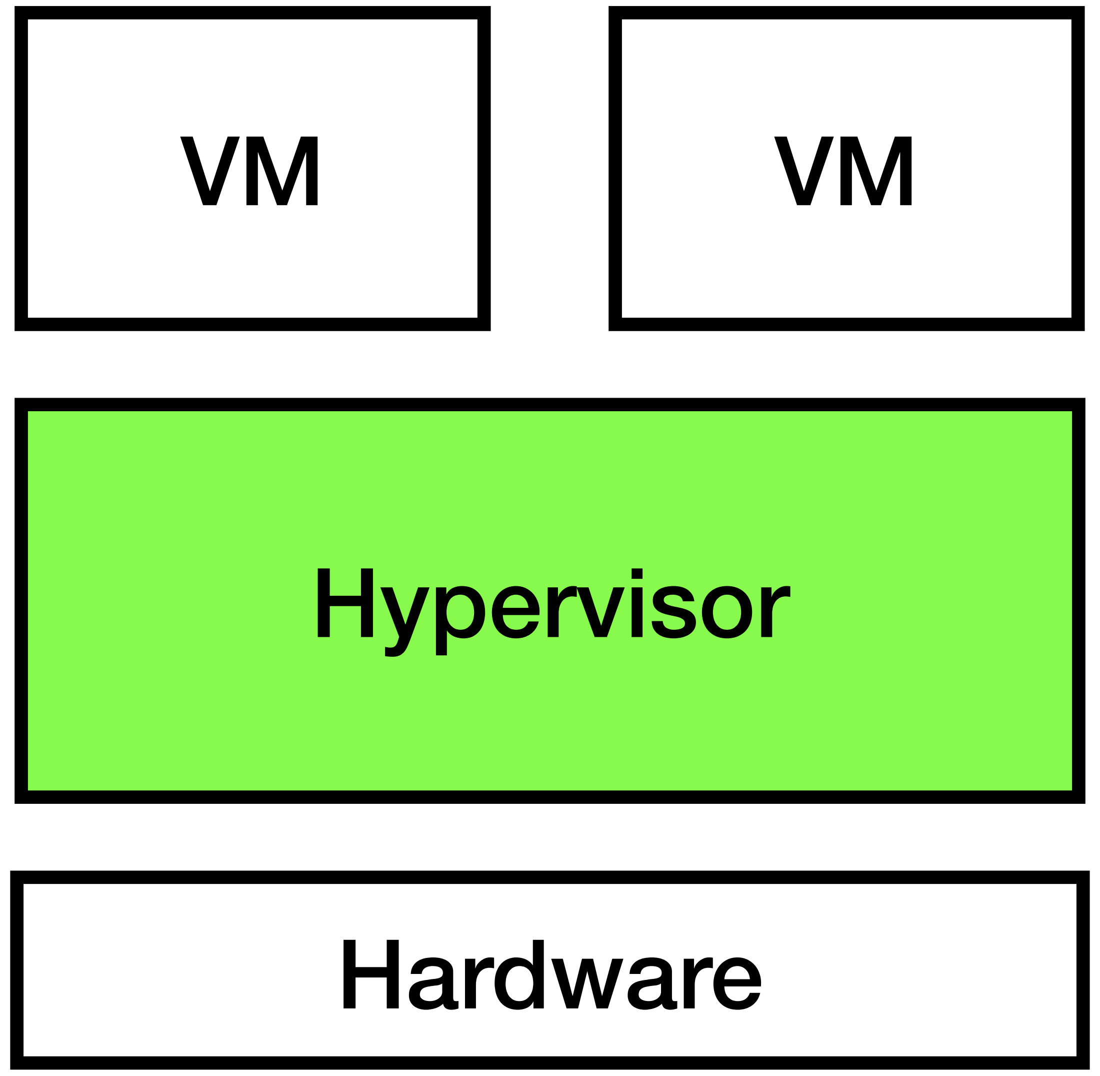
Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, John Zhuang Hui

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Virtualization

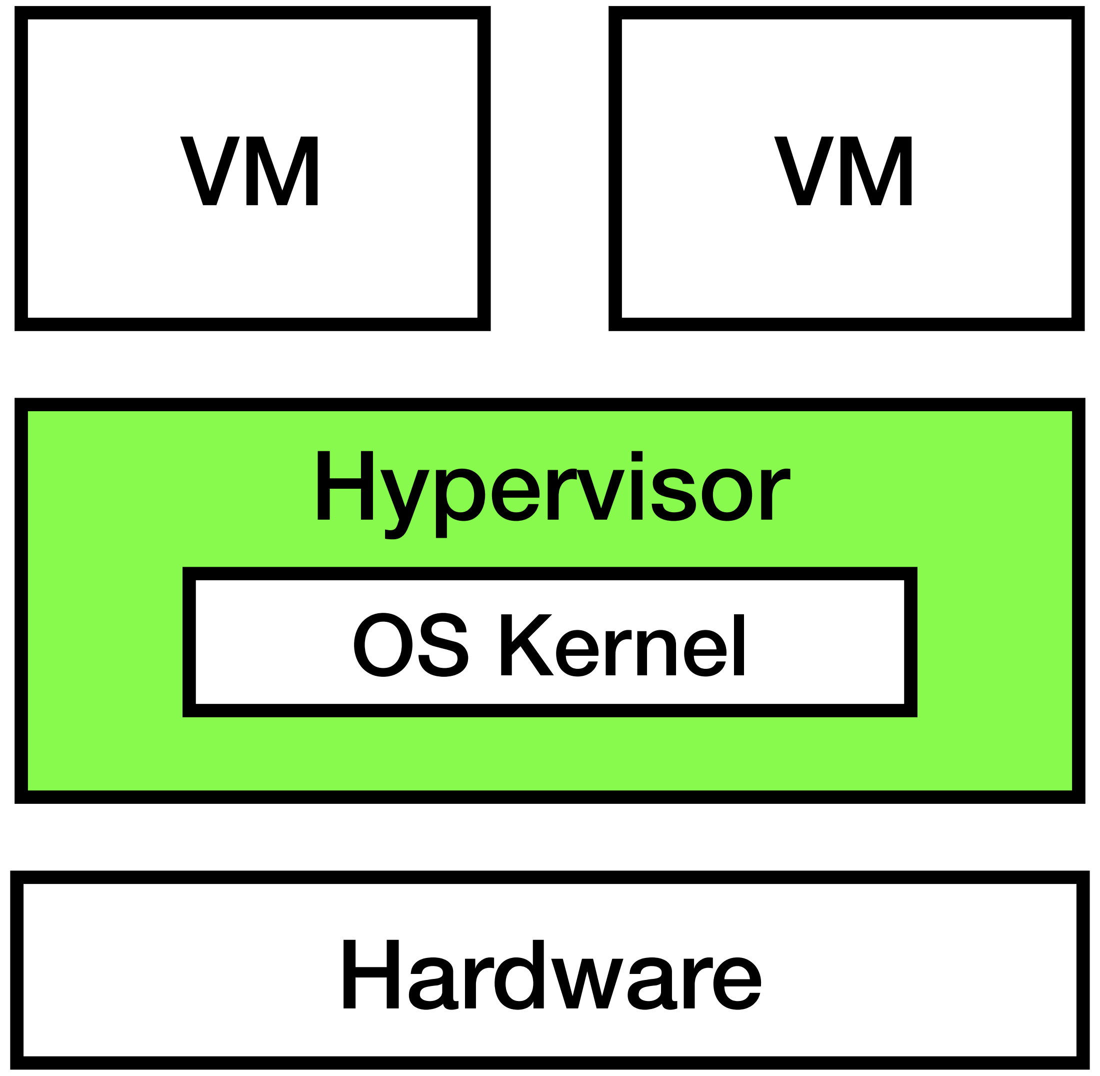


aws

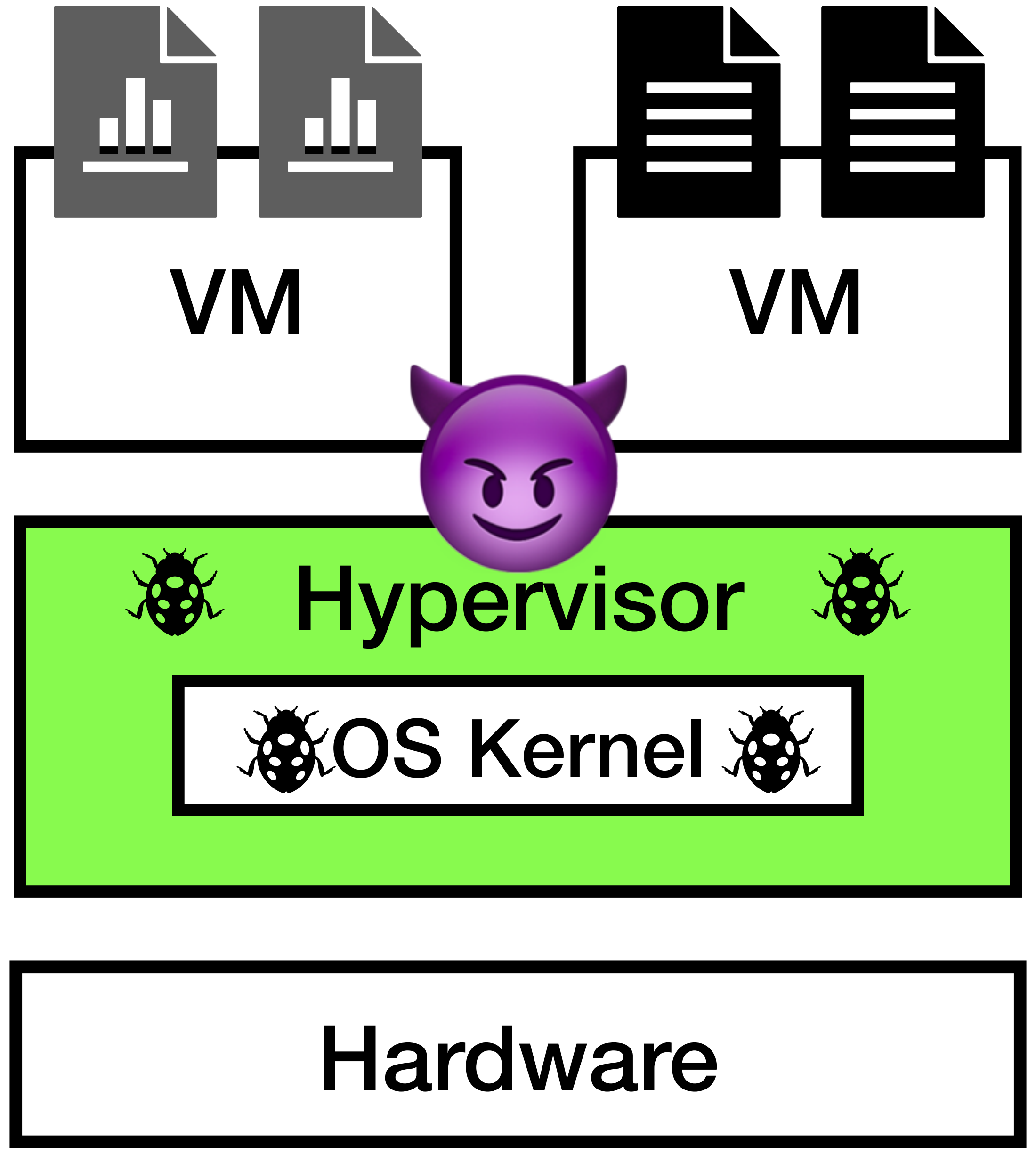
Microsoft Azure

Google Cloud

Growing Hypervisor Complexity



Security Risks of Hypervisors



Formal Verification (1)

