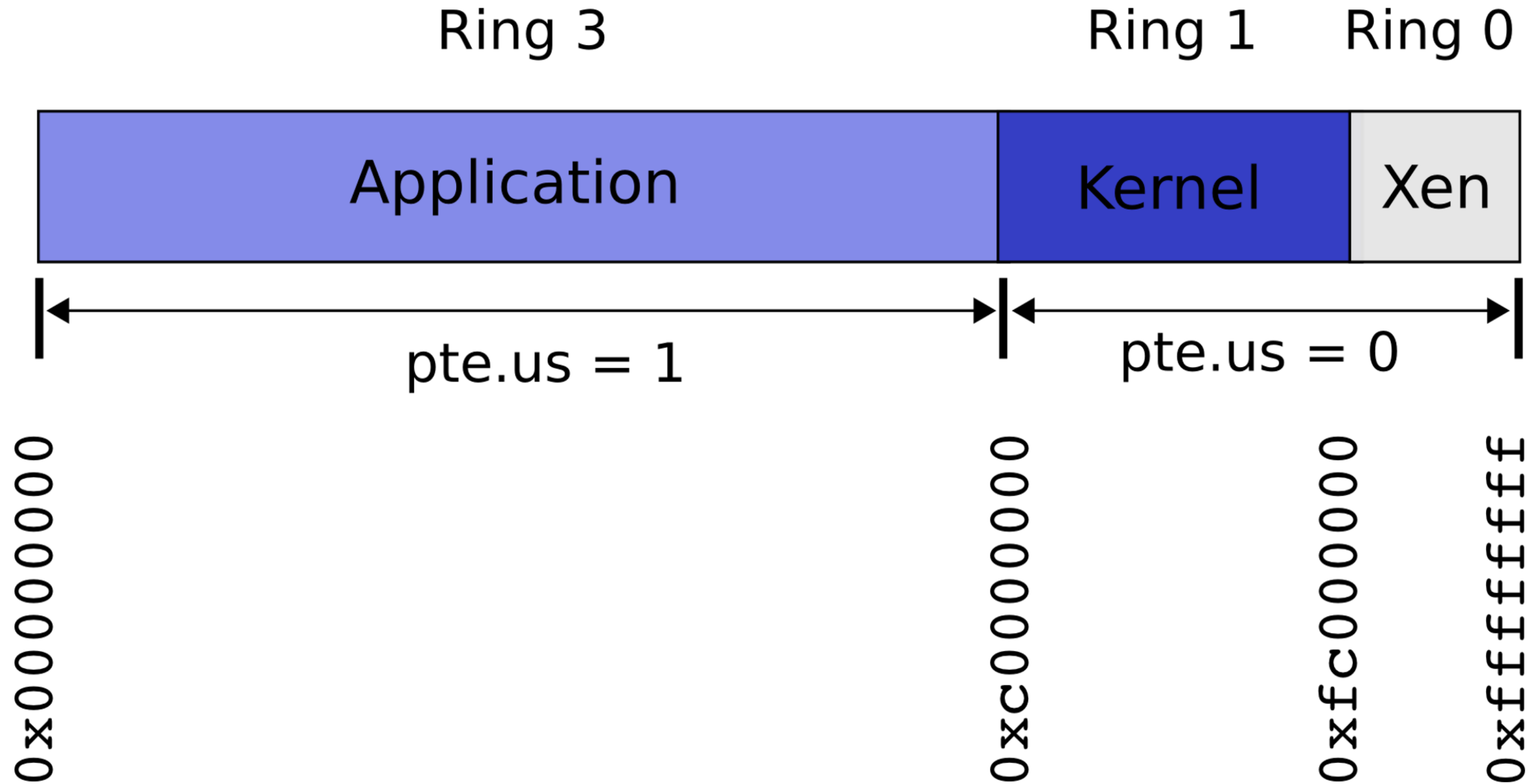
X86-64 · KERNEL AND APPS BOTH IN RING 3



Why not ring 1: long mode drops segment limits, and paging treats rings 0–2 alike.
Apps vs kernel: separate page tables, switched by Xen.
Kernel vs Xen: the U/S bit, since the kernel now runs as user.

Every 64-bit PV syscall and kernel–user switch now crosses Xen and flushes the TLB. That cost pushed Xen toward VT-x and AMD-V, where the guest kernel runs in ring 0 of non-root mode, and later PVH.

Hypervisor protection



Unified interfaces: one driver per device class

The PC standardizes hardware interfaces but not the interface to system software. The paper defines an idealized, high-level interface per device class, and deliberately avoids **interface unioning**: a narrower interface rather than the aggregate of every existing driver.

OS vendors write one small, quality-controlled driver per class. Hardware vendors implement the interface once and support every OS. Efficient device channels need three things:

NO COPIES

Data moves by reference to guest pages, never by copying.

BACK PRESSURE

A full ring throttles the data source.

ASYNC NOTIFICATION

A flexible, asynchronous event primitive signals the other end.

The architecture separates what is essential, hardware isolation, from what is merely desirable, a shared driver interface.

03

I/O Spaces

A vertical protection domain

An **I/O Space** is a protection domain that can be assigned the physical resources for one specific driver–device interaction: memory, device registers, and interrupts. x86 provides no hardware support for limiting a driver to one device, or a device to part of memory.

- Driver-to-device isolation (Requirement 2) is implemented inside Xen, emulating the I/O-Space function of a next-generation chipset.
- Device-to-host isolation (Requirement 3) needs chipset support; the design shows the minimal change needed to use it.
- Configuration space is virtualized so each driver sees only the devices it may touch.

■ The authors argue this is the strongest isolation possible without hardware modifications.

Restricting I/O registers

MEMORY-MAPPED I/O

Xen already validates every address-space update. It extends those checks so a domain may map **non-RAM page frames** holding its own device's registers.

Page granularity is enough: register blocks of different devices are conventionally aligned on at least a page boundary.

PORT I/O

x86 has a 16-bit I/O port space that can be restricted per port by an access bitmap the processor checks on every access.

Xen rewrites the bitmap when context-switching between domains.

```
domain requests mapping of frame F:
```

```
  F is RAM    → allowed only if the domain owns F
```

```
  F is MMIO   → allowed only if F belongs to the domain's device
```

```
domain executes IN/OUT on port P:
```

```
  CPU checks P in the current domain's I/O bitmap
```

