# Safe User-Level Sharing of Memory-Mapped Resources

Michael L. Scott


www.cs.rochester.edu/research/synchronization/

Joint work with Mohammad Hedayati, Chris Kjellqvist,
Spyridoula Gravani, Ethan Johnson, and John Criswell at Rochester;
Kai Shen and Mike Marty at Google

Hydra Distributed Systems Conference, July 2020

# The University of Rochester

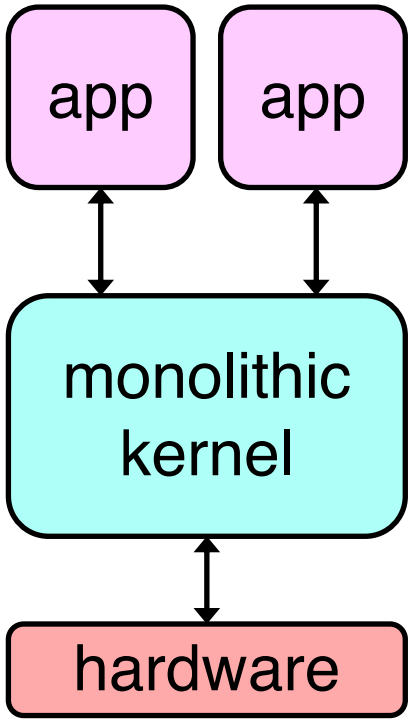

- Small private research university
- 6800 undergraduates
- 5000 graduate students
- Set on the Genesee River in Western New York State, near the south shore of Lake Ontario
- 250km by road from Toronto; 590km from New York City

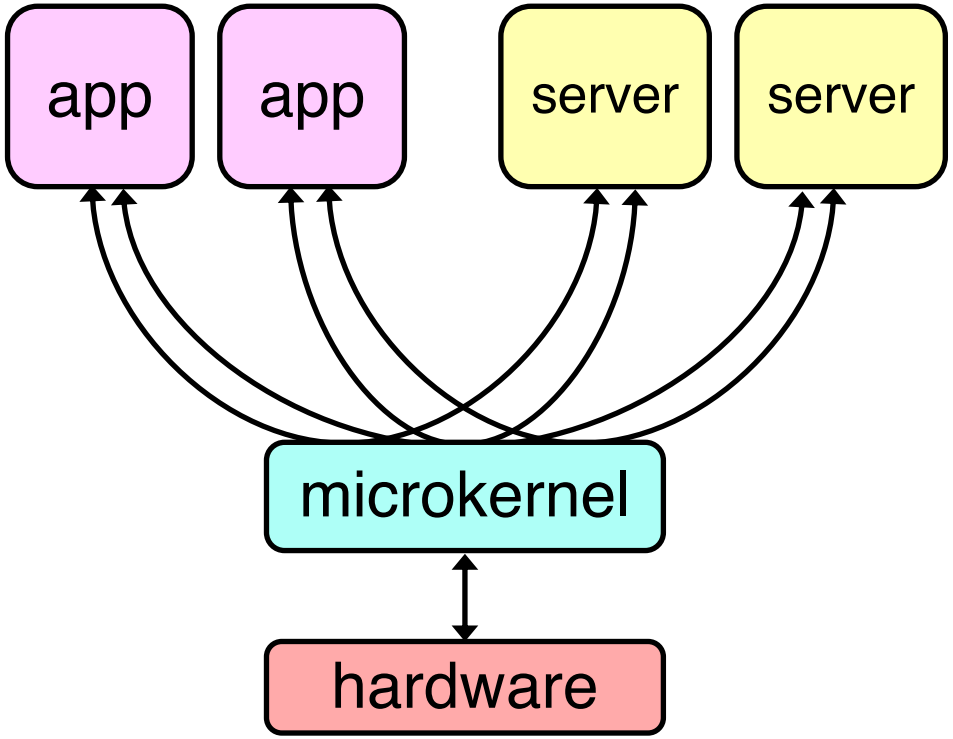

# The Computer Science Dept.

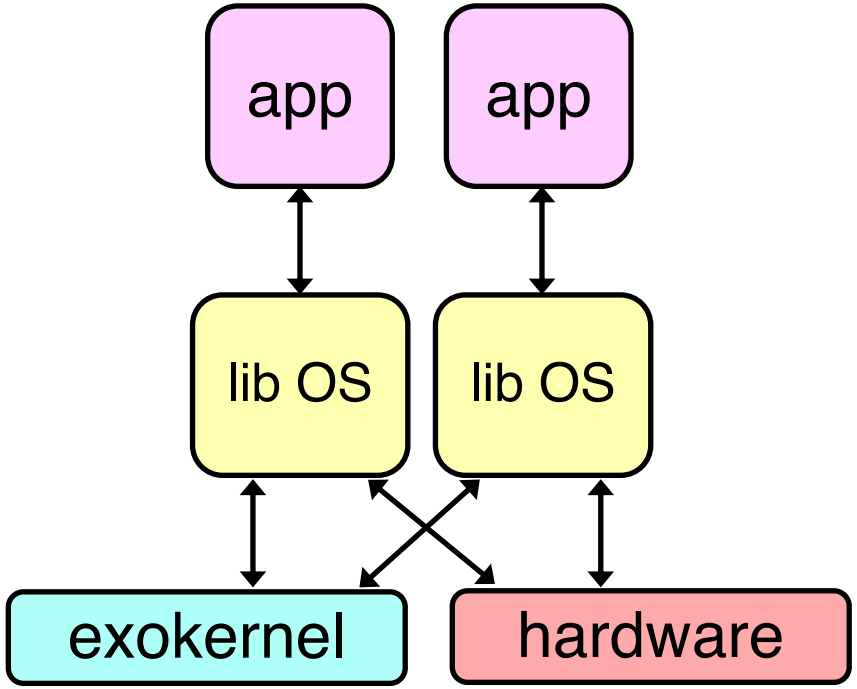
- Founded in 1974
- 20 tenure-track faculty; 70 Ph.D. students
- Specializing in AI, theory, HCI, and parallel and distributed systems
- Among the best small departments in the US

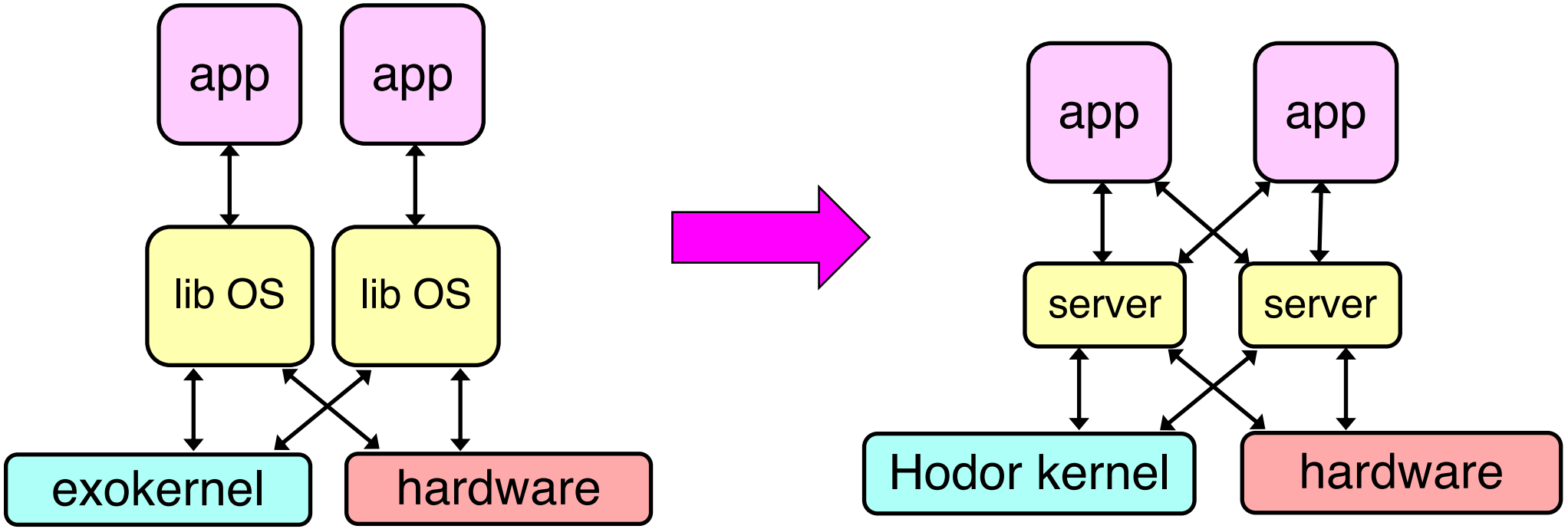
# A Bit of OS History

- Monolithic kernels
  - All OS functionality in a single large kernel — long the dominant approach
  - Fast, but hard to understand / modify / customize / secure
- Microkernels
  - Increase modularity, for easy update / customization; bug containment
  - Export functionality into servers; communicate via the kernel
- Exokernels / library OSes / kernel-bypass IO
  - Export functionality into *library* for faster access
  - Kernel involved only on setup; app subsequently touches hardware directly