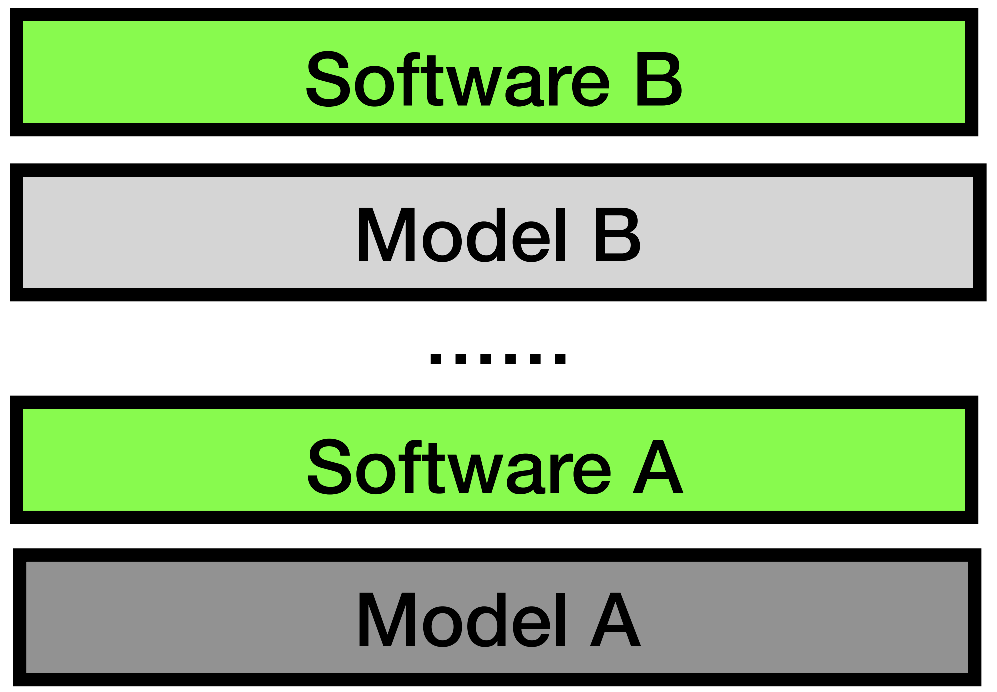
- Verify functional correctness of the program
  - Includes three components: implementation, specification, hardware model
- Prove the implementation running on the hardware model satisfies the specification
  - Soundness of the proofs relies on the accuracy of the hardware model

Formal Verification (2)

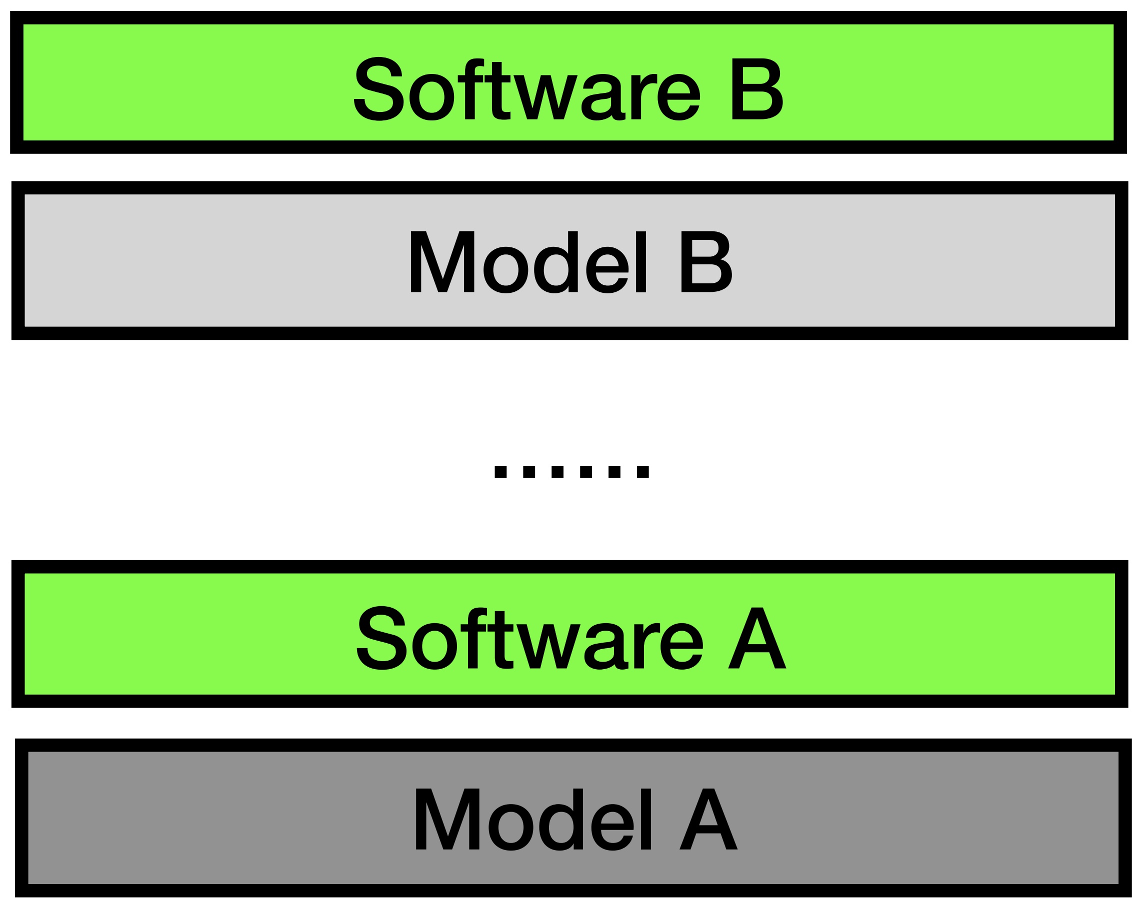
- Previous systems [seL4, CertiKOS] were verified using simplistic models
  - Proofs may not hold on real multiprocessor server hardware
- Previous work proposes hardware models [Promising Arm] that account for detailed hardware behaviors
  - Have not been shown to be feasible to verify real software

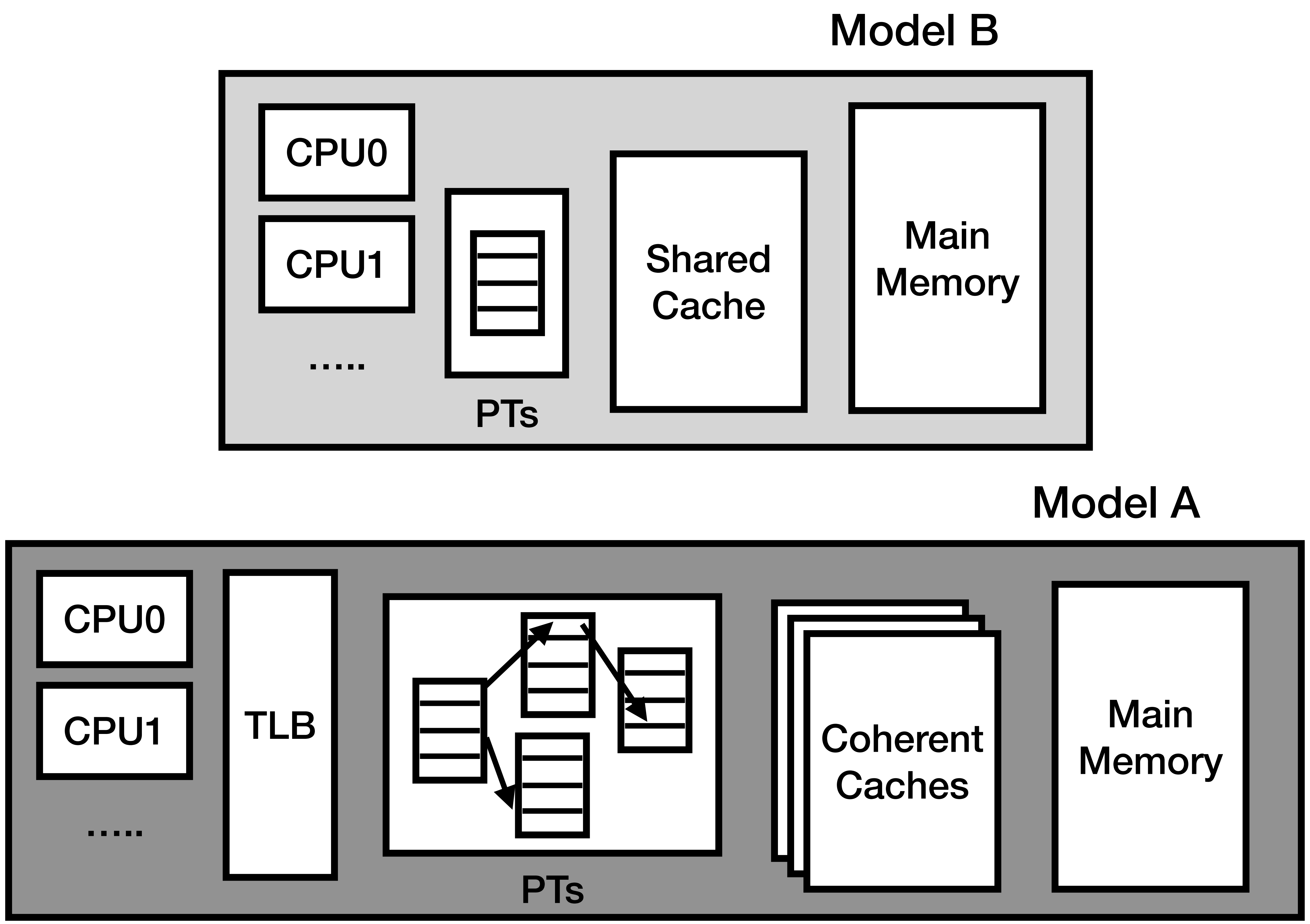
Layered Hardware Model (1)

- Capture realistic multiprocessor hardware features
- Ensure the model is simple enough to use for verifying commodity software
  - Tailor the complexity of the hardware model for the software needs



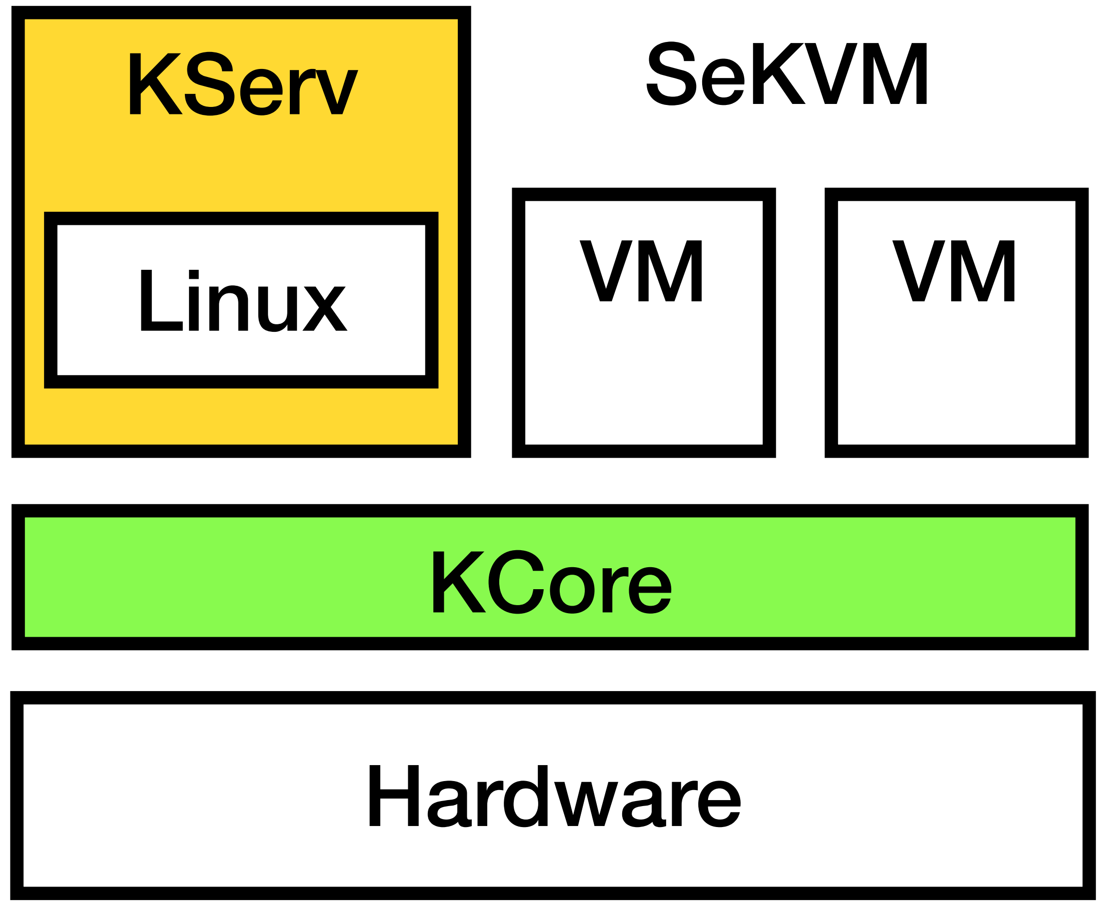
Layered Hardware Model (2)