How the CPU checks the I/O bitmap

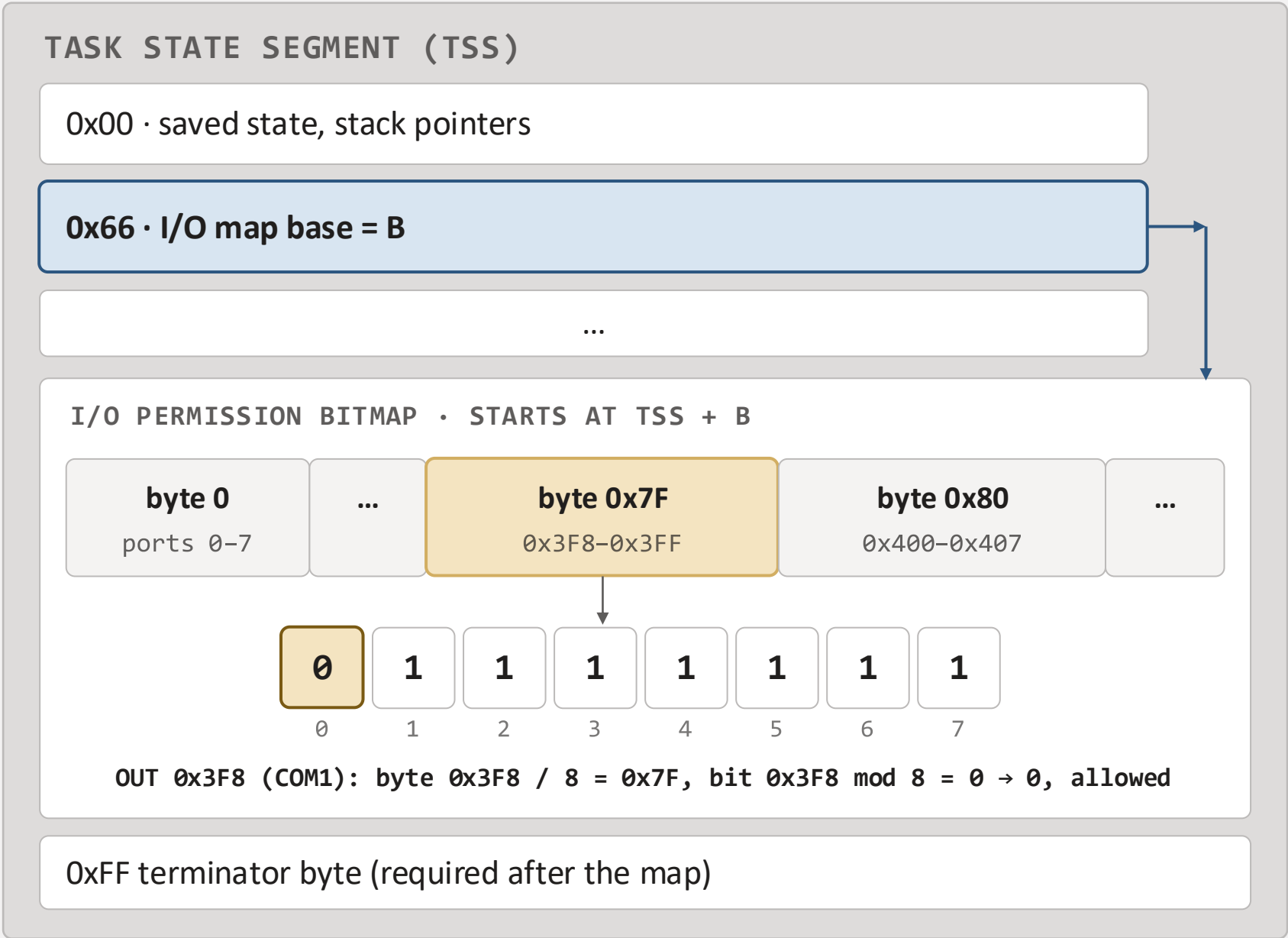
- 1

IOPL gate. If $CPL \leq IOPL$ (EFLAGS bits 12–13), the access proceeds and the bitmap is never read.
- 2

Locate the map. The 16-bit field at TSS offset 0x66 gives the bitmap's offset from the TSS base: one bit per port.
- 3

Check every port touched. A 1-, 2- or 4-byte access checks 1, 2 or 4 consecutive bits; 0 allows, 1 denies.
- 4

Fault or proceed. Any 1 bit, or a map byte beyond the TSS limit, raises #GP(0) instead of performing the I/O.
- Ports past the end of a shortened map are denied. A full map covering all 65,536 ports is 8 KB.



Multi-byte accesses, Xen, and VT-x

```

if (CPL > IOPL || vm86_mode) {
    off = TSS.io_map_base + port / 8
    if (off + 1 > TSS.limit) → #GP(0)
    bits = read16(TSS.base + off) >> (port % 8)
    mask = (1 << size) - 1    // size = 1, 2, 4
    if (bits & mask)          → #GP(0)
}
perform the I/O

```

- A 2-byte access to port 0x3FF straddles two map bytes: bit 7 of bit 7 of 0x7F and bit 0 of 0x80. Reading 16 bits covers that case, that case, and the trailing 0xFF byte keeps the read from picking up stray allowed bits.

XEN'S USE (2004)

Guest kernels run in ring 1 with IOPL 0. Since CPL 1 > IOPL 0, every port every port access falls through to the bitmap. Xen keeps one bitmap per bitmap per domain, describing its I/O Space, and rewrites the TSS bitmap TSS bitmap on each domain switch.

CAVEAT

The bitmap is per TSS, not per ring: it cannot tell a guest kernel from that guest's own processes. Later Xen versions deny all ports in hardware, catch the #GP, and check the domain's bitmap in software.

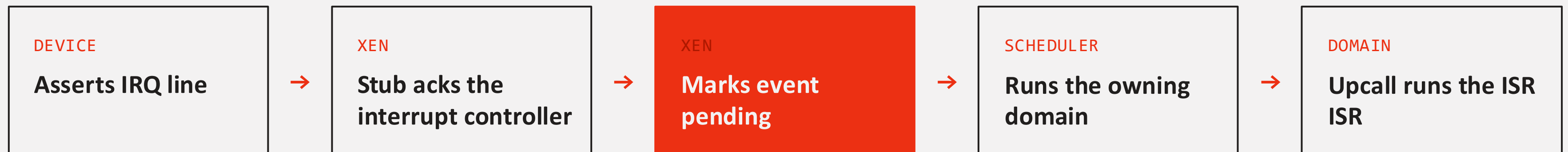
VT-X

The VMCS adds I/O bitmaps A (0x0000–0x7FFF) and B (0x8000–0xFFFF). A set bit causes a VM exit instead of #GP. The guest's own IOPL and TSS check runs first, and its #GP takes priority.

Interrupts become events

A device's interrupt runs a stub in Xen rather than entering the owning domain directly. Xen keeps tight control of when the driver's interrupt service routine runs.

PHYSICAL INTERRUPT TO DRIVER ISR



Taking the interrupt in Xen gives a timely acknowledgement to the interrupt controller, which Xen always owns, and allows the address-space switch when another domain is running.

Event channels: two bits in a shared page

Each domain can be allocated up to 1024 event channels. Each is a pair of flags in a memory page shared between the domain and Xen.

FLAG	WRITTEN BY	EFFECT
Pending	Xen	Signals an event. Only the first transition to pending triggers an upcall (and wakes a blocked domain); later sets are ignored until the domain clears it.
Mask	Domain	Suppresses notification: no upcall and no wake-up when a masked event becomes pending.

Setting the mask before clearing the pending flag lets a domain avoid needless upcalls for a partially handled event source.

- Level-triggered lines stay masked at the controller until the domain calls down to unmask. An unshared line can usually be reconfigured as edge-triggered, removing that downcall.

The shared-interrupt caveat

When several devices share a line, Xen must wait for an unmask downcall from **every** domain that manages one of them before it unmask the line.

WHY ISOLATION IS IMPERFECT

A domain that never unmask the shared line prevents the other domains from receiving device notifications. Xen cannot guarantee isolation for a domain given a shared interrupt.

WHY IT IS TOLERABLE

Shared interrupts are rare on servers, whose IRQ-steering and interrupt controllers have a pin per device. Message-based interrupts in PCI Express make the problem disappear.

Device-to-host: the DMA protection gap

INTERRUPT ASSERTION: CONTAINED

Except for shared lines, each device is wired to its own controller pin, so it physically cannot assert another line. Xen owns the controller and can guard against IRQ storms.

DMA: NOT CONTAINED

A driver blocked from writing a page with the CPU can program its device to DMA to that page instead. Commodity 2004 hardware offers no good defence.

```
driver:    *(foreign_page) = x           → blocked by Xen page-table checks
```

```
driver:    dev.dma(dst = foreign_page) → device writes it anyway
```

The fix is an **IOMMU** in the chipset, configured by Xen with the same ownership checks it applies to the CPU's MMU.

Virtualizing PCI configuration space

PCI configuration space is how devices are detected and configured. Native access methods are unsafe: protected-mode BIOS calls, or a small I/O-port window shared by every device. Domains therefore get only a virtualized interface provided by Xen.

FILTER WRITES

A domain may read and write registers only of devices it owns.

FILTER THE VIEW

A hardware probe detects only the devices the domain may access.

VALIDATE CHANGES

Xen rejects moves of an I/O-register block that could collide with another device.