app
app
monolithic kernel
hardware

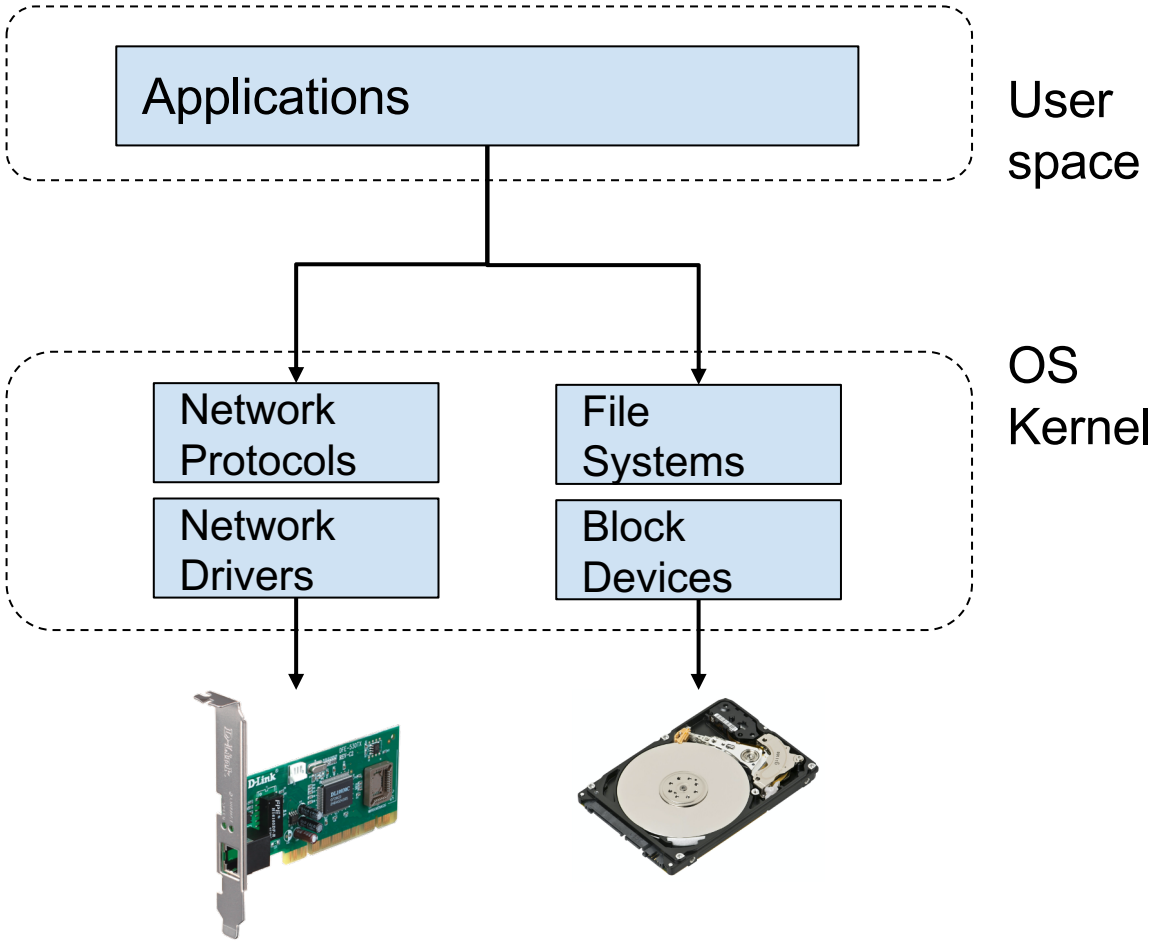

app
app
server
server
microkernel
hardware


app
app
lib OS
lib OS
exokernel
hardware


app
app
lib OS
lib OS
exokernel
hardware
app
app
server
server
Hodor kernel
hardware

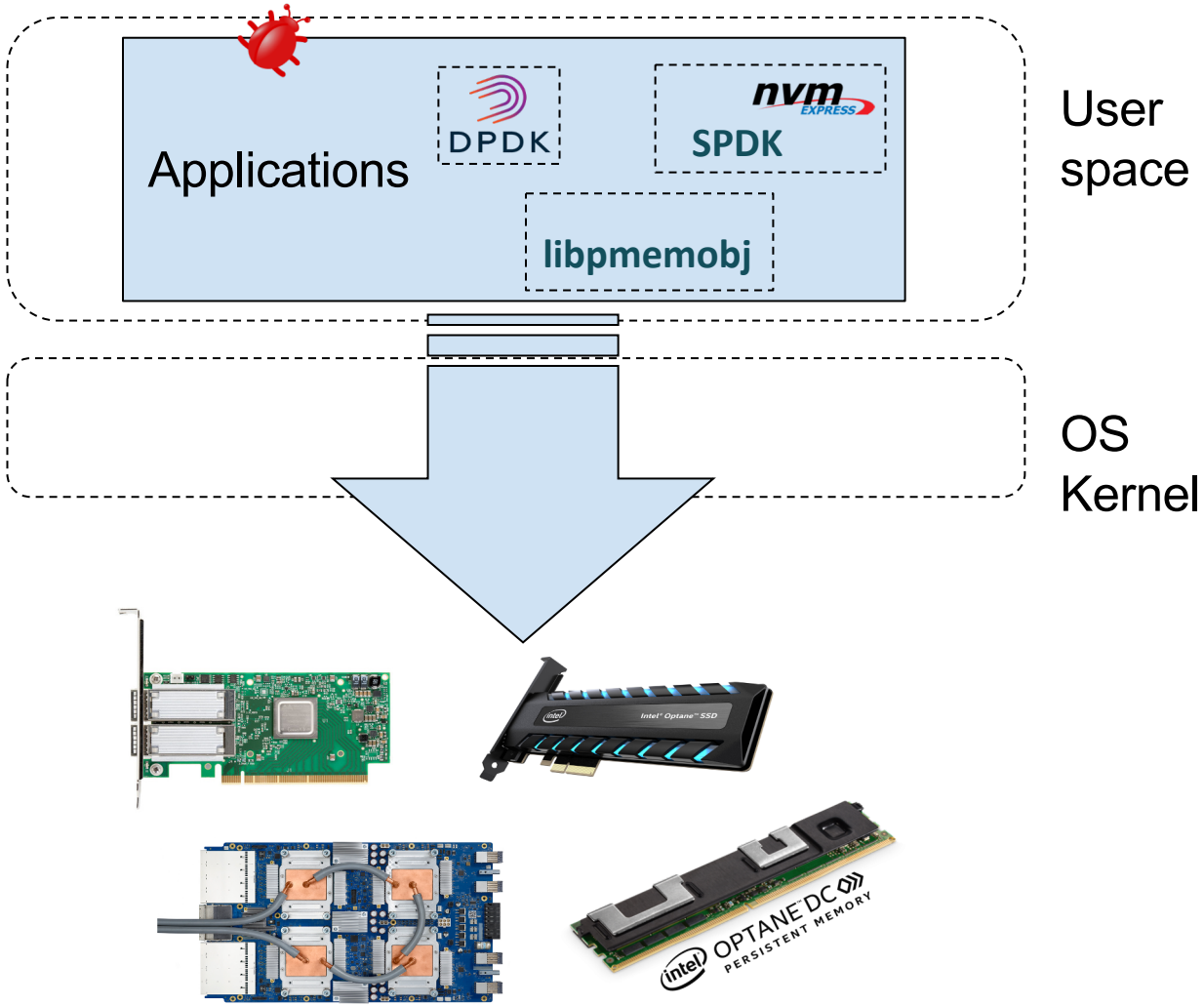
# Kernel I/O

- Enables sharing
- Provides guarantees
  - Fairness
  - Recovery
  - Security
- All without needing to trust