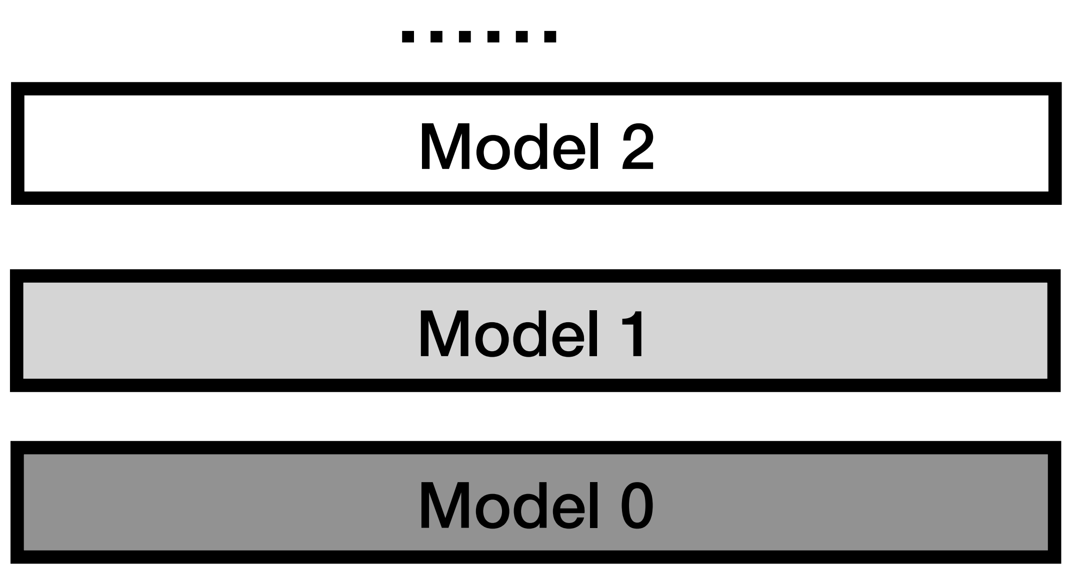




Verifying a Commodity Hypervisor

- Build on SeKVM [S&P 21], a verified multiprocessor KVM hypervisor
  - Use the layered hardware model to verify SeKVM
  - Ensure SeKVM's proofs hold on multiprocessor server hardware

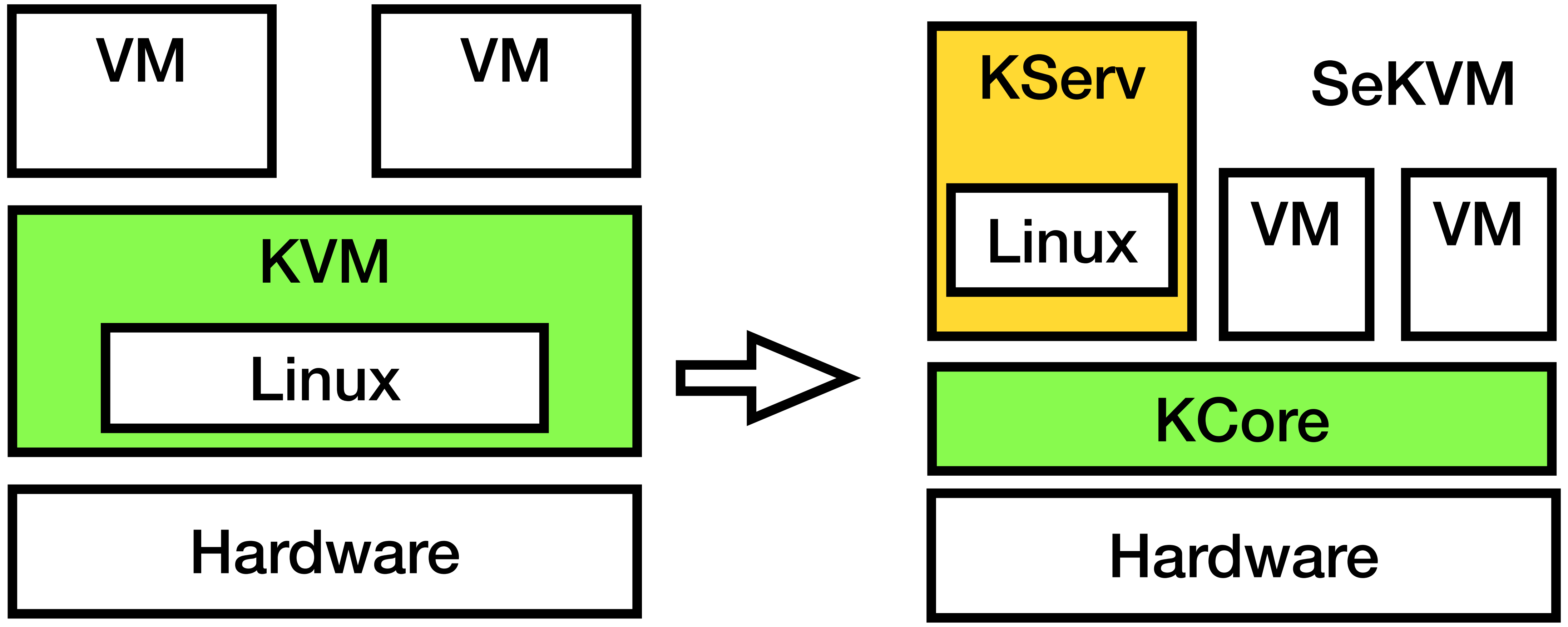




SeKVM

A Verified Commodity Multiprocessor KVM Hypervisor

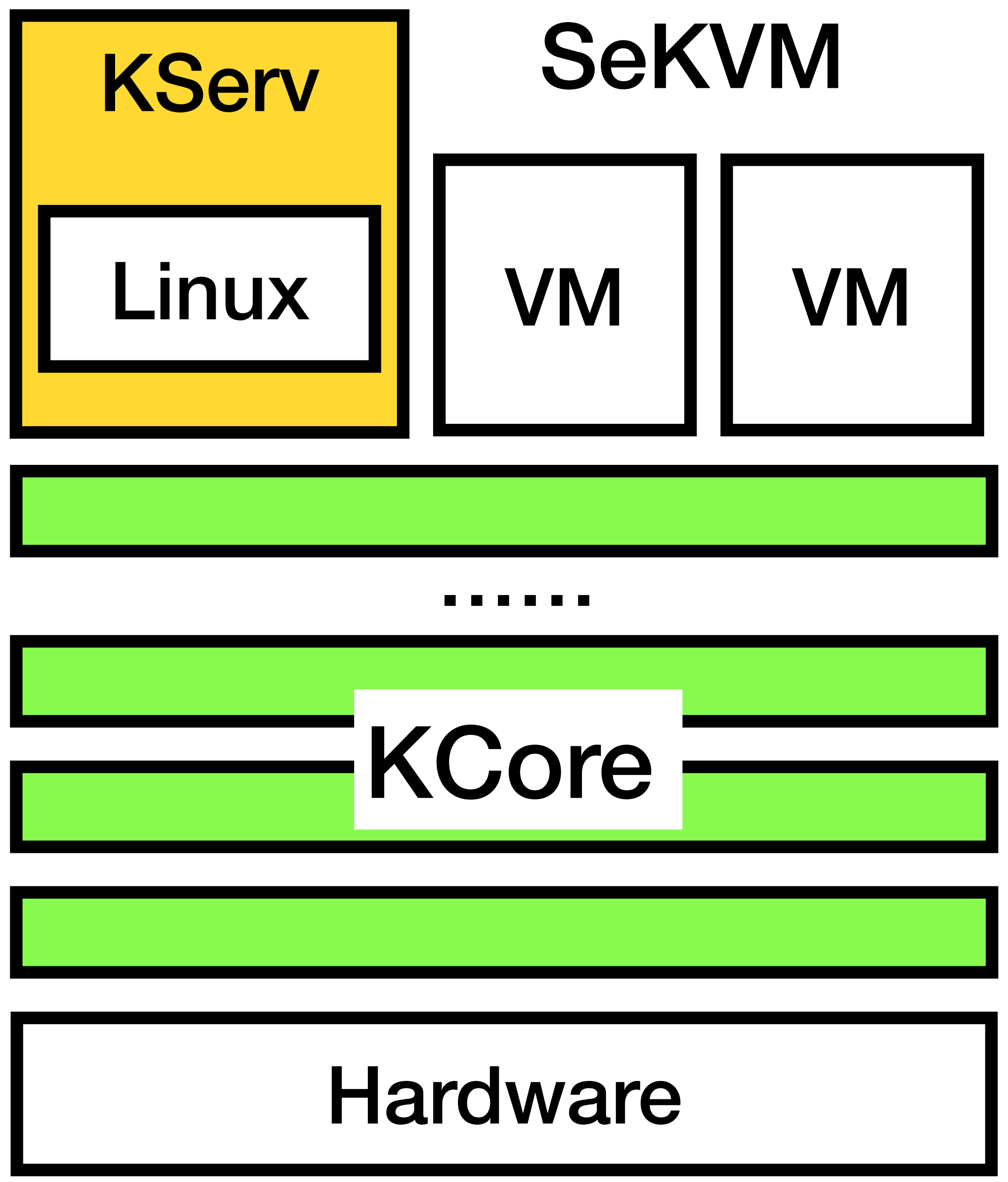
- SeKVM leverages Arm Virtualization Extensions and retrofits KVM into:
  - A *KCore* that protects VM confidentiality and integrity, serves as KVM's TCB
  - An untrusted *KServ* that provides virtualization functionality