04

Device channels

Isolated driver domains and device channels

A guest could run its own drivers inside an I/O Space, but the preferred structure runs each driver in an **isolated driver domain (IDD)**. Guests talk to IDDs over device channels: point-to-point links on which either side can send asynchronous messages.

- The system controller creates a control channel with every domain it starts, and uses it to introduce IDDs and guests to each other.
- Messages travel through shared pages allocated by the guest and also mapped into the IDD.
- Xen has no concrete notion of a channel; it only permits restricted sharing of pages between domains.

■ Channels are built from two primitives: **grant-based page sharing** and **event channels**.

Grant tables: asymmetric, transitory sharing

Every page has at most one owner. To map a foreign page, a domain presents a **grant reference** instead of a page number: the granting domain's identity plus an index into its private grant table.

A usage count on each active entry stops the granter from reusing the reference until every foreign mapping of it is destroyed.

grant table entry (in domain A):

(grant, D, P, R, U)

D may map page P

R = read-only, U = in use (Xen)

B presents (A, G):

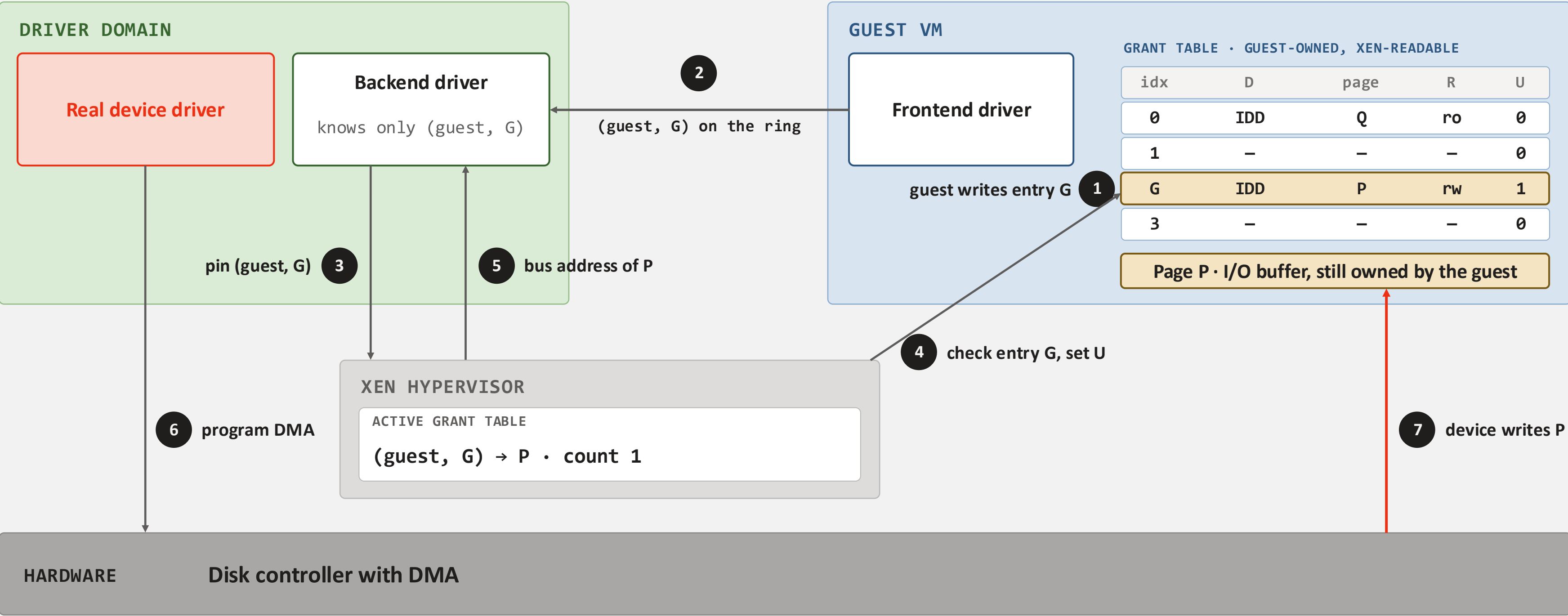
G in A's active table? → reuse

else check type = grant,

D = B, R ok for access

→ copy into active table, set U

Grant tables in action: a disk read into page P



A DMA request through a device channel

The guest asks the IDD to transfer data into buffer page P2. Seven steps from Figure 2:

- 1** The guest places a grant reference (GR) for page P2 on the device channel.
- 2** The IDD removes GR from the ring.
- 3** The IDD sends a pin request containing GR to Xen.
- 4** Xen looks up GR in the active grant table.
- 5** If needed, Xen validates GR against the guest's grant table.
- 6** Xen acknowledges the pin, returning an address for P2.
- 7** The IDD sends the DMA request to the device.

Descriptor rings

Asynchronous transfers use a shared ring with two pairs of producer–consumer indexes. Requests and responses share the same ring; a unique identifier on each allows the IDD to reorder.

```
requests:  guest advances req_prod   →   IDD advances req_cons
responses: IDD advances rsp_prod     →   guest advances rsp_cons
notify:    bidirectional inter-domain event channel
```

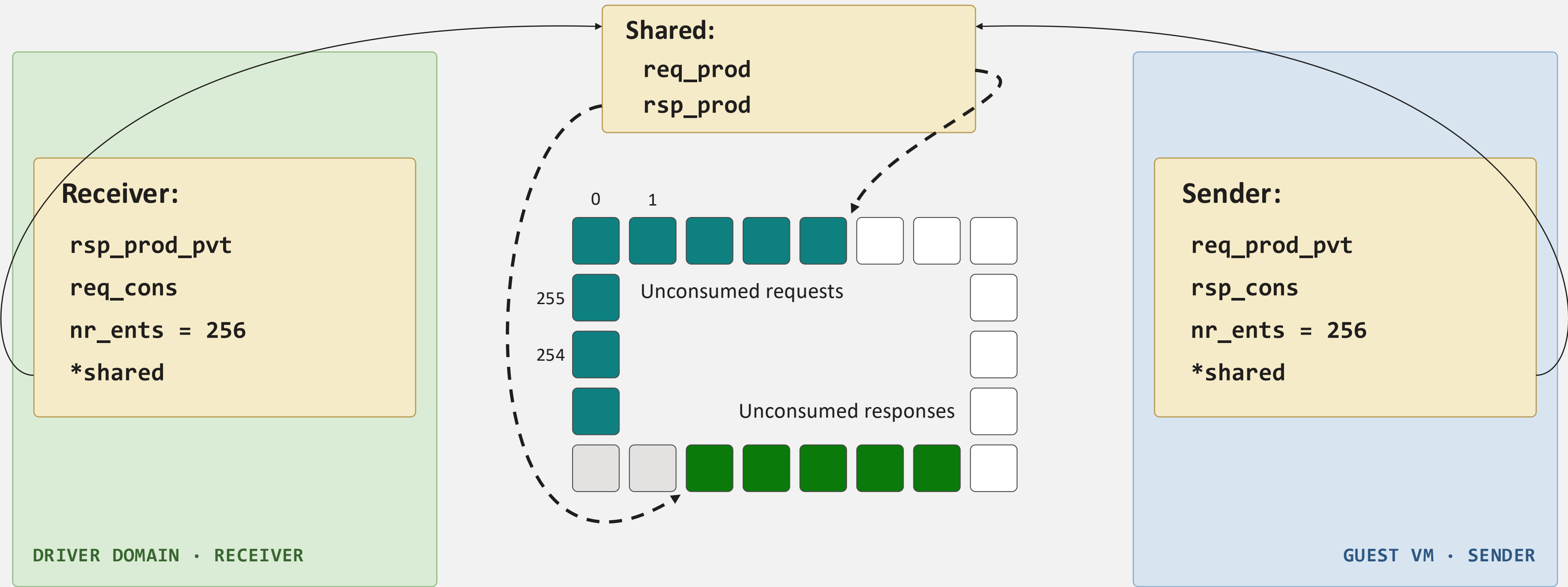
INTER-DOMAIN EVENT CHANNELS

Like interrupt channels, but notified by the domain at the other end rather than by Xen, and each end can notify or mask the other.

