Persistent State Machines for Recoverable In-memory Storage Systems with NVRam

Wen Zhang
UC Berkeley

Scott Shenker
UC Berkeley
ICSI

Irene Zhang
Microsoft Research
University of Washington

Distributed in-memory systems are ubiquitous

Memcached

redis

Building Consistent Transactions with Inconsistent Replication

Irene Zhang Naveen Kr. Sharma Adriana Szekeres
Arvind Krishnamurthy Dan R. K. Ports

Just Say NO to Paxos Overhead:
Replacing Consensus with Network Ordering

Jialin Li Ellis Michael Naveen Kr. Sharma Adriana Szekeres Dan R. K. Ports

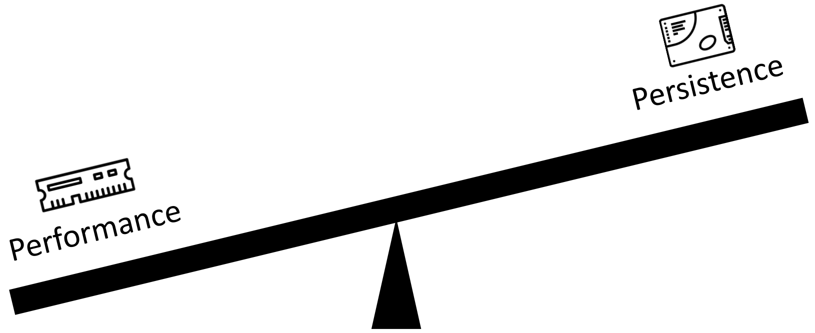
In-memory systems: Pros and cons

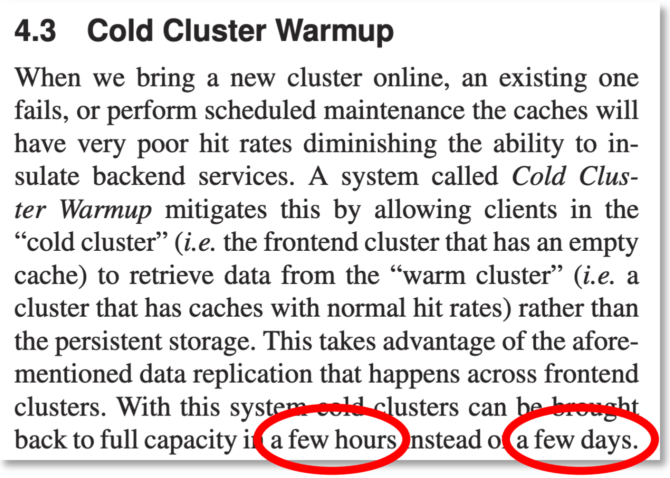
✅ High performance.

- Kernel bypass ➔ microsecond-level latency in datacenter.

❌ No persistence.

- Node failure ➔ recover from replica or storage.
- Datacenter failure ➔ potential data loss.