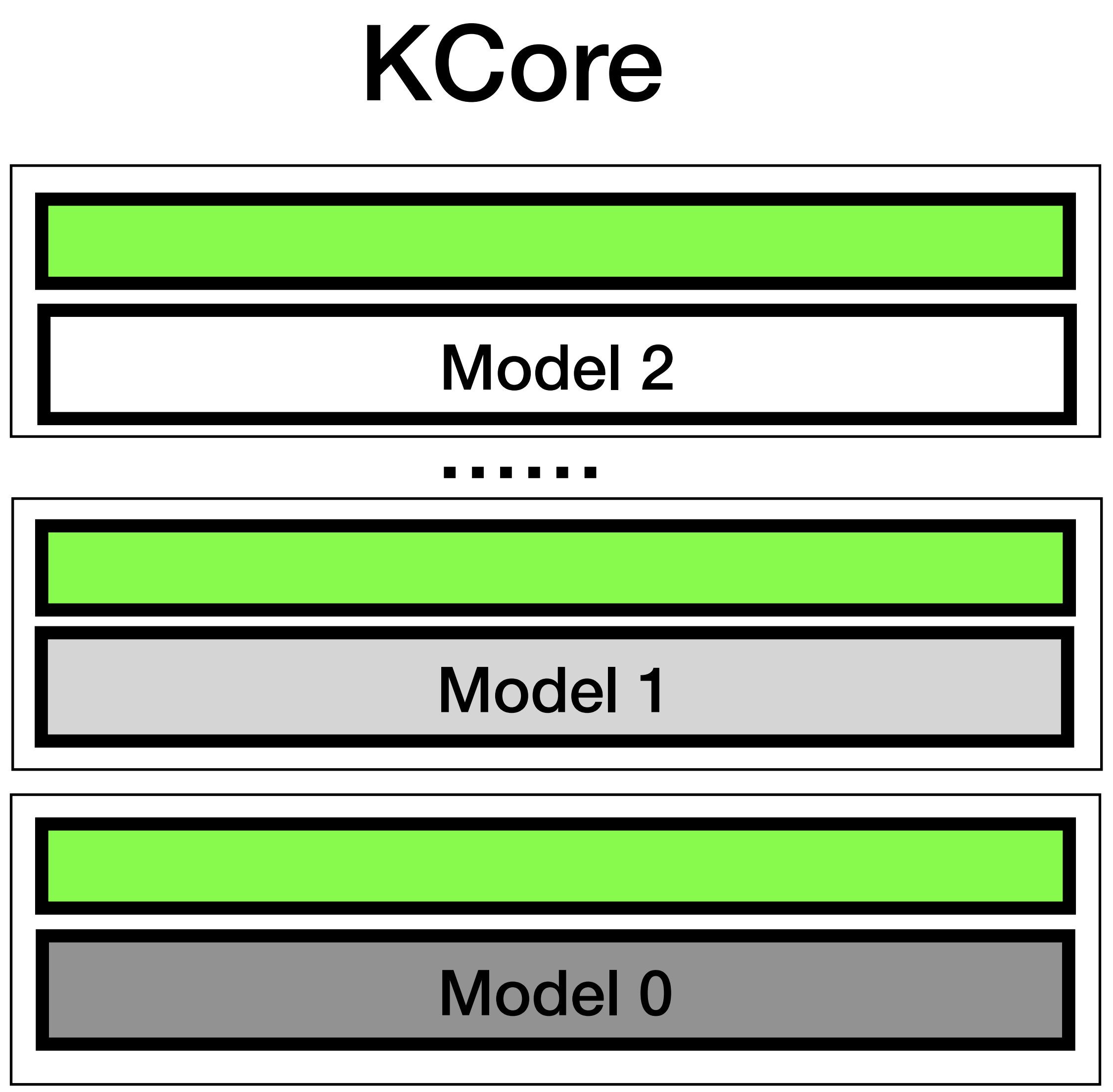


Virtualization Extensions (VE)

Verifying KCore

- Structure KCore as a stack of layered modules to match the layered model



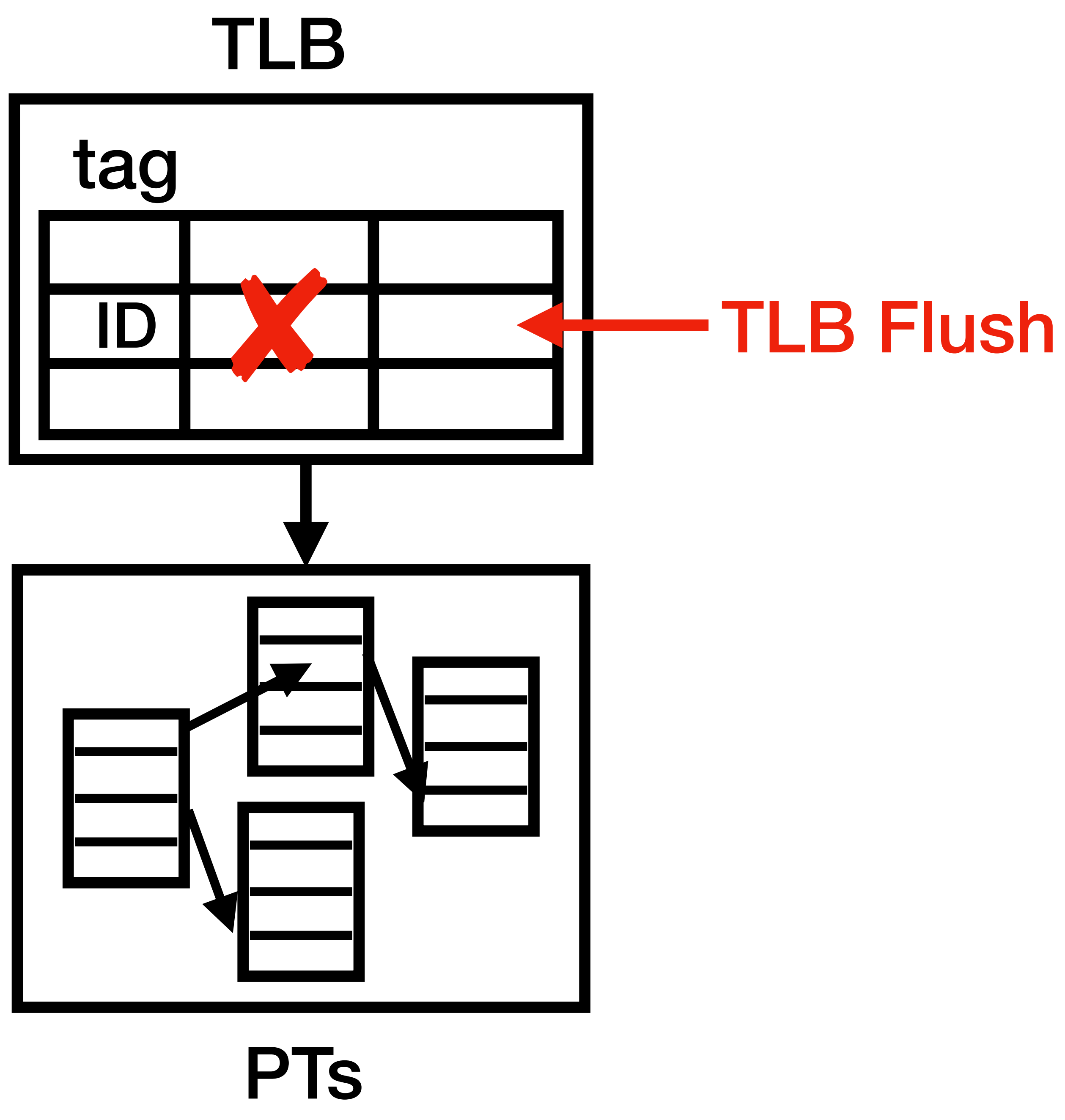


Case Study: Verify KCore's TLB Management (1)

- TLB caches page table translations
- Arm provides tagged TLB to avoid flushes when switching CPU execution
  - Software flushes TLB when updating page tables

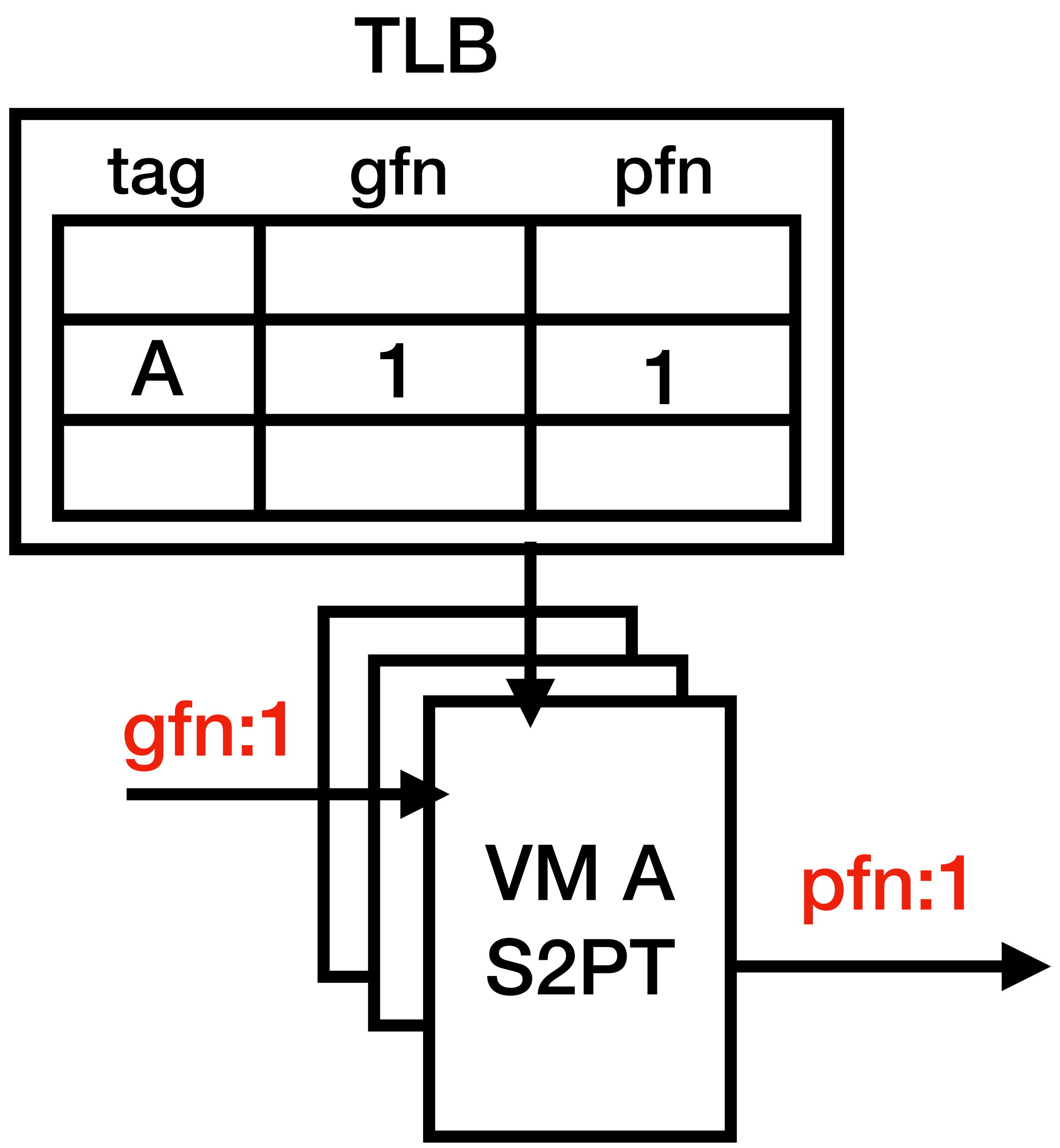
ARM

Virtualization Extensions (VE)



Case Study: Verify KCore's TLB Management (2)