DECOUPLED NOTIFICATION

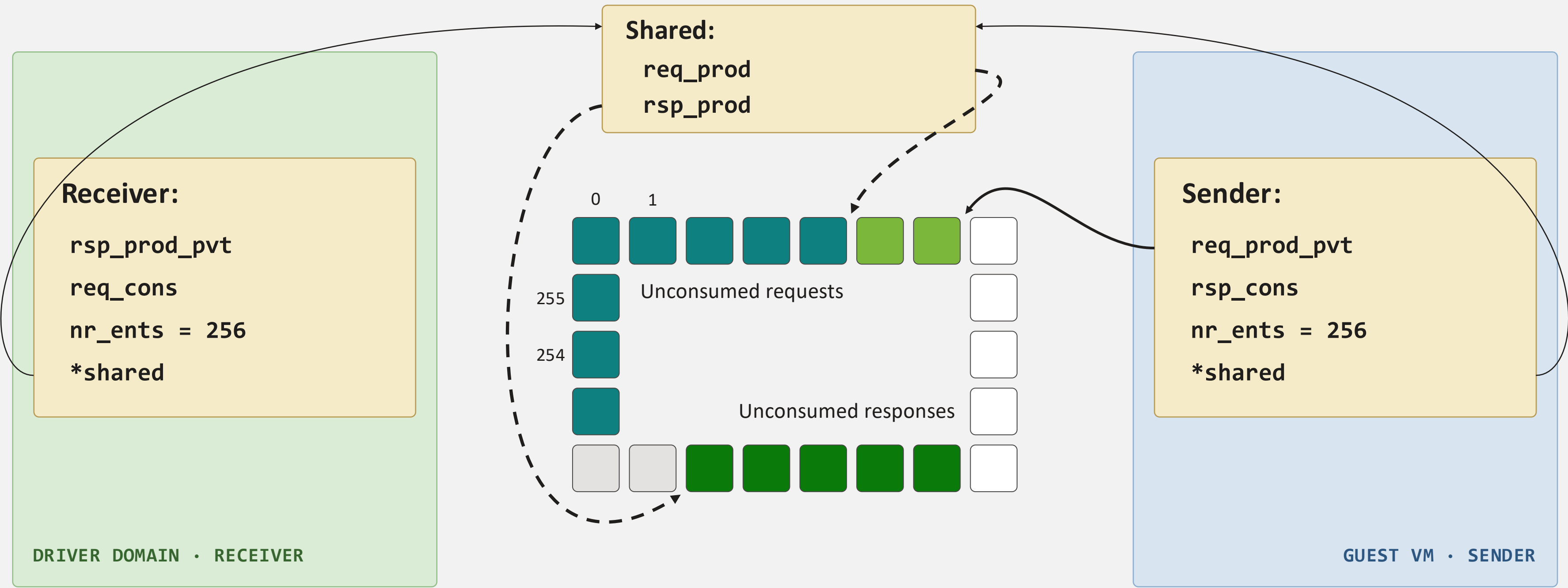
A guest can queue many requests before notifying; it can also ask for a response notification only after a threshold. Each side balances latency against throughput.

Shared rings



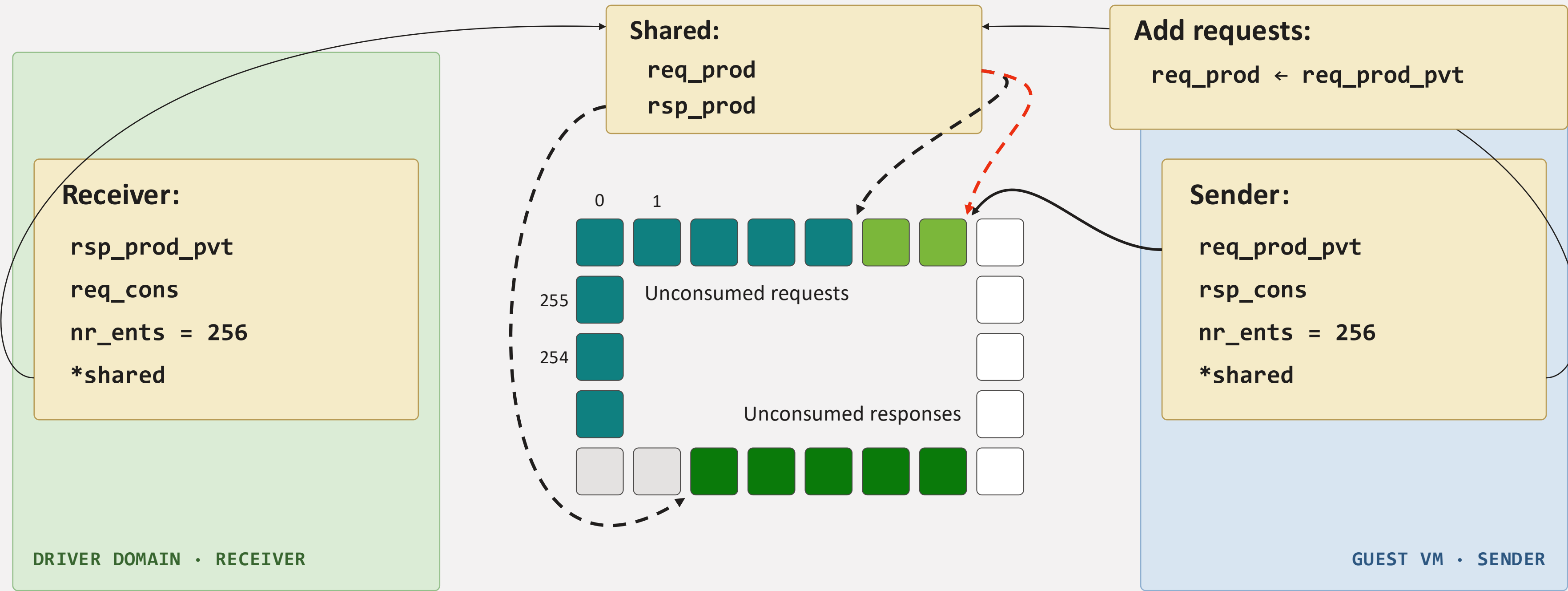
Step 1 of 4 · The ring page holds the shared indices; each side keeps private ones. Teal: requests not yet consumed. Dark green: responses not yet consumed. Gray: requests taken by the requests taken by the receiver, response still pending.