# Kernel-bypass I/O

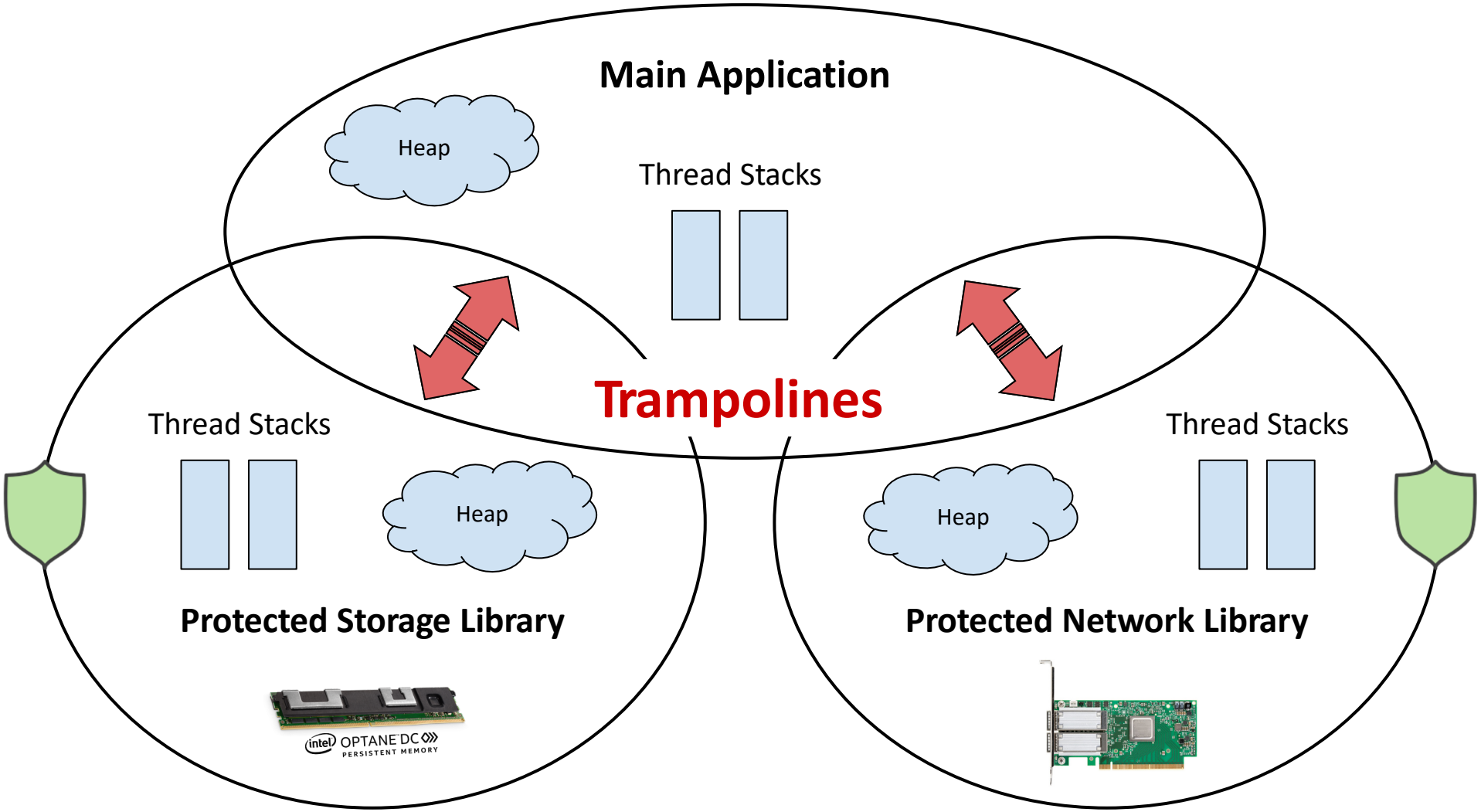
- Pros:
  - Lower latency
  - Rapid development
  - Specialization
- Cons:
  - No guarantees (e.g., of fairness)
  - Hard to multiplex



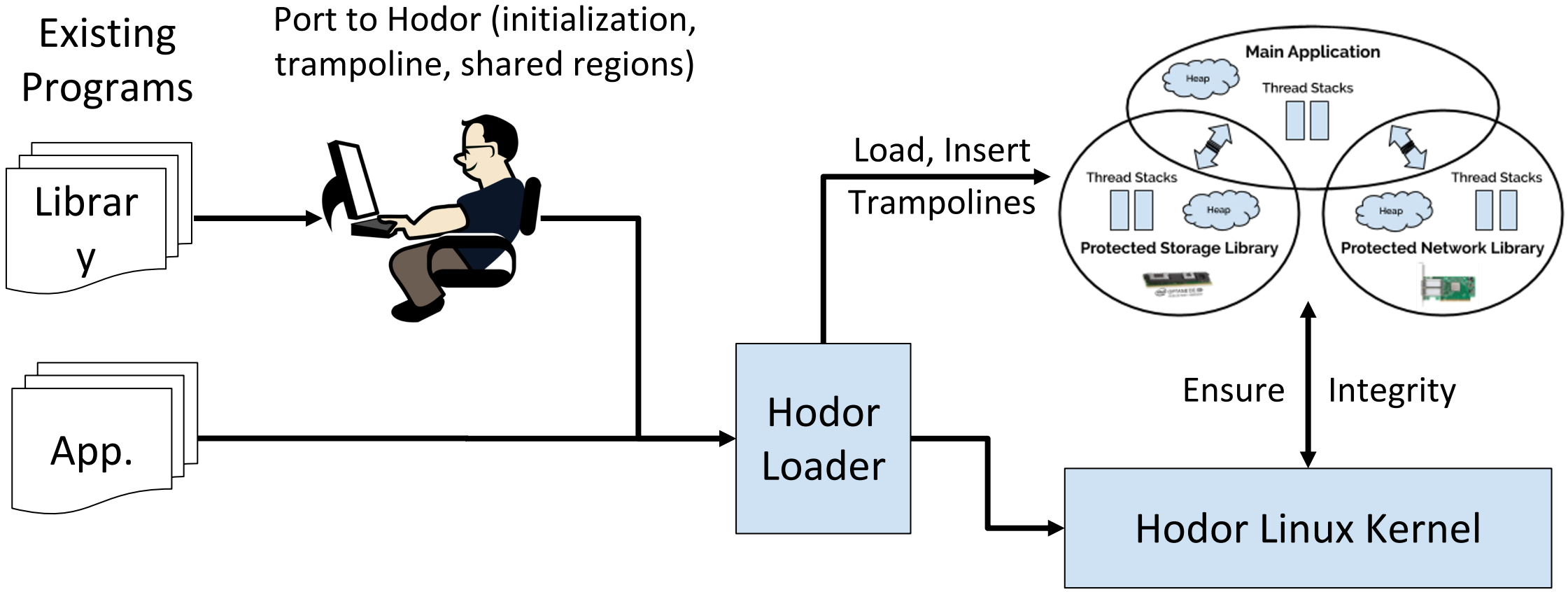
# Overview

- Motivation
- Design of Hodor
- Fast Memory Isolation
- IO Performance
- Cross-Application Sharing
- Conclusion