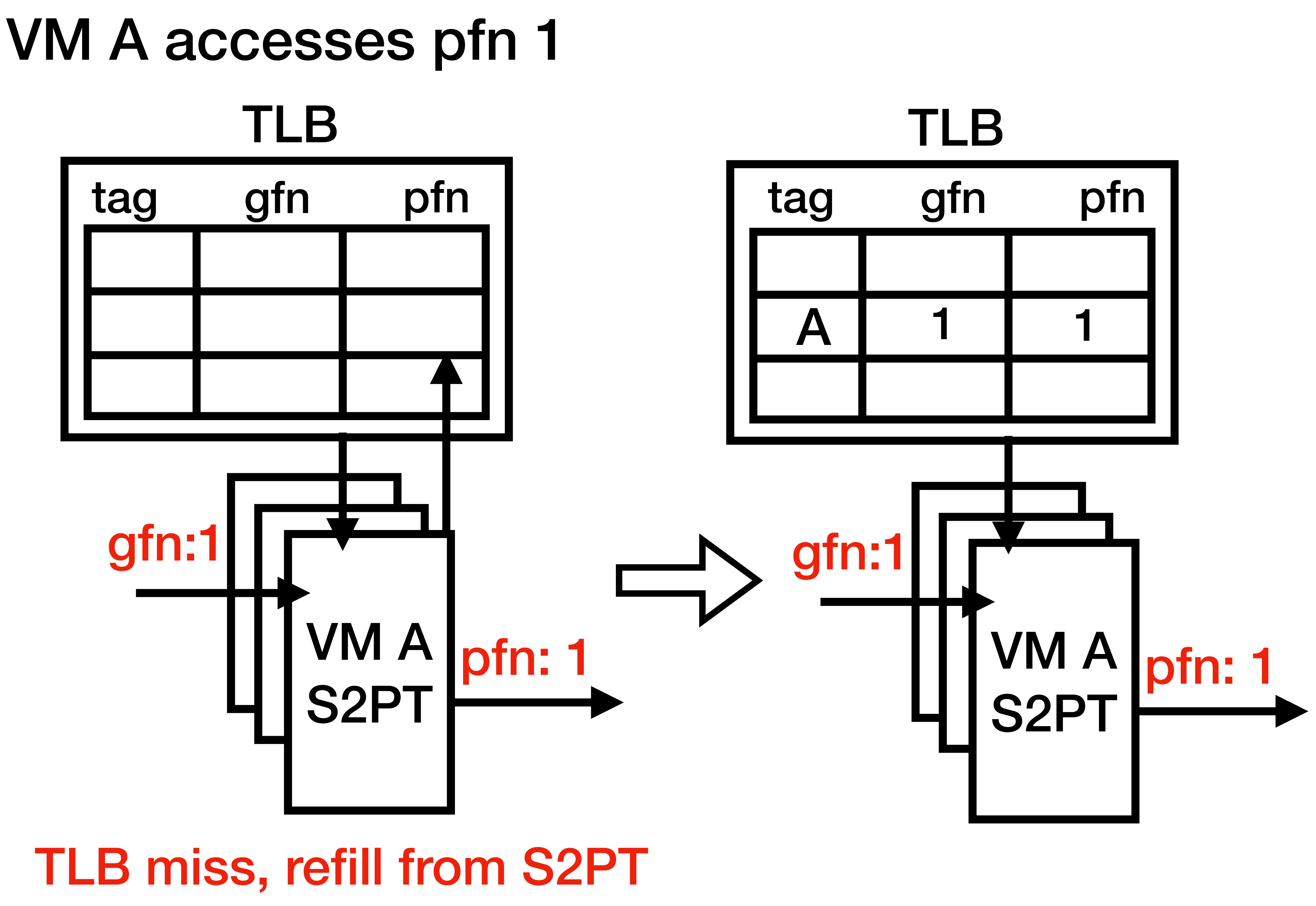
- Consider the TLB caches entries from Arm's stage 2 page tables (S2PT) — translate a guest physical page (gfn) to a physical page (pfn)

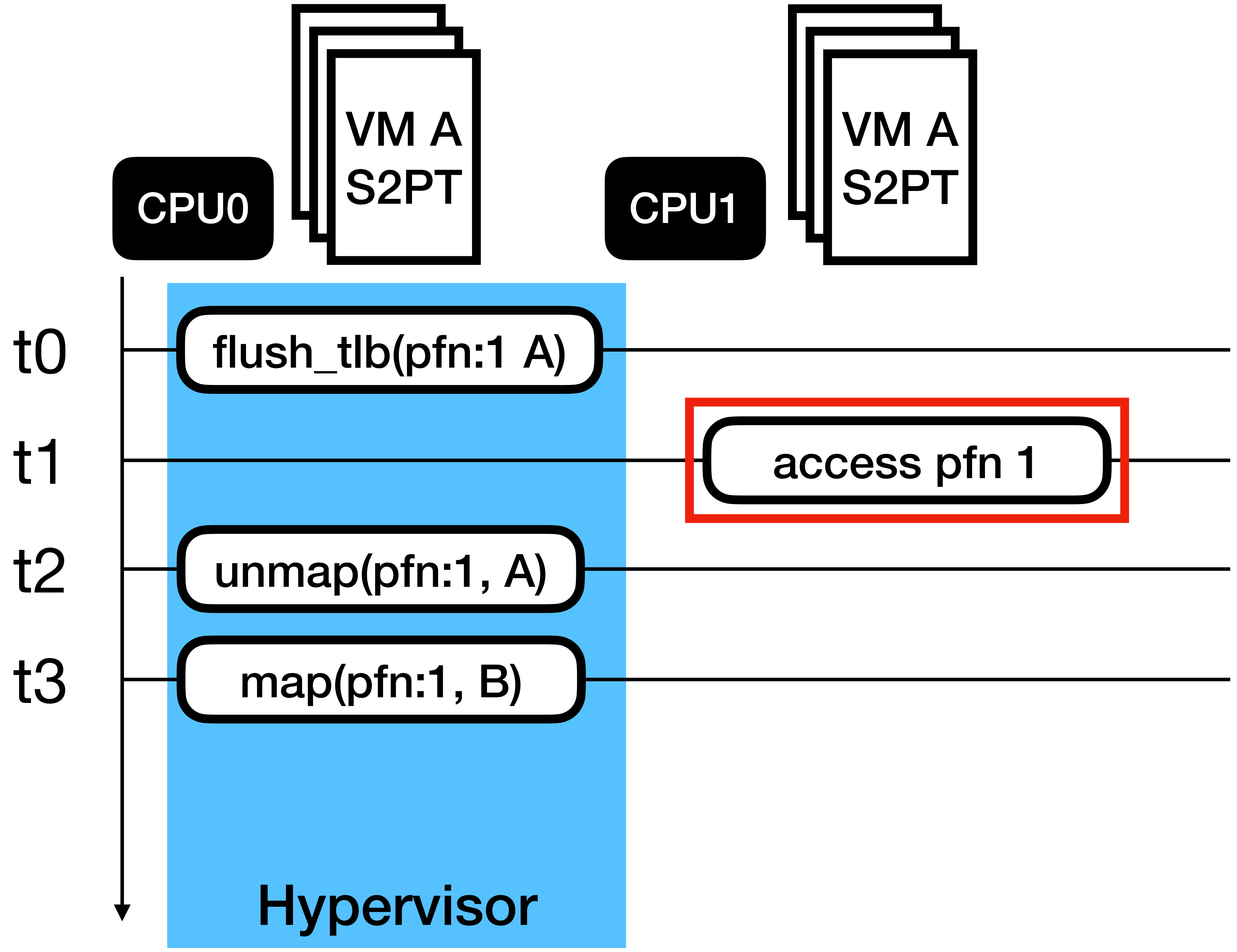


1. flush_tlb(pfn:1, A)
2. unmap(pfn:1, A)
3. map(pfn:1, B)

Case Study: Verify KCore's TLB Management (3)

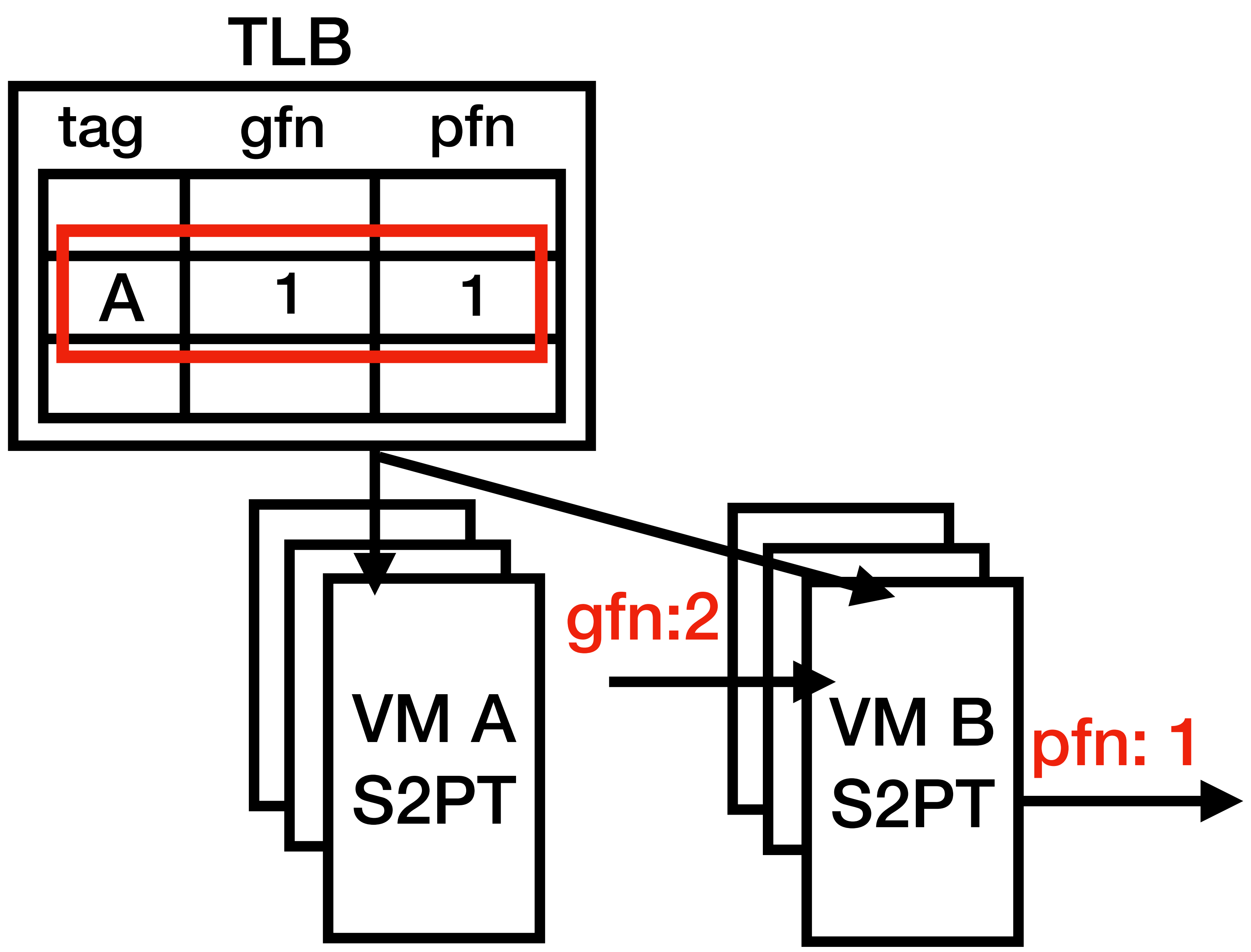
- Multiprocessor VM A that accesses pfn 1 results in caching of pfn 1's mapping in the TLB