Shared rings



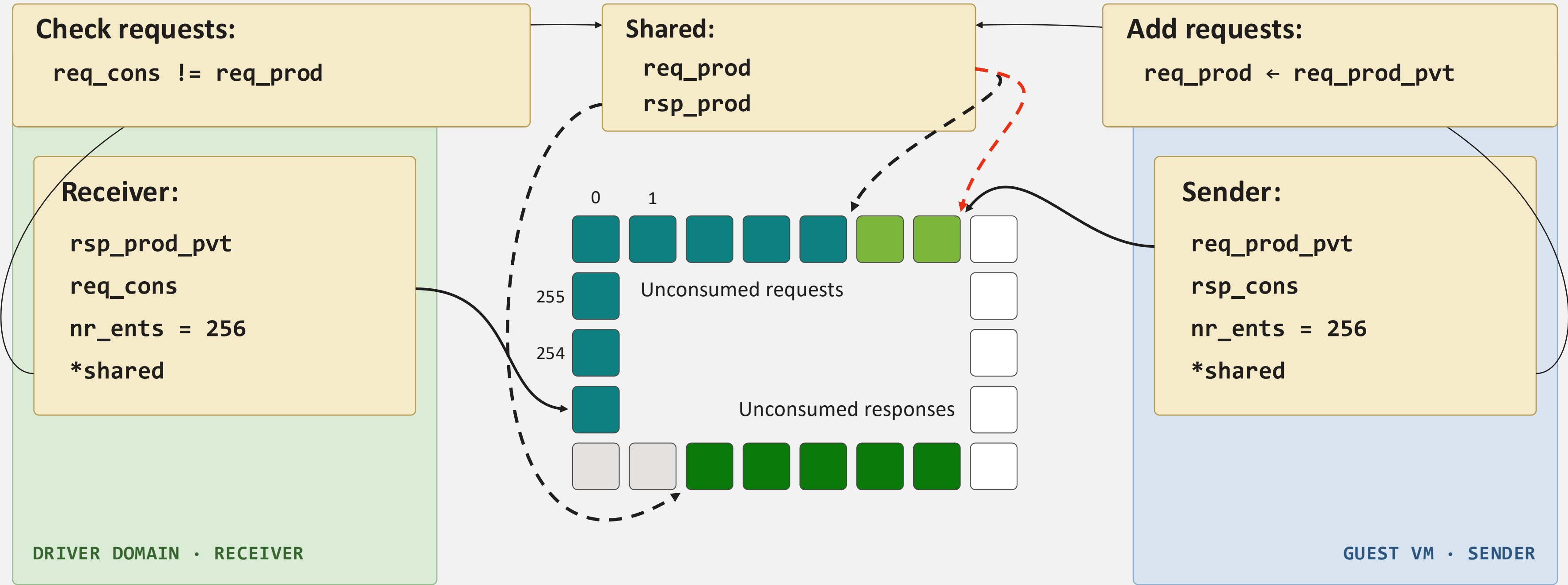
Step 2 of 4 · The sender fills new requests (light green) and advances its private req_prod_pvt. The receiver cannot see them yet.

Shared rings



Step 3 of 4 · Publishing is one store, `req_prod ← req_prod_pvt`, after a write barrier. The sender then notifies over the event channel if the receiver asked to be woken.

Shared rings



Step 4 of 4 · The receiver wakes or polls, sees `req_cons != req_prod`, and consumes requests from `req_cons` up to `req_prod`.

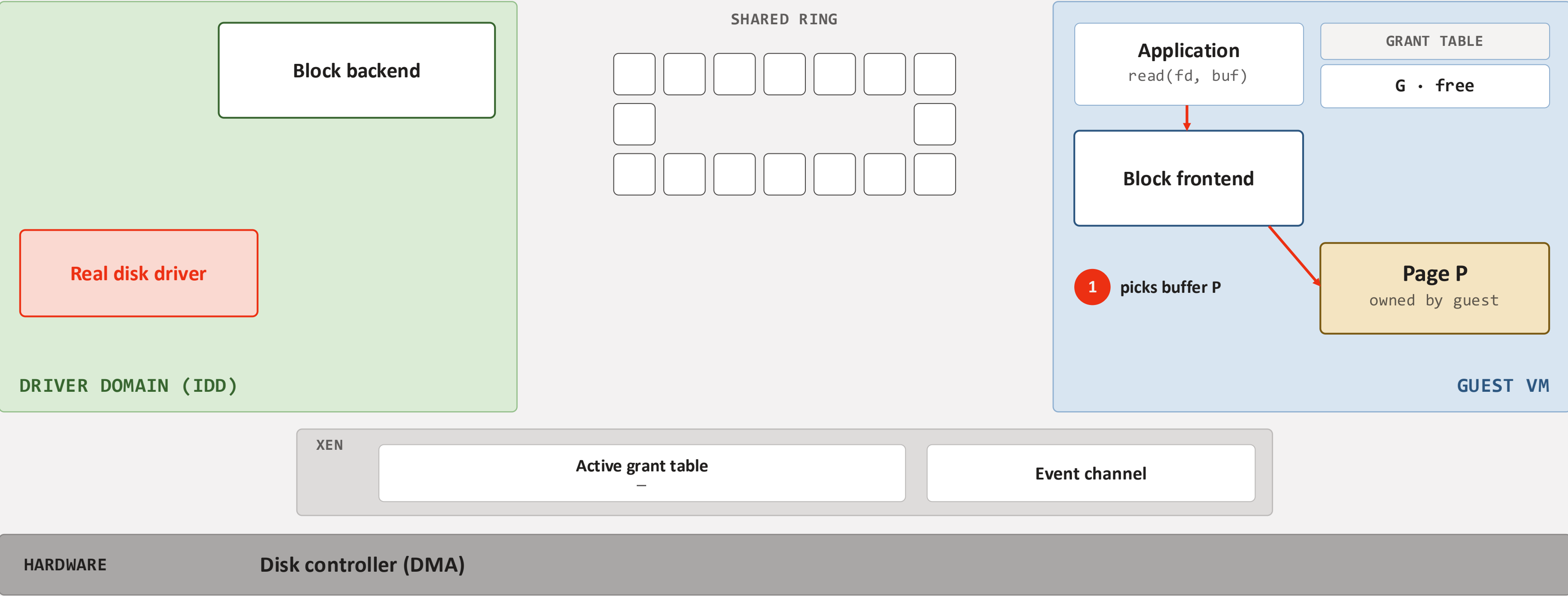
Data transfer and pinning

Small payloads can live in descriptors. High-bandwidth DMA devices instead use buffers the guest allocates out of band and references indirectly, always by grant reference rather than by address.

- Before programming DMA, the IDD must **pin** the buffer by presenting the grant reference to Xen.
- Xen applies its address-space rules: the page must belong to the right domain and must not circumvent memory management, for example by targeting its own page tables.
- The pin response returns a bus address the device can use directly.

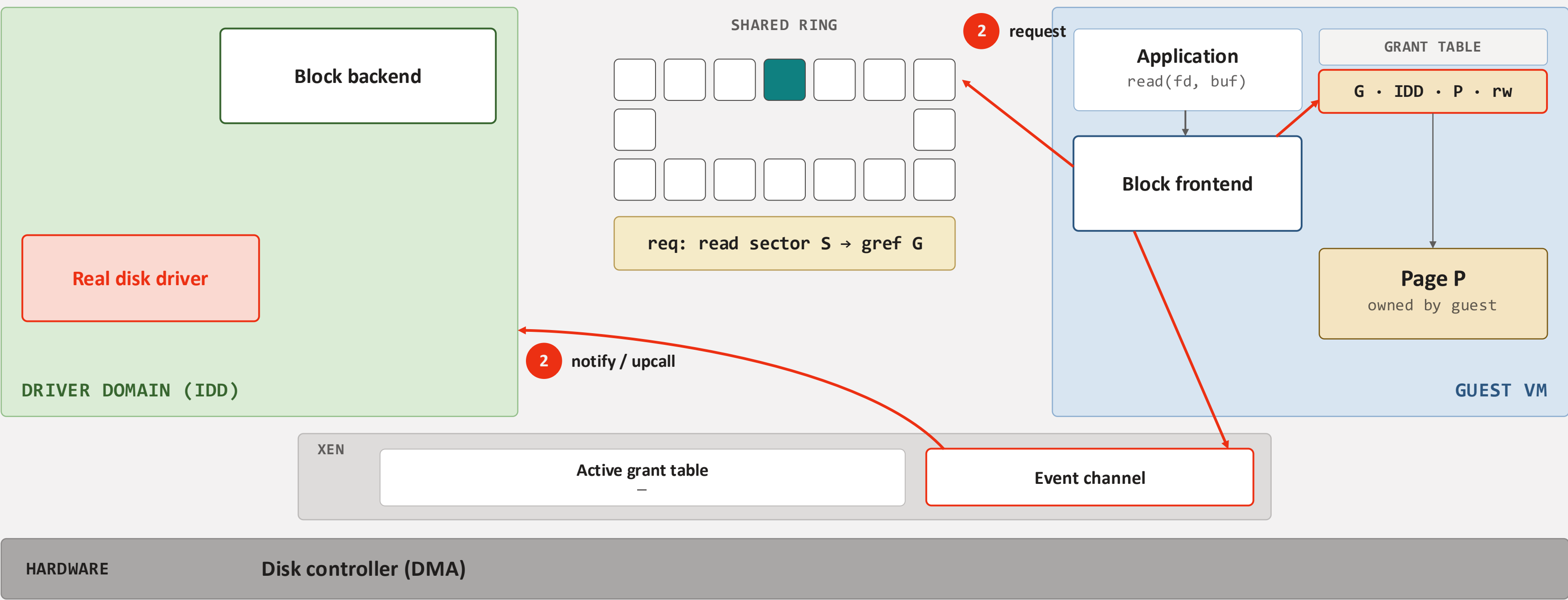
With an IOMMU, pinning would also allocate an IOMMU entry. **That is the only change needed for safe DMA,** and only Xen sees it; IDDs are unaware.