# Protected Library



# Hodor


Existing Programs
Port to Hodor (initialization, trampoline, shared regions)
Librar y
App.
Load, Insert Trampolines
Hodor Loader
Main Application
Heap
Thread Stacks
Thread Stacks
Heap
Protected Storage Library
Thread Stacks
Heap
Protected Network Library
Ensure
Integrity
Hodor Linux Kernel

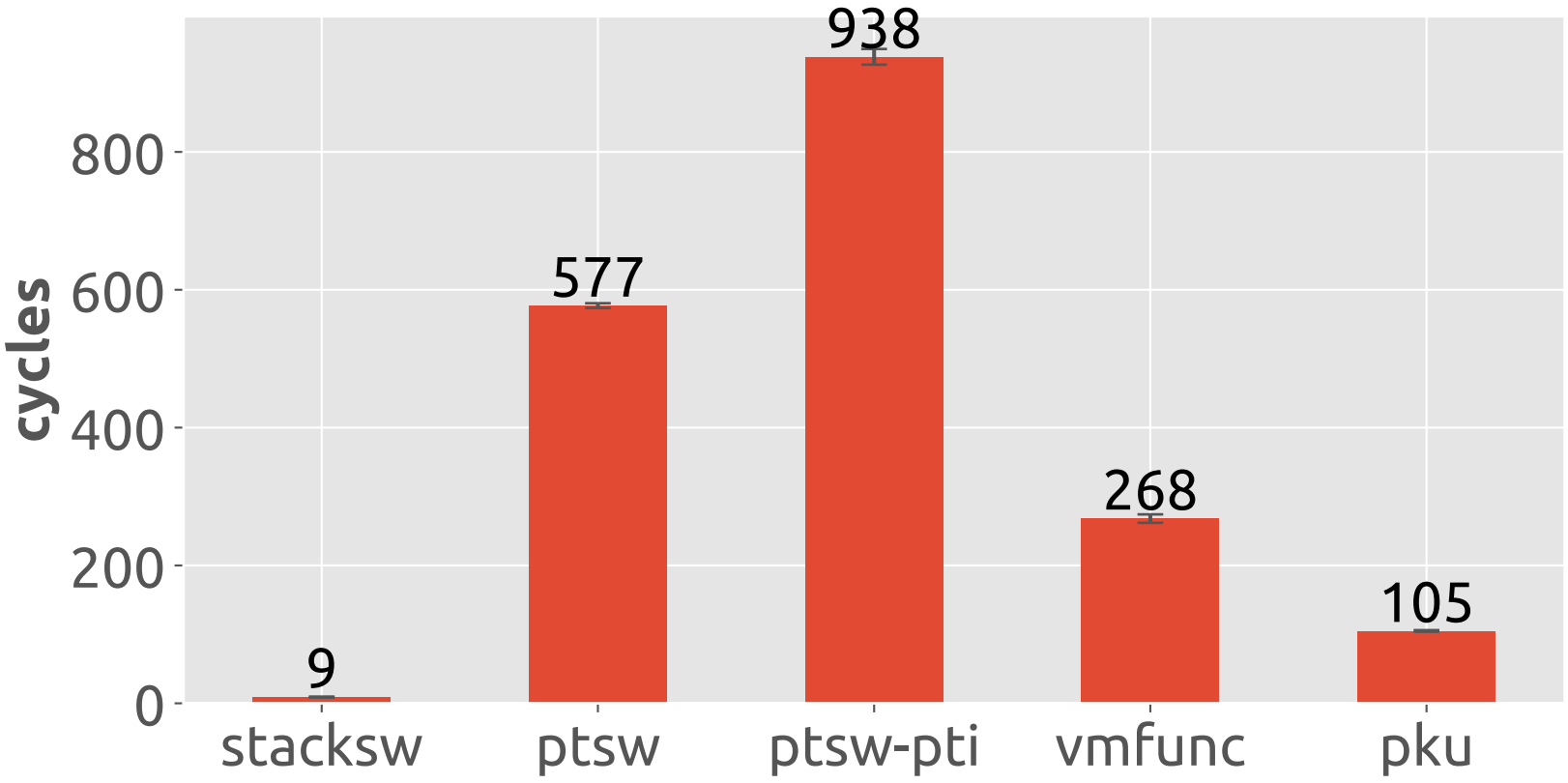
# Overview

- Motivation
- Design of Hodor
- Fast Memory Isolation
- Evaluation
- Conclusion

# Possible isolation strategies

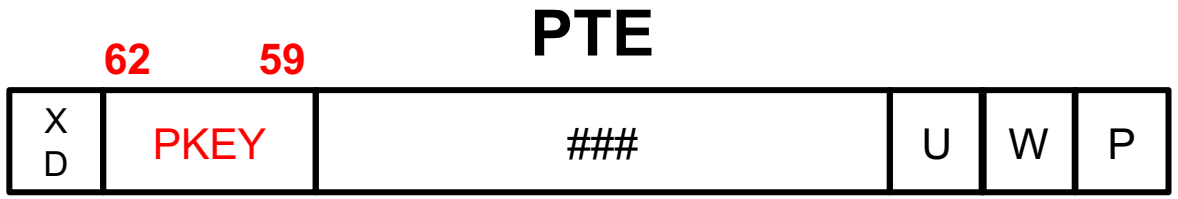
- Separate address spaces
  - » Conventional hardware; high switching overhead
- Software fault isolation (SFI — a.k.a. sandboxing)
  - » Trusted compiler; no special hardware; high overhead throughout app
- Virtualization (as in Dune [Belay OSDI 2012])
  - » Hardware enforced; single protected domain; high switching overhead; additional overhead from VMEXITs and 2-level paging
- VMFUNC on recent Intel machines
  - » Multiple protection domains in one VM guest; low switching overhead; but still the overhead of VMEXITs and 2-level paging
- ★ User-space protection keys [Hedayati ATC 2019; Vahldiek-Oberwagner Sec 2019]