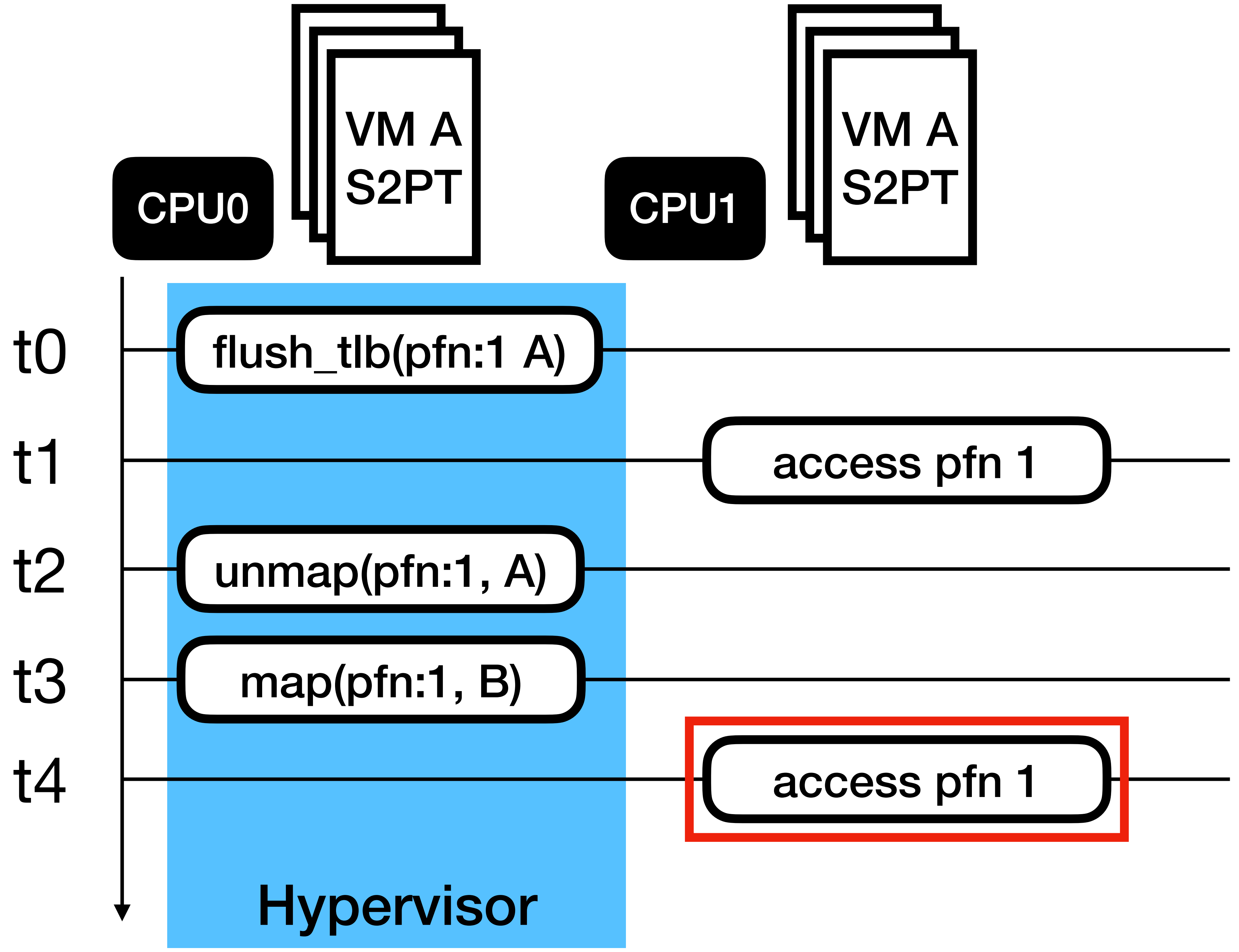




Case Study: Verify KCore's TLB Management (4)

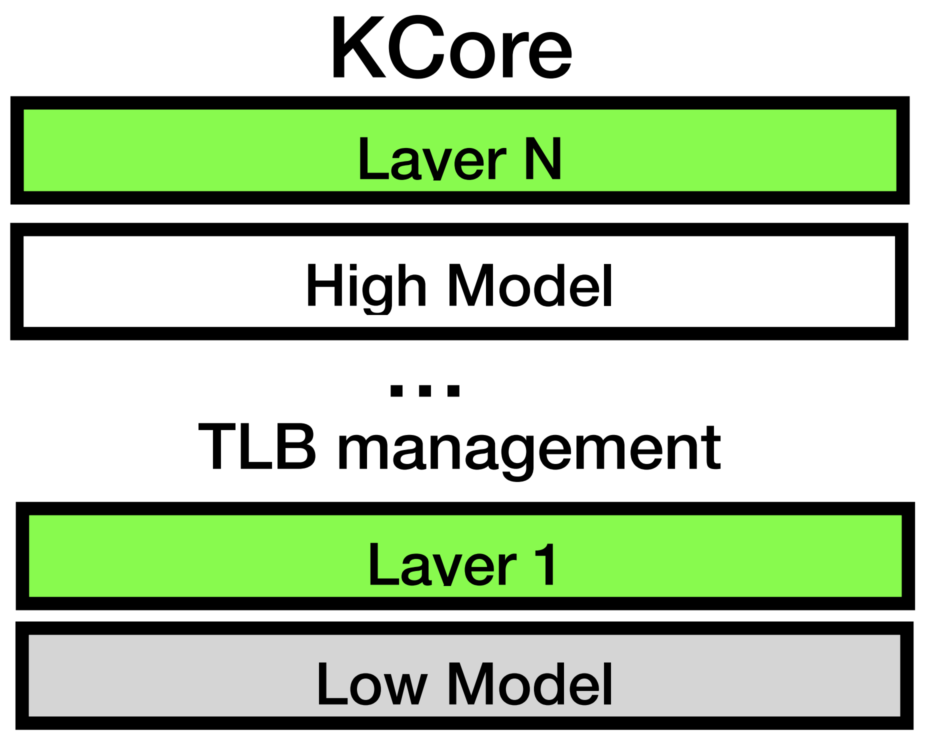
- VM A can access pfn 1 through the TLB and breaks VM isolation

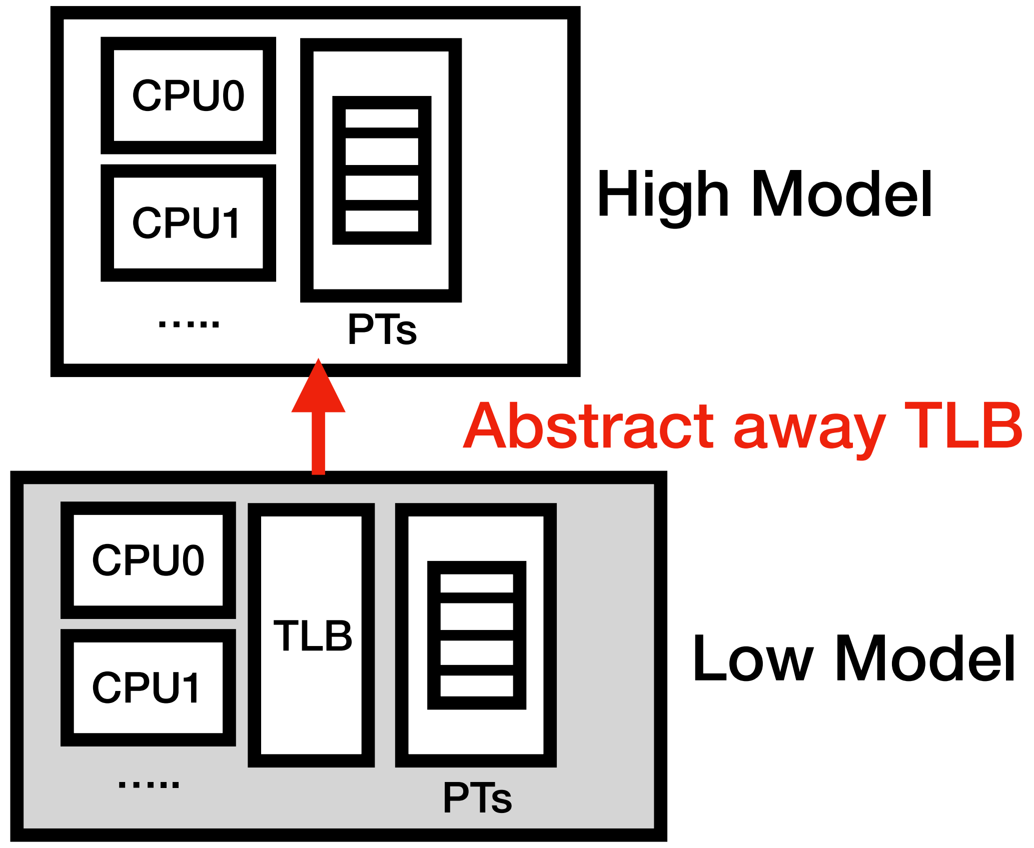




Case Study: Verify KCore's TLB Management (5)

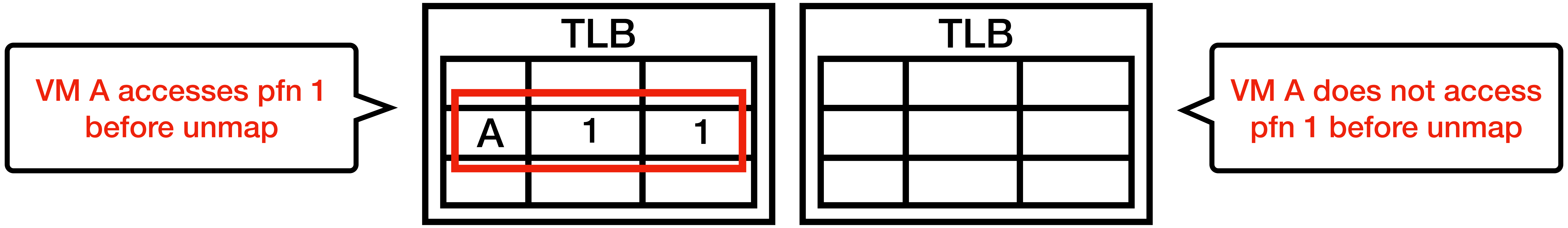
- Verify KCore's code that manages TLBs using a hardware model with tagged TLB behaviors
- Refine the complex model with TLBs and page tables into the simpler model with only page tables
    - Verify KCore's code that does not manage TLBs using a simpler model





Case Study: Verify KCore's TLB Management (6)