The disk read path, end to end



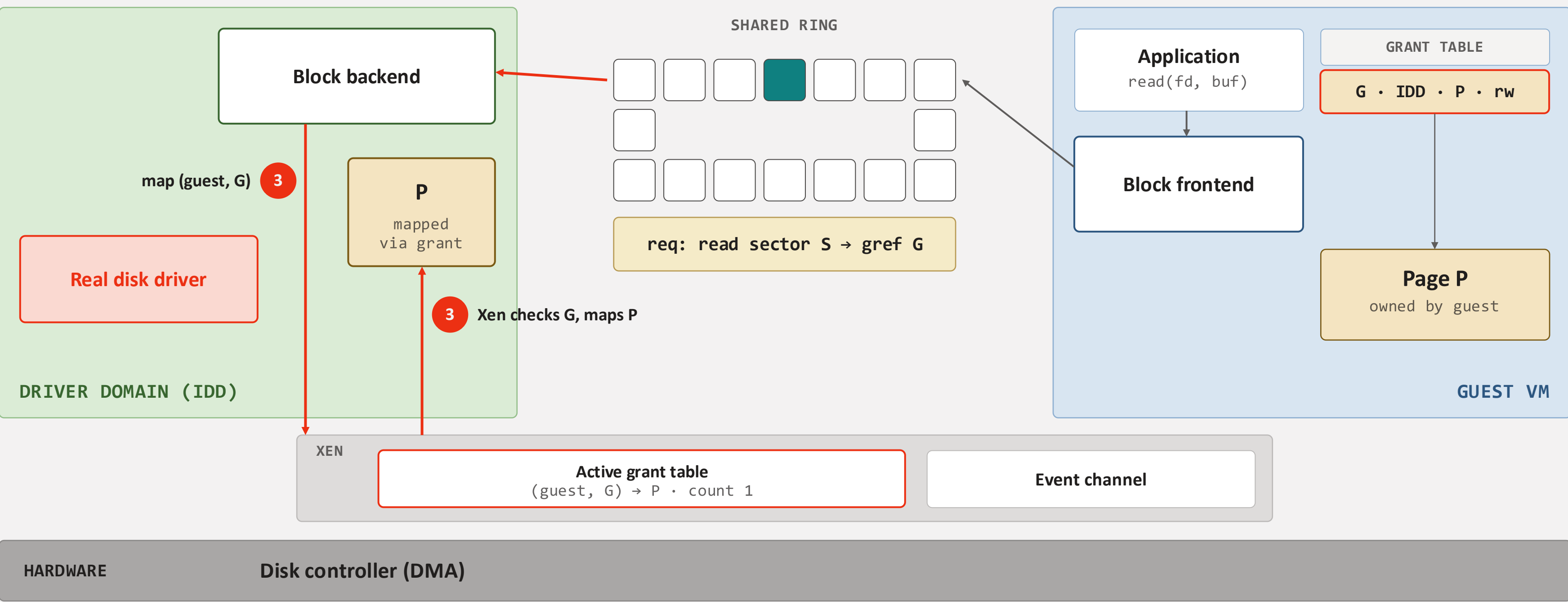
Step 1 of 5 · An application calls `read()`. The guest's block frontend picks a buffer page P in guest memory. P belongs to the guest and never changes owner.

The disk read path, end to end



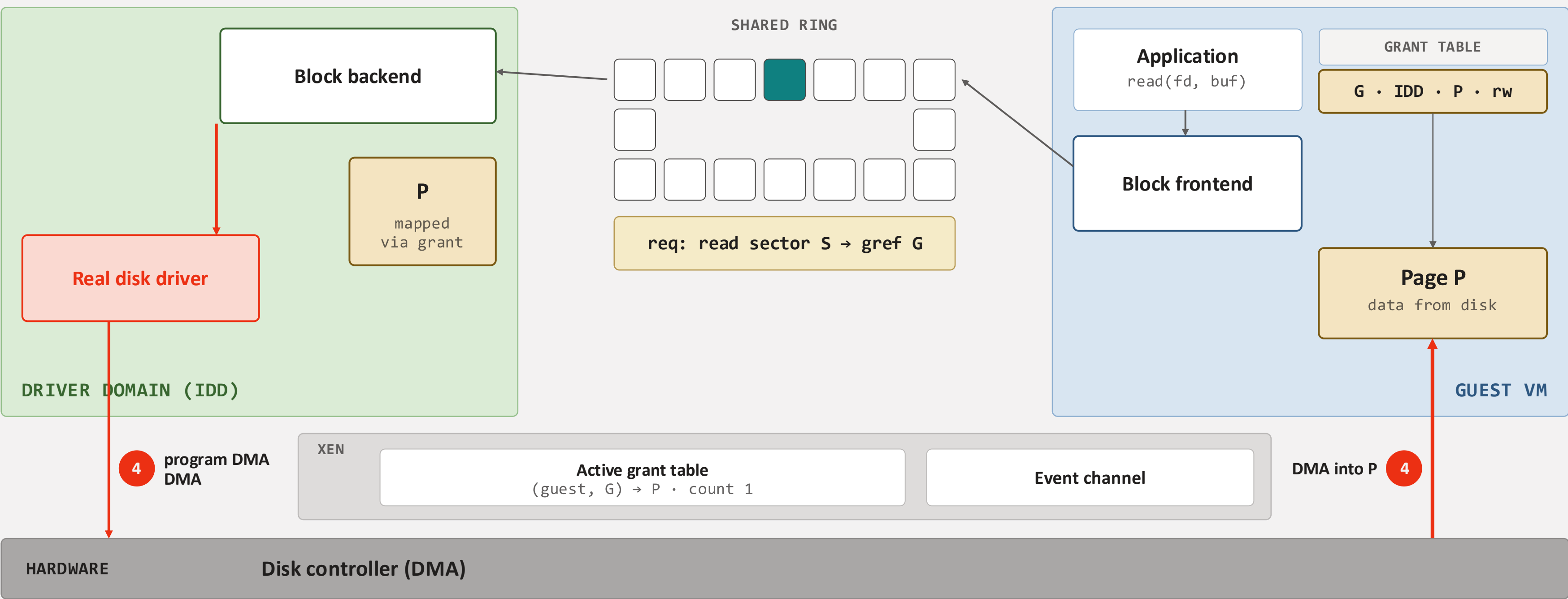
Step 2 of 5 · The frontend grants `P` to the driver domain (entry `G`, writable), puts a request with sector `S` and grant reference `G` on the shared ring, and notifies the event channel.