# Cost of an empty library call

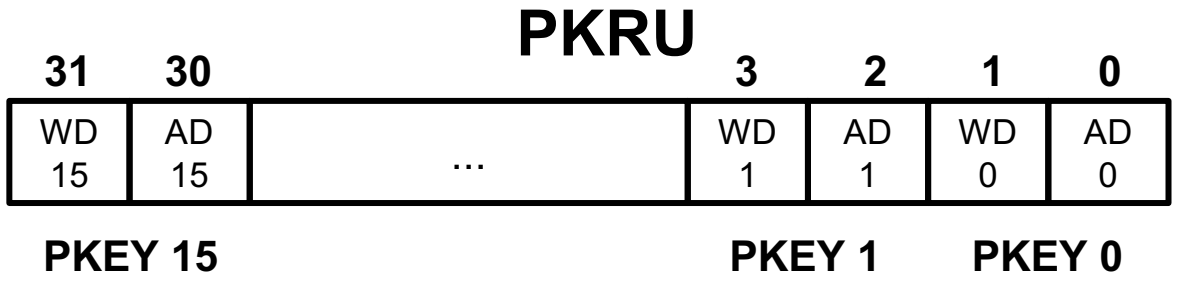


# Preferred strategy: Intel PKU

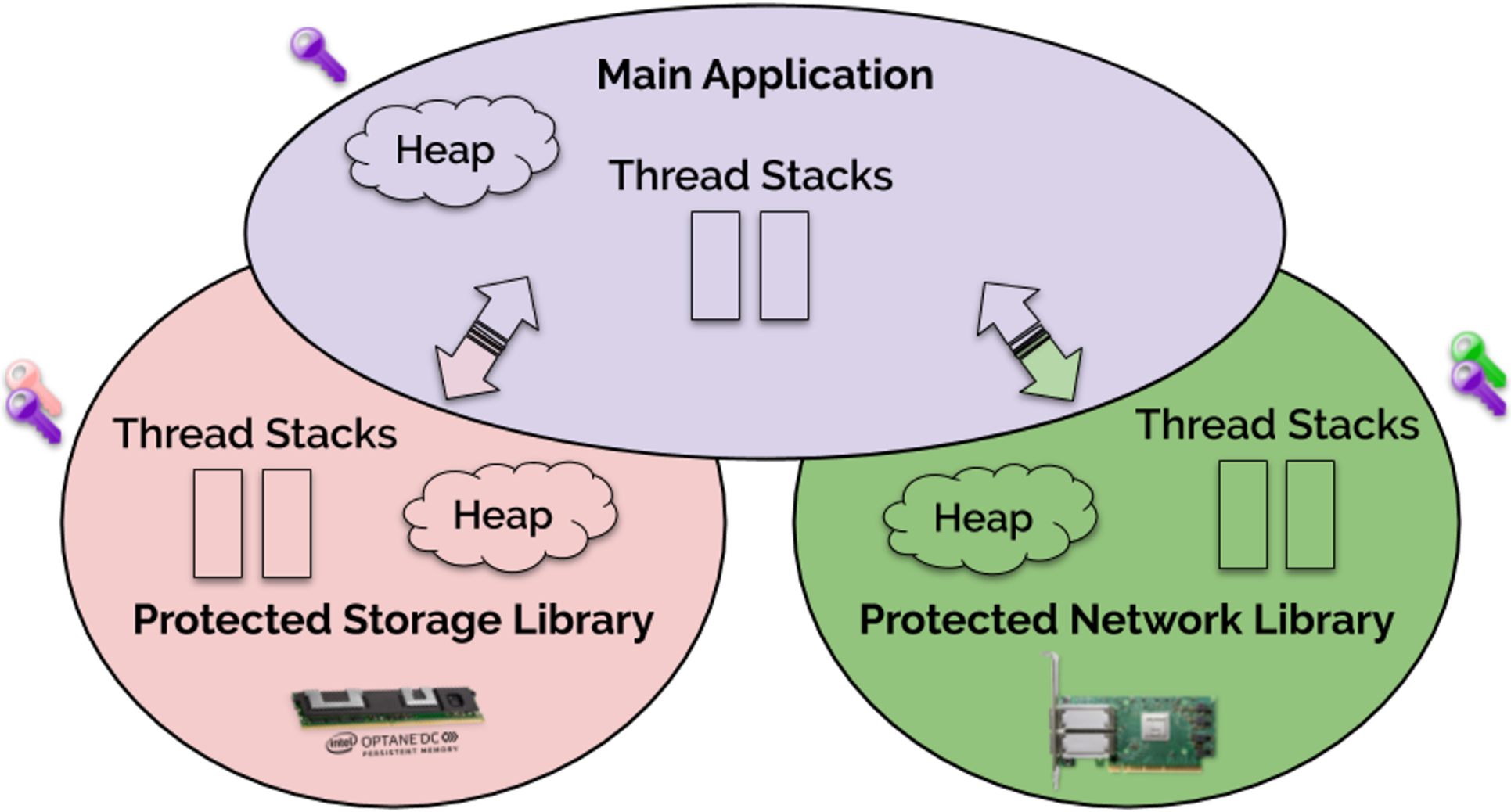
- Protection Keys for User-Space (a.k.a. MPK)
- Introduced in Skylake-SP



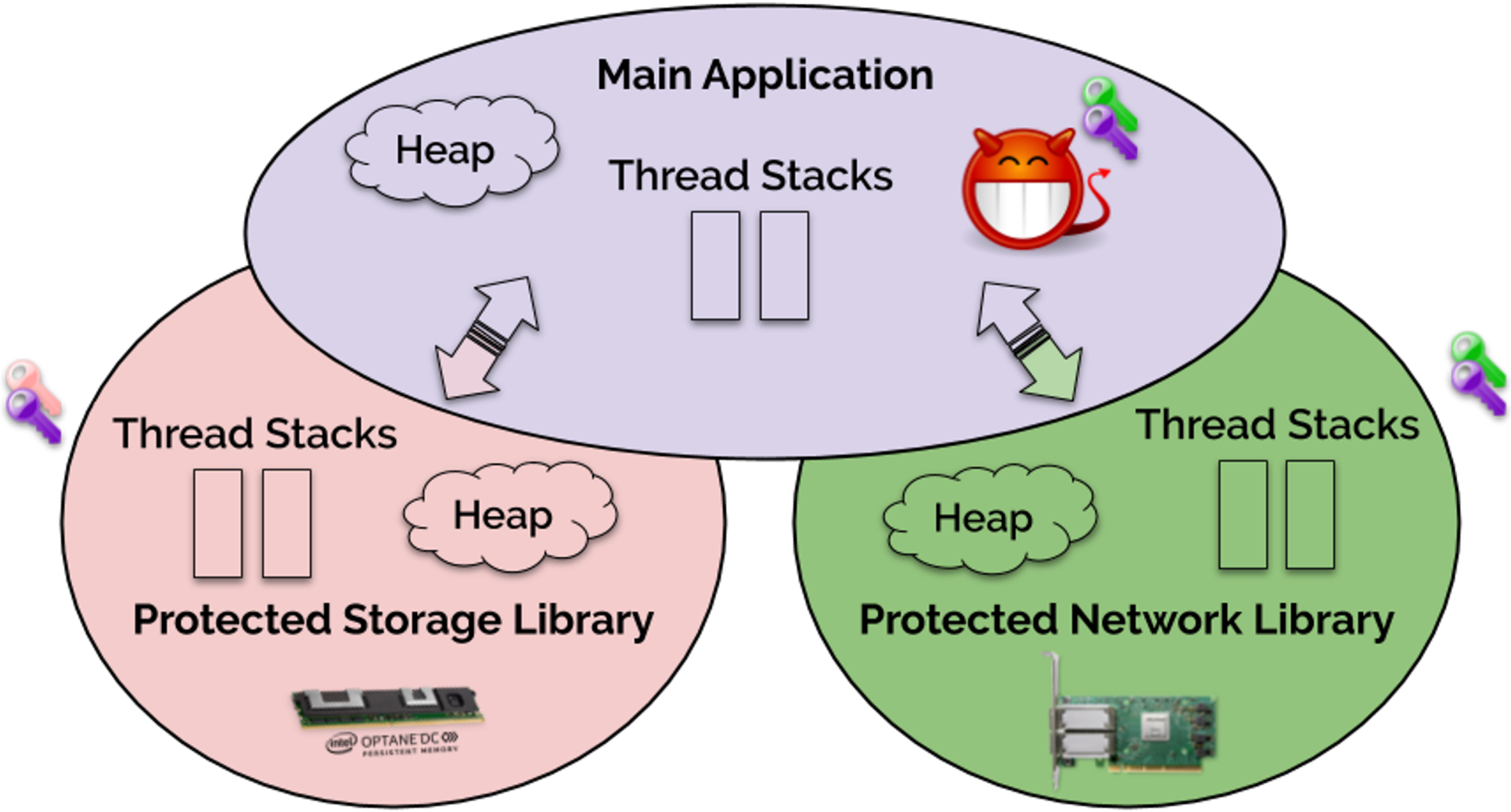

- 32-bit PKRU register (Access/Write Disable)
- WRPKRU/RDPKRU



# Hodor: Memory Isolation



# Hodor: Memory Isolation



# Hodor: Vetting `WRPKRUs`

- Inspect executable regions
  - » Load (by Hodor loader)
  - » W⟶X change (by Hodor kernel at run-time)
- Look for WRPKRU (0f 01 ef) instances