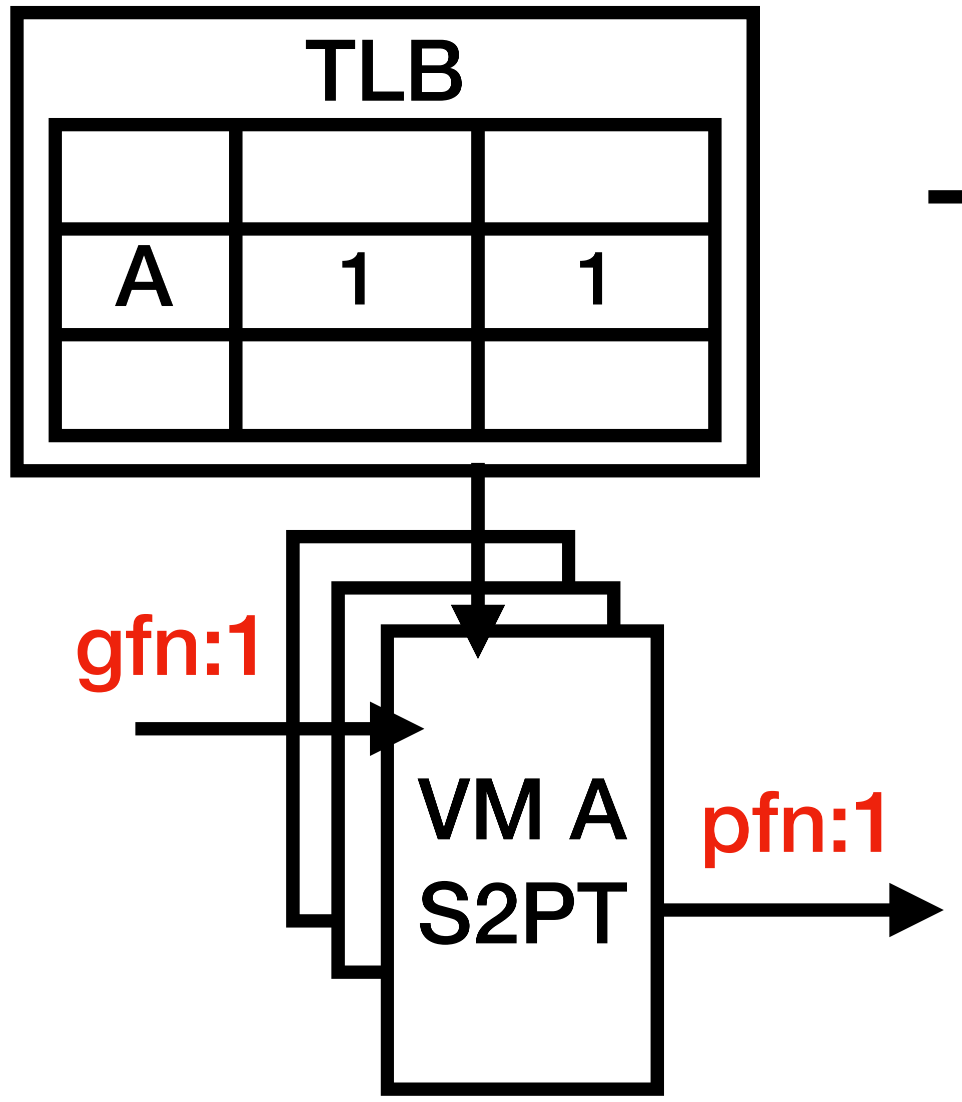
- Intuition: Pages observable through the incorrectly managed TLB will a ***superset*** of the ones through page tables
  - The TLB may include stale entries if not flushed after page table updates



1. flush_tlb(pfn:1, A)
2. unmap(pfn:1, A)
3. map(pfn:1, B)

Case Study: Verify KCore's TLB Management (7)

- Introduce ***page observers*** — the set of principals (VMs or KServ) who can observe a physical page (pfn) through TLBs or page tables
- Merge consecutive page observers into ***page observer groups***



Page observers TLB	Page observers PT
{1: A}	{1: A}

Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}

Case Study: Verify KCore's TLB Management (8)

- Consider the following execution steps

1. $\boxed{\text{unmap(pfn:1, A)}}$
2. flush_tlb(pfn:1, A)
3. map(pfn:1, B)

Page observers TLB	Page observers PT	Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}	{1: A}	{1: A}
{1: A}	{1: __}	{1: A}	{1: A}, {1: __}

Case Study: Verify KCore’s TLB Management (9)

- Consider the following execution steps

1. unmap(pfn:1, A)
2. flush_tlb(pfn:1, A)
3. map(pfn:1, B)

Page observers TLB	Page observers PT	Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}	{1: A}	{1: A}
{1: A}	{1: __}	{1: A}	{1: A}, {1: __}
{1: __}	{1: __}	{1: A}, {1: __}	{1: A}, {1: __}

Case Study: Verify KCore's TLB Management (10)

- Consider the following execution steps

1. unmap(pfn:1, A)
2. flush_tlb(pfn:1, A)
3. map(pfn:1, B)

Page observers TLB	Page observers PT
{1: A}	{1: A}
{1: A}	{1: __}
{1: __}	{1: __}
{1: B}	{1: B}

Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}
{1: A}	{1: A}, {1: __}
{1: A}, {1: __}	{1: A}, {1: __}
{1: A}, {1: __}, {1: B}	{1: A}, {1: __}, {1: B}

Case Study: Verify KCore's TLB Management (11)

- Prove KCore correctly manages the TLBs by showing that TLBs and page tables produce the same sequence of page observer groups

1. unmap(pfn:1, A)
2. flush_tlb(pfn:1, A)
3. map(pfn:1, B)

Page observers TLB	Page observers PT
{1: A}	{1: A}
{1: A}	{1: __}
{1: __}	{1: __}
{1: B}	{1: B}

Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}
{1: A}	{1: A}, {1: __}
{1: A}, {1: __}	{1: A}, {1: __}
{1: A}, {1: __}, {1: B}	{1: A}, {1: __}, {1: B}

Same

Case Study: Verify KCore's TLB Management (12)

- Use this approach to detect incorrect TLB management

1. flush_tlb(pfn:1, A)
2. unmap(pfn:1, A)
3. map(pfn:1, B)

Page observers TLB	Page observers PT
{1: A}	{1: A}
$\boxed{\text{\{1: A\}}}$	{1: A}
{1: A}	{1: __}
$\boxed{\text{\{1: A,B\}}}$	{1: B}

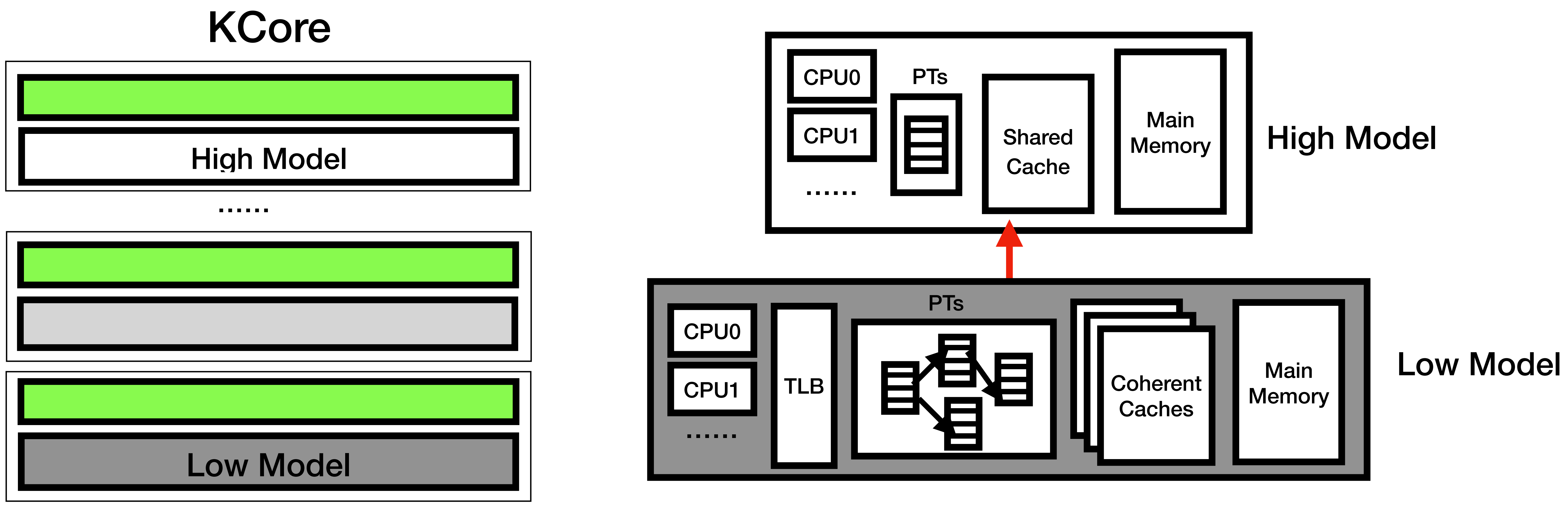
Can be refilled after TLB flush

Page observer groups TLB	Page observer groups PT
{1: A}	{1: A}
{1: A}	{1: A}
{1: A}	{1: A}, {1: __}
{1: A}, {1: A,B}	{1: A}, {1: __}, {1: B}

Different

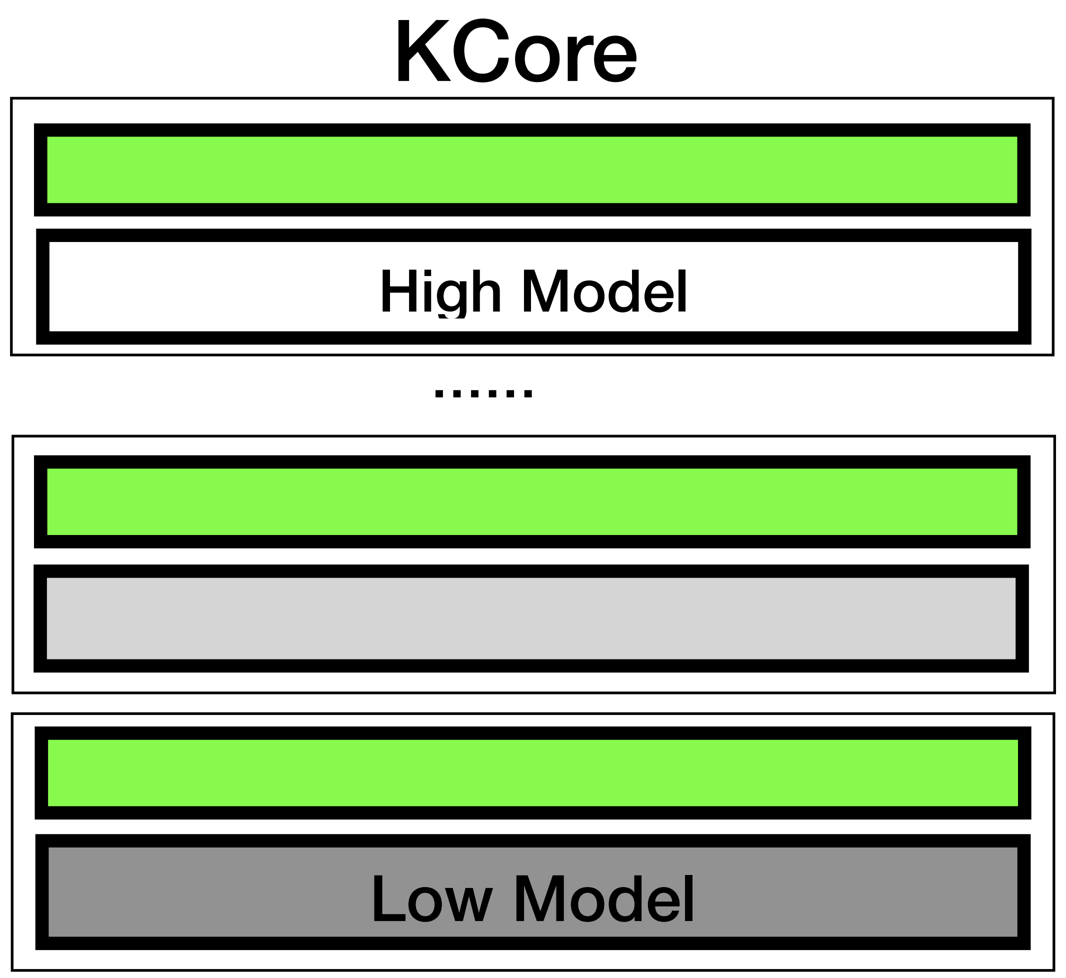
Verify SeKVM using layered hardware model (1)