The disk read path, end to end



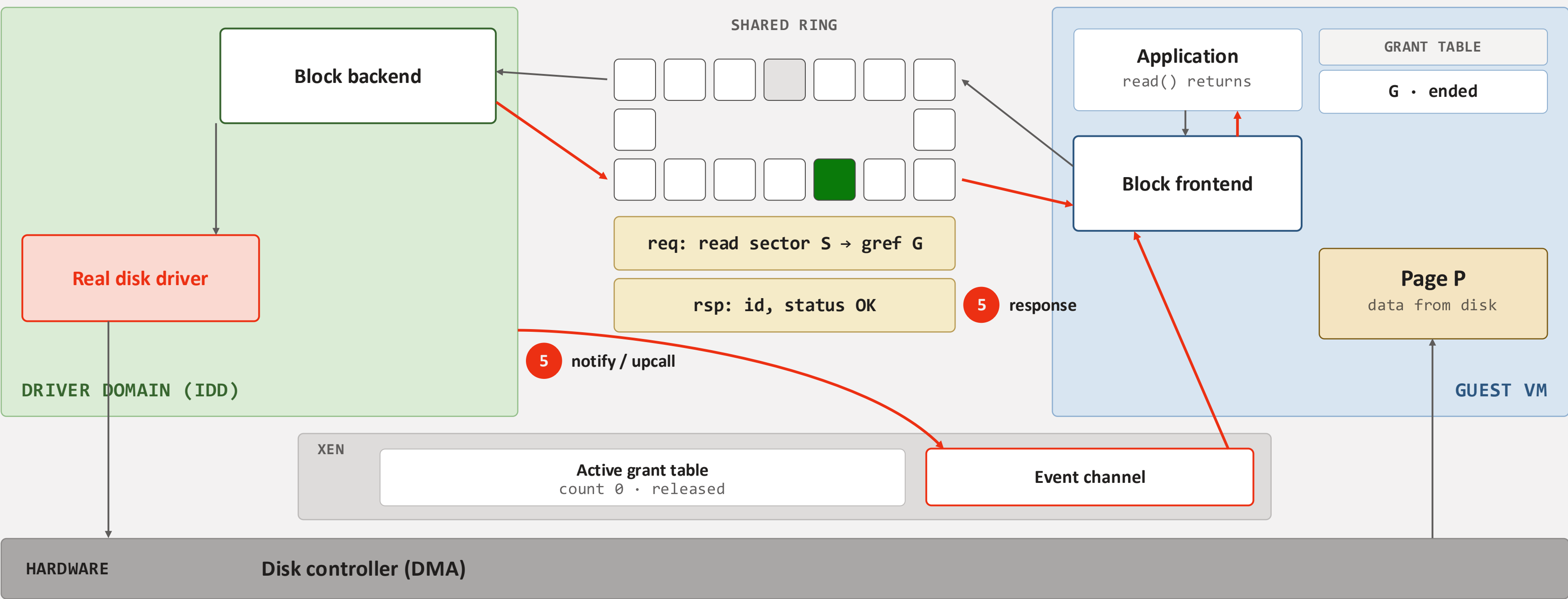
Step 3 of 5 · The backend takes the request and hands (guest, G) to Xen. Xen checks entry G, records it in its active grant table, and maps P into the driver domain. The paper calls this pinning.

The disk read path, end to end



Step 4 of 5 · The real disk driver programs the controller with P's bus address. The controller DMA's the data straight into P, with no copy through the driver domain.

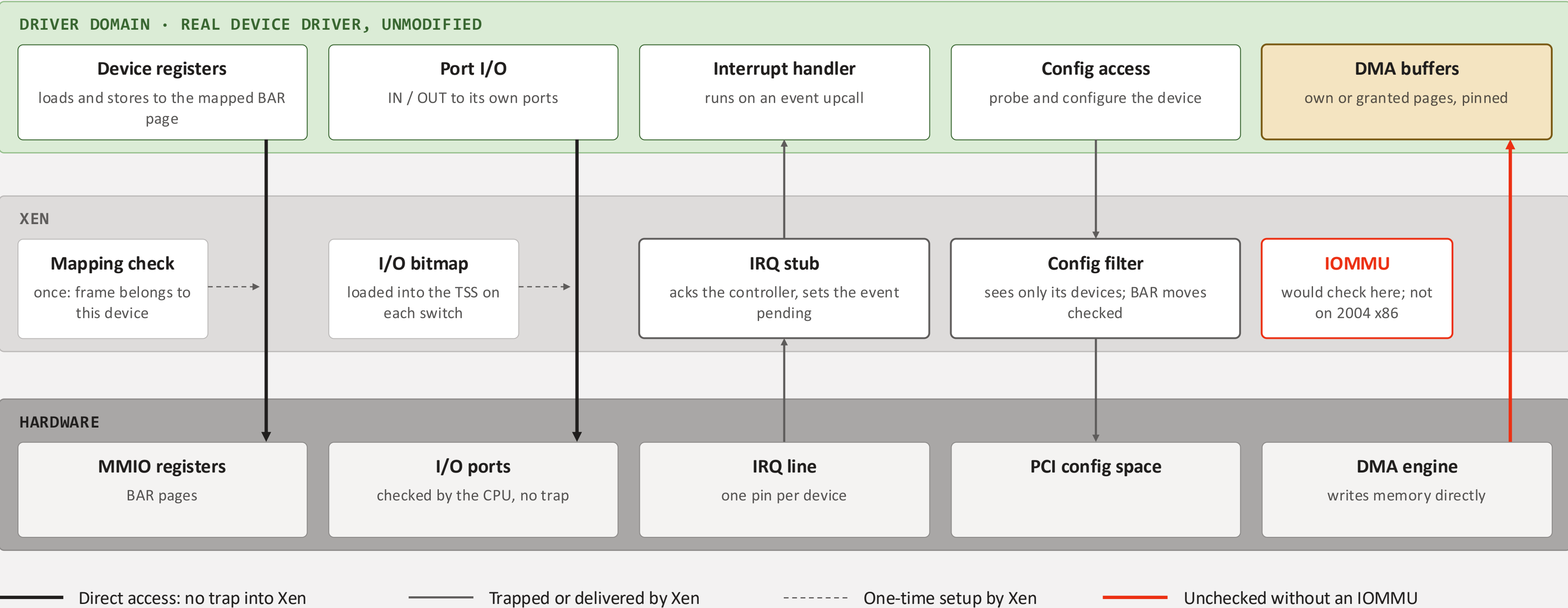
The disk read path, end to end



Step 5 of 5 · The backend unmaps P, posts a response, and notifies the event channel. The frontend's upcall reads the response, ends the grant, and `read()` returns.

The guest never gives a driver an address. It gives a capability, and Xen checks the capability every time.