**glibc-devel-static-2.27-alt9.x86_64**

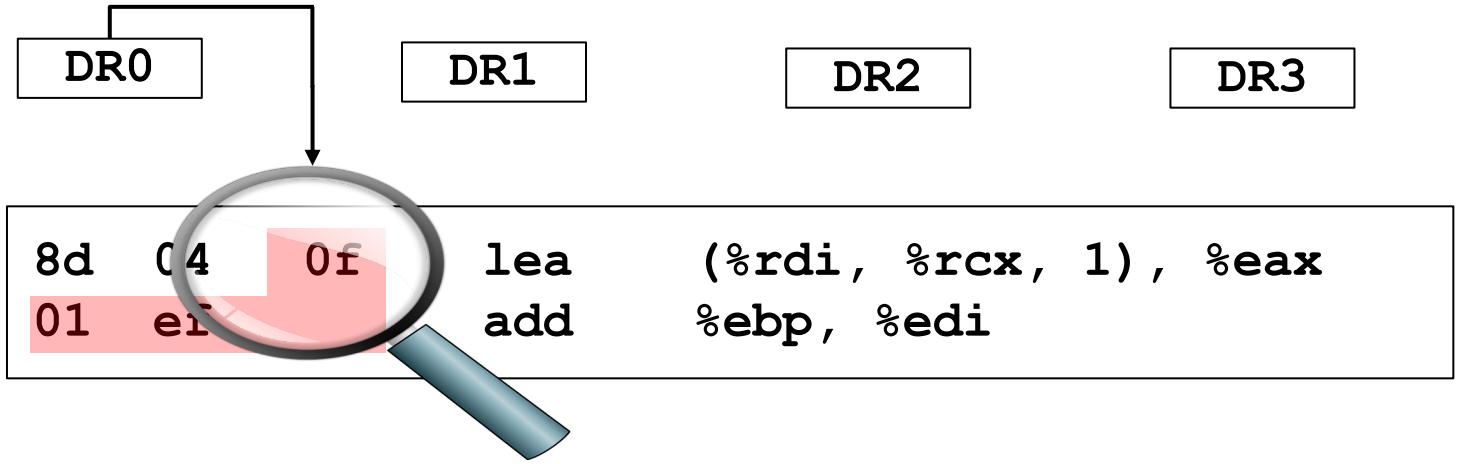
```
f7 d2          not     %edx
21 d0          and     %edx,%eax
44 89 c2       mov     %r8d,%edx
09 f0          or      %esi,%eax
0f 01 ef       wrpkru
31 c0          xor     %eax,%eax
```

**blender-2.79b-7.fc29.x86_64**

```
8d  04   0f     lea     (%rdi, %rcx, 1), %eax
01  ef          add     %ebp, %edi
```

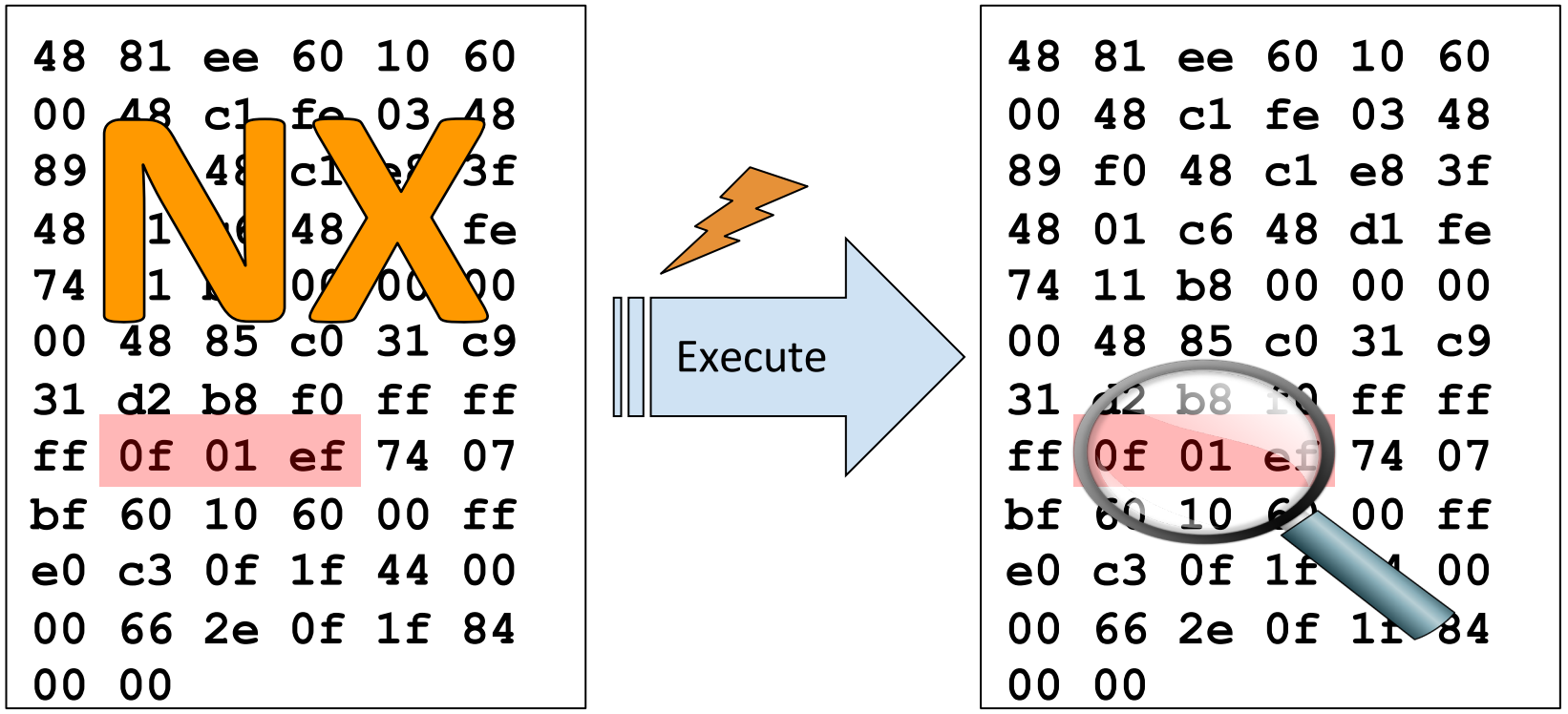
# Hodor: Vetting `WRPKRUs`

- Hardware watchpoints
  - » DR# registers point to the beginning of illegal byte sequence
  - » No spurious traps when correctly aligned execution runs past an implicit instance



# Hodor: Illegal `WRPKRUs`

- Limited hardware watchpoints
  - » Only 4 on Intel Processors
  - » HW watchpoints as cache for illegal sequences

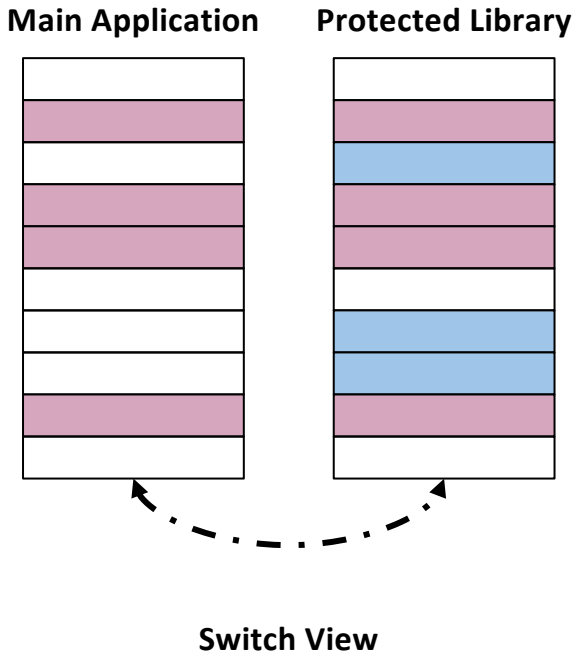


# Hodor: Vetting `WRPKRU`s

- Vetting cost
  - » Implicit instances incur no run-time overhead
  - » Explicit instances should use pkey_set()
  - » No measurable overhead as long as:
    - – #hot illegal seq. fewer than #hw watchpoints
- How often?
  - » 58,273 rpm packages on Fedora 29 (108K executables)
  - » Only 123 binaries with one or more illegal byte sequences
    - – Only 2 (less than 0.02%) with more than 4