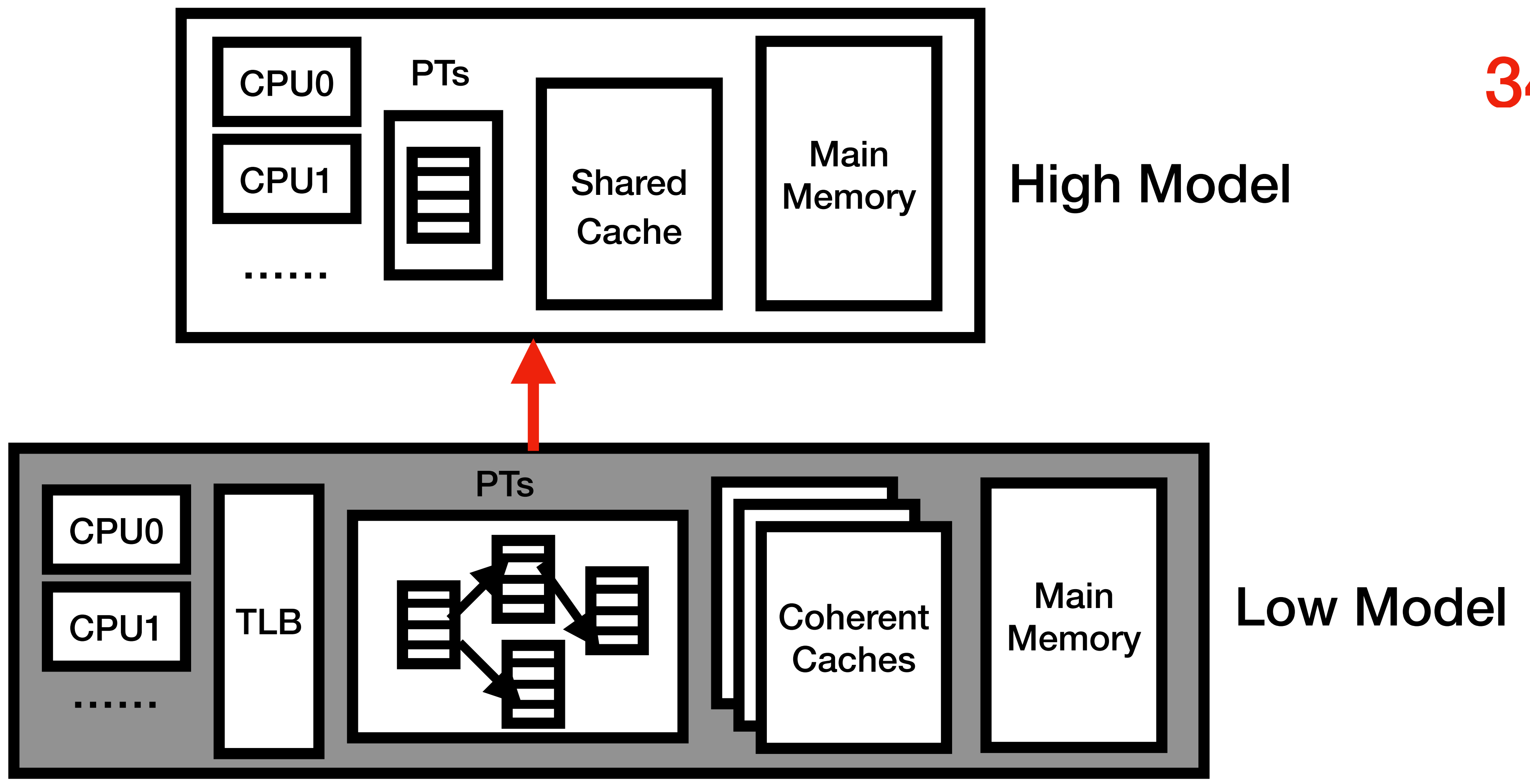
- Verify KCore's lower layered software using the detailed hardware model refines higher layered software with the simpler abstract hardware model
  - Verify higher layered software using the abstract hardware model

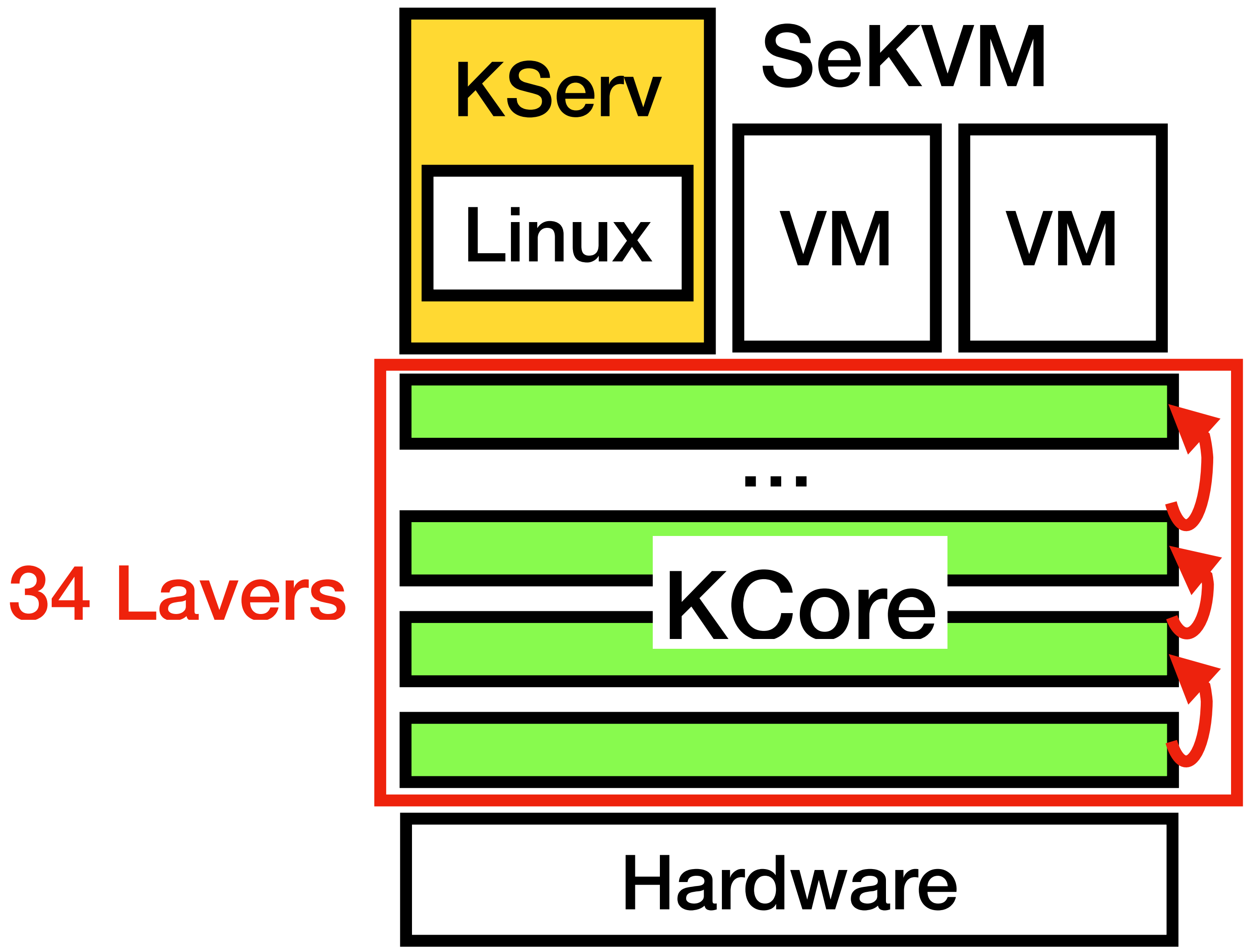


Verify SeKVM using layered hardware model (2)

- Use Coq to implement the layered hardware model and verify SeKVM
  - Verify functional correctness of KCore
  - Verify SeKVM's protection of VM data







KCore	LOC
verified C and ASM code	3.8K

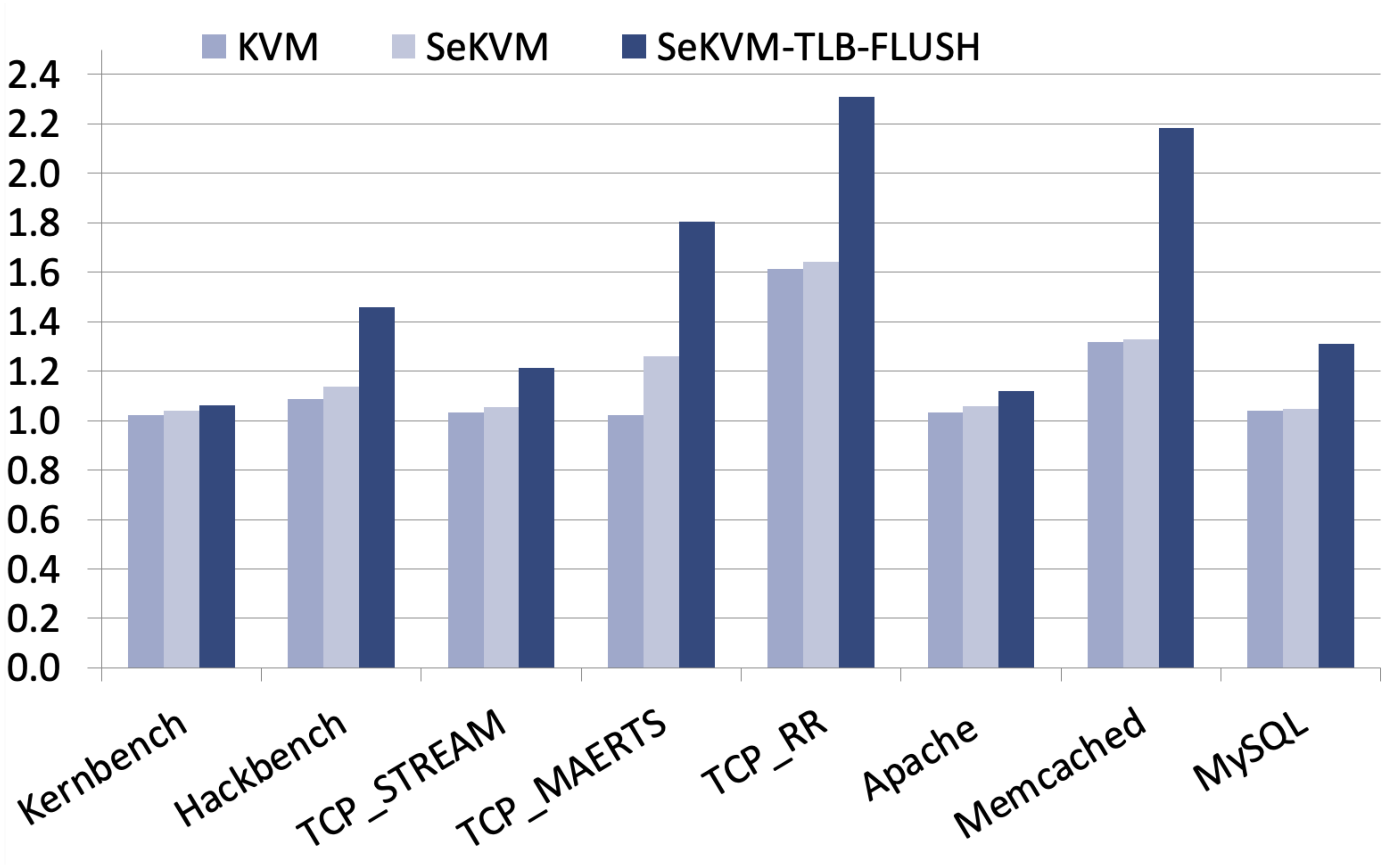
Performance Evaluation

Experimental Setup

- Measure network benchmarks from a bare metal client communicating with the server in the VM
- VMs using virtio with end-to-end encrypted I/Os
- All workloads run on Arm server using Linux/KVM v4.18 based systems on Ubuntu 16.04

Applications	Description
Kernbench	Kernel compile
Hackbench	Scheduler stress
Netperf	Network performance
Apache	Web server stress
Memcached	Key value store
MySQL	Database workload

Performance Evaluation



Summary

- Introduced a layered hardware model that is simple to use for verification while accounting for realistic multiprocessor hardware features
- Used the model to verify the correctness and security guarantees of SeKVM, a multiprocessor KVM implementation
- SeKVM takes advantage of the widely used multiprocessor features to retain KVM's commodity feature set and performance

Q&A