Device passthrough: how the driver domain reaches its device



06

Evaluation

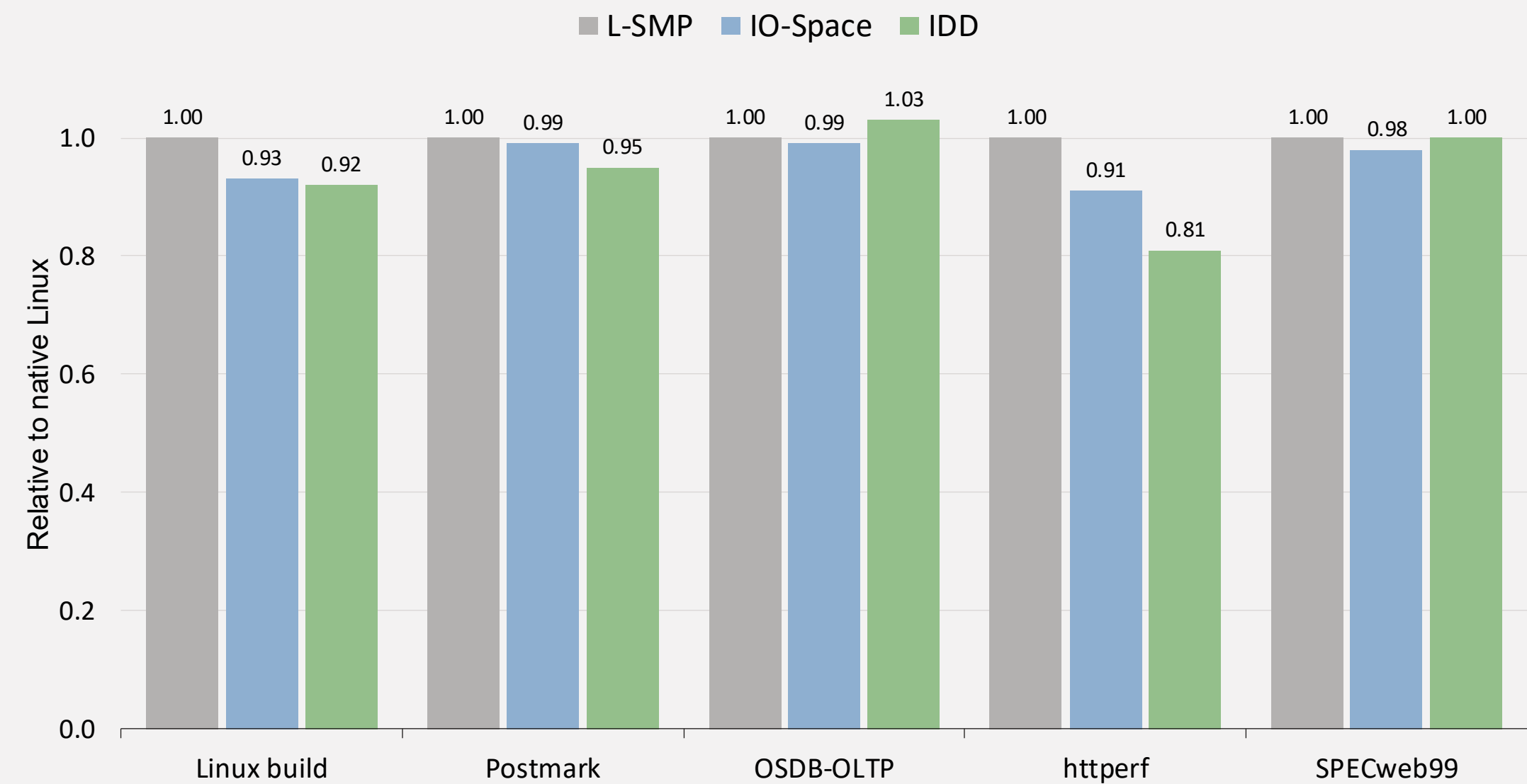
Experimental setup

Dell PowerEdge 2650: two 3.06 GHz Xeons, 1 GB RAM, two Broadcom Tigon 3 gigabit NICs, Adaptec Ultra160 SCSI with two 10K RPM disks. Every configuration uses the same Linux 2.4.26 driver source.

CONFIGURATION	WHERE DRIVERS RUN	WHAT IT MEASURES
L-SMP	In a vanilla Linux 2.4.26 SMP kernel	Native baseline
IO-Space	In Xen/Linux, reaching hardware through the protected interface	Cost of safe hardware access alone
IDD	In separate driver domains, reached over device channels	Cost of the full architecture

■ Each IDD and the Xen/Linux guest run in their own Xen domain on a separate physical CPU.

Application benchmarks



- L-SMP** native Linux 2.4.26 SMP, drivers in the kernel: the baseline
- IO-Space** Linux on Xen, driving its devices directly through an I/O Space
- IDD** drivers in a separate driver domain, reached over device channels

BUILD AND POSTMARK

I/O Spaces cost about 7% and 1%; IDDs about 9% and 5%.

SPECWEB99

A demanding mixed workload: overheads of just 1% and 2%.

HTTPPERF

One request in flight exposes interrupt and notification costs; IDD stays within 19% of native.

OSDB-OLTP

IDD beats native, explained by the disk-write effect in the dd results.

Network throughput (ttcp)

TCP throughput over gigabit Ethernet to a native host, 128 KB socket buffers. The small 552-byte MTU, common for dial-up PPP clients, stresses per-packet CPU overhead.

MB/S	MTU 1500 · TX	MTU 1500 · RX	MTU 552 · TX	MTU 552 · RX
L-SMP	897	897	808	808
IO-Space	897 (0%)	898 (0%)	718 (−11%)	769 (−5%)
IDD	897 (0%)	898 (0%)	778 (−3%)	663 (−18%)

- Virtualized interrupt dispatch costs 11% on transmit and 5% on receive under the heaviest packet rate: **safe interrupt control at reasonable cost.**
- L-SMP:** native Linux. **IO-Space:** Linux on Xen, driving devices directly through an I/O Space. **IDD:** drivers in a separate driver domain, reached over device channels.

Disk throughput (dd)

Repeatedly write, then read, a 4 GB file on ext3. Disk I/O moves data in large units with little per-operation overhead, so it strains the CPU less than networking.

MB/S	READ	WRITE
L-SMP	66.01	47.36
IO-Space	65.78 (−0%)	46.74 (−1%)
IDD	65.16 (−1%)	58.47 (+23%)

Reads are nearly identical in every configuration. Write measurements suffer from an oscillatory behaviour of the Linux 2.4 memory system under bulk writes.

- The IDD's **+23%** write result is not free speed: the device channel adds a stage of queueing that happens to stabilize throughput.
- L-SMP:** native Linux. **IO-Space:** Linux on Xen, driving devices directly through an I/O Space. **IDD:** drivers in a separate driver domain, reached over device channels.

Driver recovery

275

MILLISECONDS OF NETWORK OUTAGE PER RESTART

The network driver was made to perform an illegal memory access, so failure detection is immediate; a deadlock or infinite loop would add a timeout-dependent delay.

An external machine sent evenly spaced pings at 200 packets per second while a fault was injected every 10 seconds. The guest saw outages of around 275 ms each time.

WHERE THE TIME GOES

Most of the outage is the driver's own media-detection routine determining link status, not the isolation machinery or the controller's restart work.

07

Takeaways

What the paper shows

Placing drivers in isolated virtual machines lets existing driver code serve environments where dependability matters, while one driver instance is shared by many OSes.

-
- 1 Isolation without new hardware.** Nearly every requirement is met on 2004 x86; the notable exception is protection against erroneous DMA.
 - 2 Low overhead.** Application-level overhead is generally less than a few percent.
 - 3 Fast restart.** A failed driver is replaced within a few hundred milliseconds.
 - 4 Ready for hardware.** The design absorbs an IOMMU with a change confined to Xen's pin path.
-

Reading it today

HARDWARE THAT ARRIVED

The chipset IOMMU the paper anticipated shipped as Intel VT-d and AMD-Vi, closing the DMA gap. Message-signalled interrupts removed the shared-line problem, as the paper predicted for PCI Express.

IDEAS THAT STAYED

Grant references, shared descriptor rings, and event channels remain the basis of Xen's split front-end/back-end paravirtual drivers, and driver domains remain a Xen deployment option.

| The paper's software design defined the interface that later hardware only had to accelerate.

Questions for discussion

- 1 An IDD handles the guest's data in granted pages. What can a malicious or buggy IDD still do to a guest's data, even with an IOMMU?
 - 2 httpperf lost 19% in the IDD configuration. Which inter-domain notifications dominate a single-request workload, and how would batching help?
 - 3 Restart assumes reissued requests are idempotent. Which devices or requests break that assumption?
 - 4 Is a narrow, idealized interface realistic when devices keep adding offloads that interface unioning would have exposed?
-

Give each driver a virtual machine and only its own device,
and let the chipset close the DMA gap when it can.

SOURCE AND SCOPE

Safe Hardware Access with the Xen Virtual Machine Monitor

K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson. OASIS Workshop, 2004. University of Cambridge Computer Laboratory and Intel Research Cambridge.

RELATED READING CITED BY THE PAPER

- 1 Barham et al., **Xen and the Art of Virtualization**, SOSP 2003: the paravirtualized Xen design this work builds on.
- 2 Swift, Bershad, Levy, **Improving the Reliability of Commodity Operating Systems**, SOSP 2003: in-kernel driver isolation.
- 3 LeVasseur, Uhlig, Stoess, Götz, **Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines**, OSDI 2004.