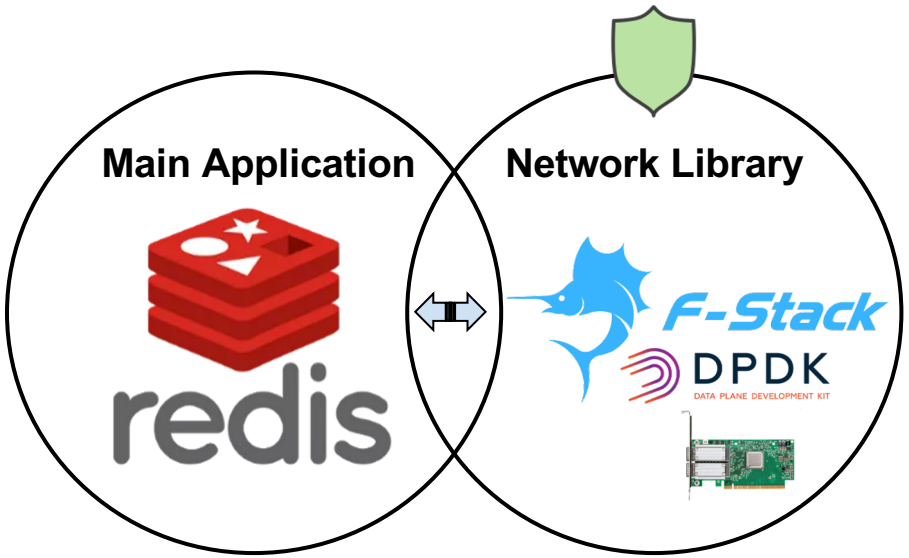
# Alternative Memory Isolation

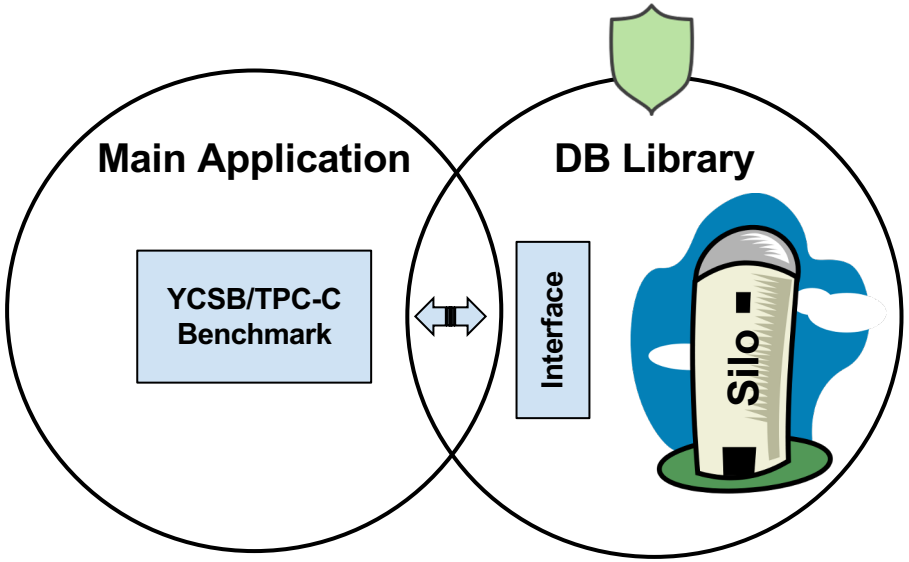
- Per-Domain Page-Table
  - » Each mapping the view of a domain
  - » Switch using system calls
- Per-Domain Extended Page-Table
  - » Requires running virtualized (in Intel VMX)
  - » Switch using VMFUNC w/o causing VMEXIT



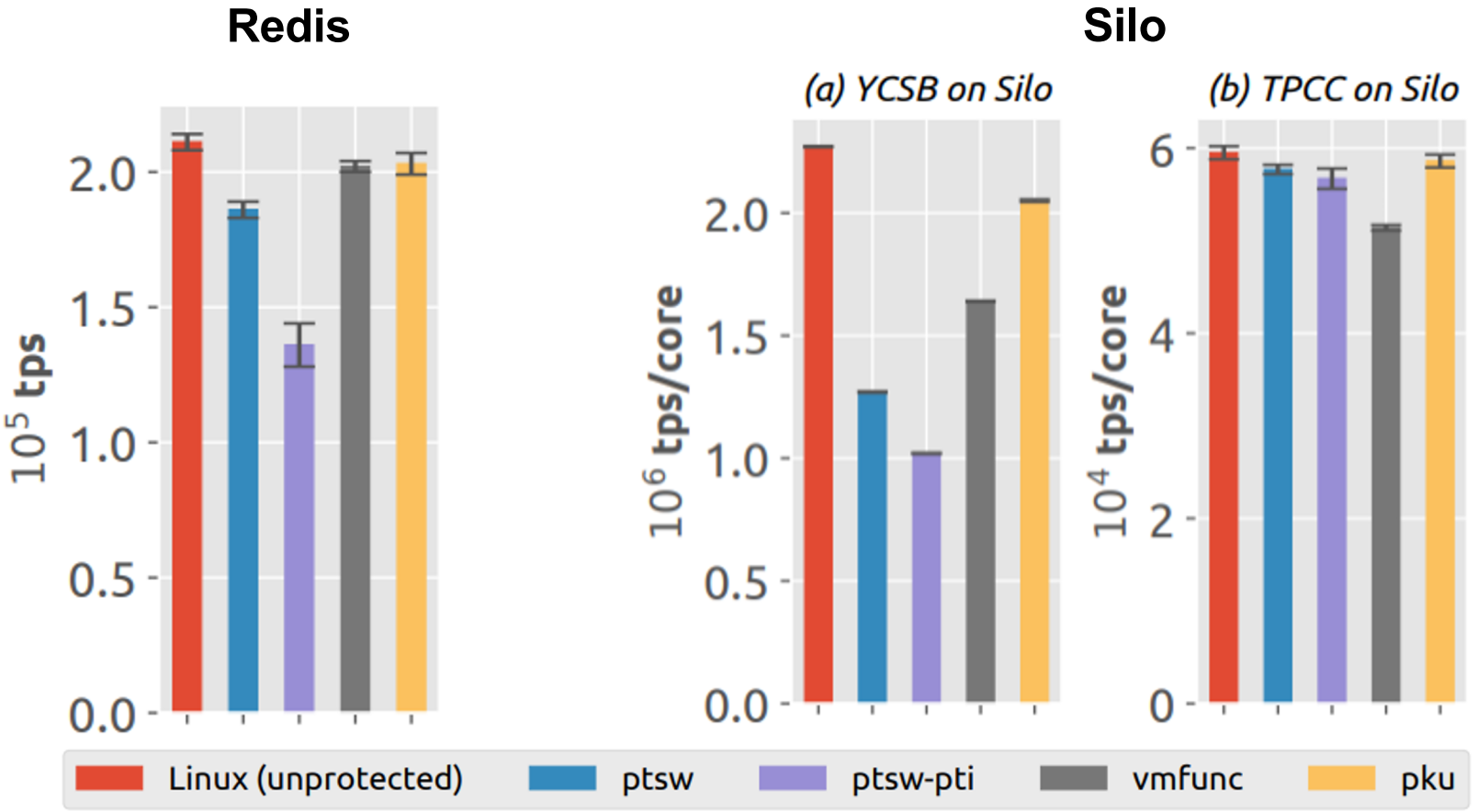
# Evaluation: Applications

- Redis (kernel-bypass network TCP/IP stack)
- Silo (in-memory database)
- DPDK (kernel-bypass packet processing) -- in the ATC paper



# Evaluation: Applications

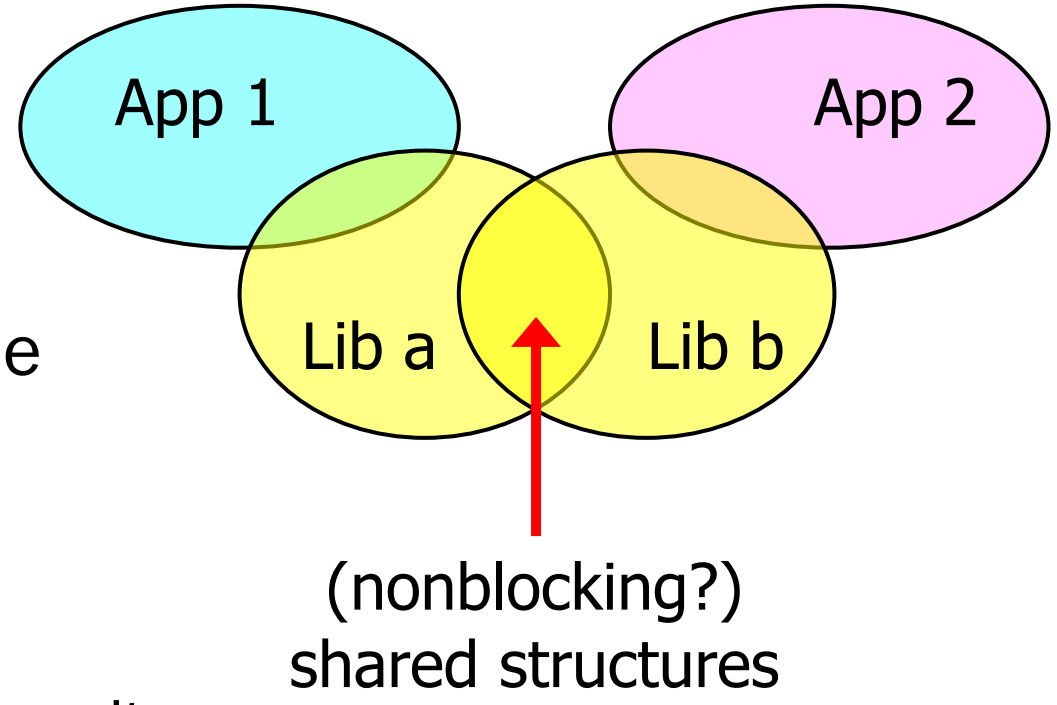



**ptsw**: page table switching
**vmfunc**: EPT switching

**ptsw-pti**: page table switching w/ KPTI
**pku**: using memory protection keys

# Cross-application sharing

- The network stack in Redis was used by one application at a time, as was the Silo database
- Sharing the network stack would enable system-wide QoS guarantees; sharing an in-core database is also appealing — Memcached as an example
- Threads of different applications can fail independently
  - » Ideal use case for nonblocking structures, as originally envisioned by the theory community

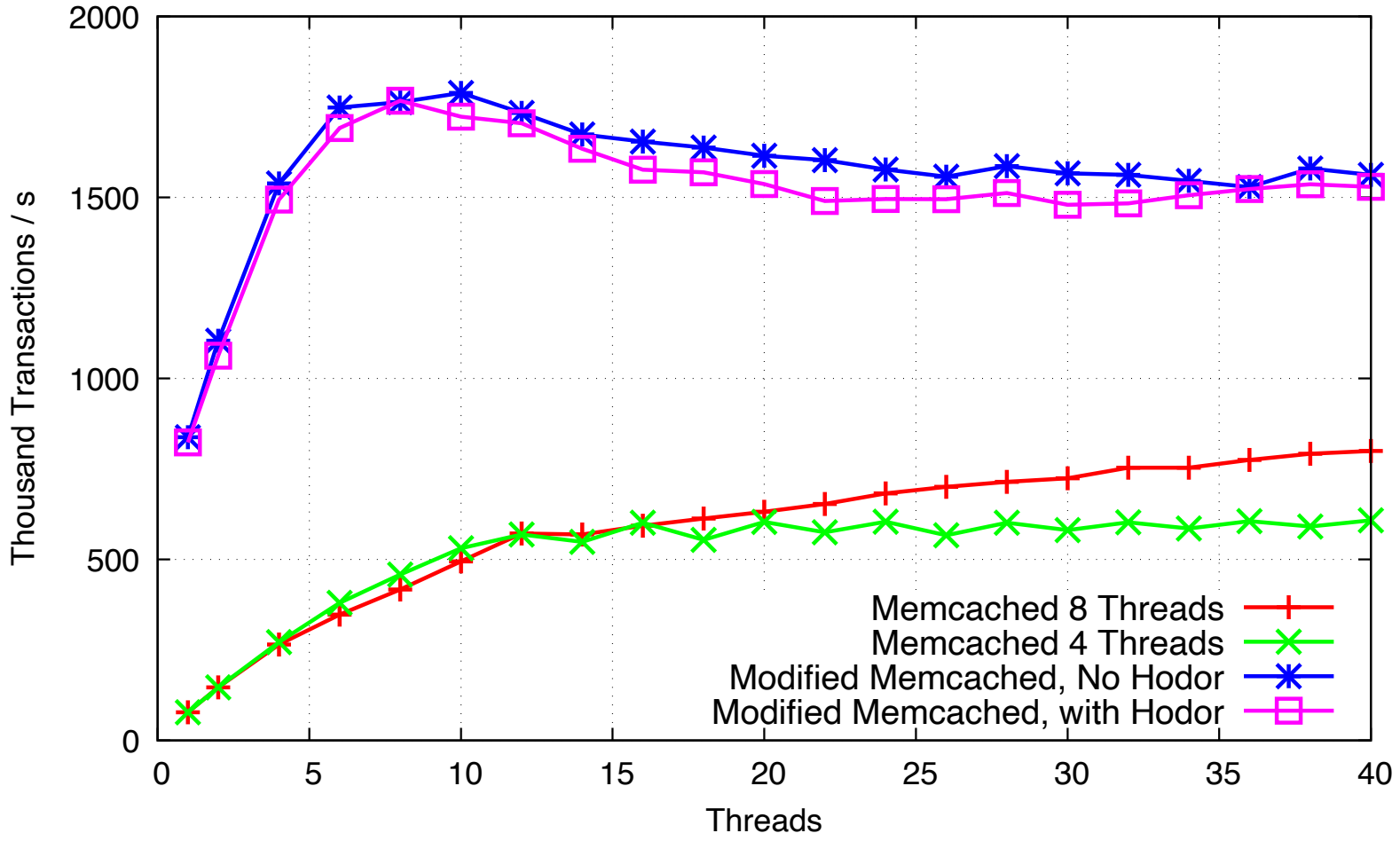


# Memcached

- Widely used in datacenters and smaller operations
  - » Typically accessed over a network, but also within a single machine, where socket overhead seems wasteful
  - » Typically used to cache disk contents, but also standalone
- Converted to work with Hodor
  - » Replaced “slab” allocator with Ralloc [Cai ISMM 2020]
  - » Elided network code
  - » Added trampolines and persistence labels
  - » Rely on Hodor to complete execution of library calls on process crash

# Memcached results

- Original 26KLoC; added ~600; removed ~6.8K (5.2k network; 1.6K alloc)
- Latency of individual ops improved by 11–56x; throughput by ~2x



# Caveats

- Protected libraries must be written with care
  - » Cannot fail in library code: must be bulletproof
  - » Must finish all ops in bounded time (kernel has limited patience)
  - » Cannot trust user data: must validate before using
  - » Cannot trust data not to change: must copy in before validating
  - » Cannot trust user locations: must copy out after dropping locks
- These are the same set of rules required for kernel code

# Conclusion

Introduced *protected libraries.* Showed that:

- Intel PKU can safely be used for isolation, retaining high performance
  - 90–98% of unprotected throughput
- Libraries can enforce global OS policy
- Mutually untrusting applications can share a protected library
  - Perform “server” operations using their own threads, without IPC
  - memcached as an example
- Appealing alternative to microkernels: low-cost modularity
  - Lots of future work!


www.cs.rochester.edu/research/synchronization/

www.cs.rochester.edu/u/scott/