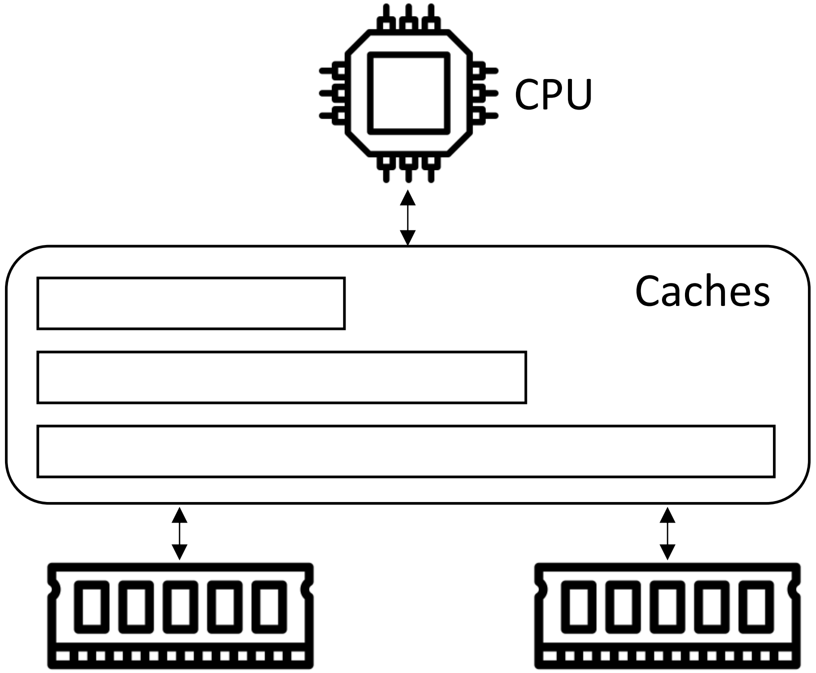


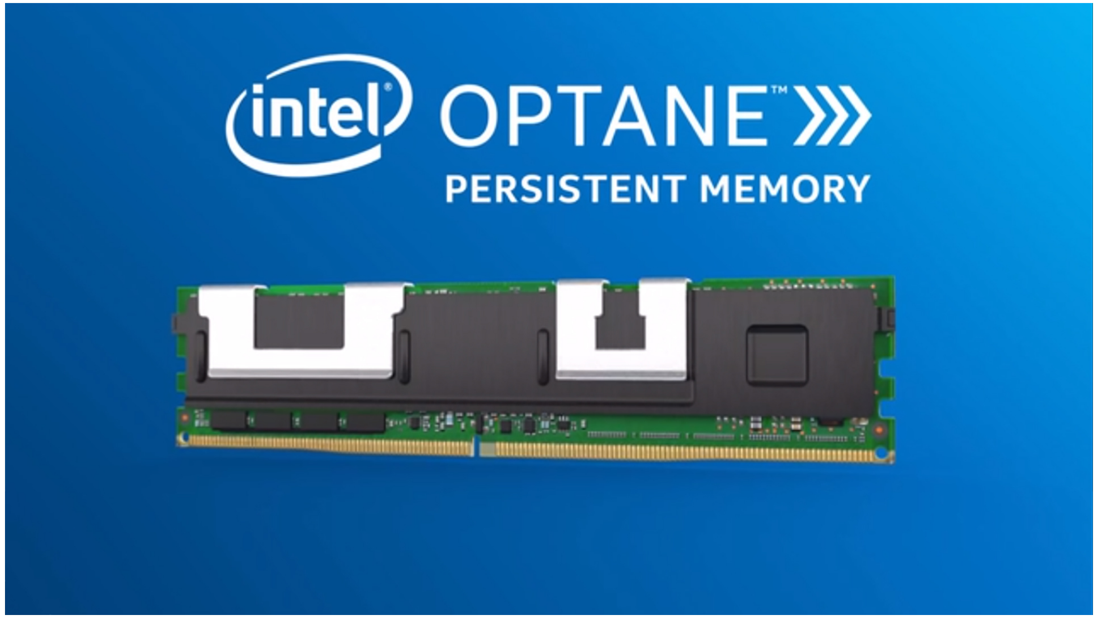


4.3 Cold Cluster Warmup

When we bring a new cluster online, an existing one fails, or perform scheduled maintenance the caches will have very poor hit rates diminishing the ability to insulate backend services. A system called *Cold Cluster Warmup* mitigates this by allowing clients in the "cold cluster" (*i.e.* the frontend cluster that has an empty cache) to retrieve data from the "warm cluster" (*i.e.* a cluster that has caches with normal hit rates) rather than the persistent storage. This takes advantage of the aforementioned data replication that happens across frontend clusters. With this system cold clusters can be brought back to full capacity in a few hours instead of a few days.

Persistent memory (PM) is here





Persistent   High performance

Persimmon

Using PM to add persistence to in-memory storage systems.

Outline

- **Background**: Challenges and key insight
- **Persimmon overview**: API and guarantees
- **Persimmon runtime**: Design and implementation
- **Evaluation**: Programming experience and performance

Outline

- **Background**: Challenges and key insight
- **Persimmon overview**: API and guarantees
- **Persimmon runtime**: Design and implementation
- **Evaluation**: Programming experience and performance

Porting in-memory systems to PM is not trivial

Redis

+

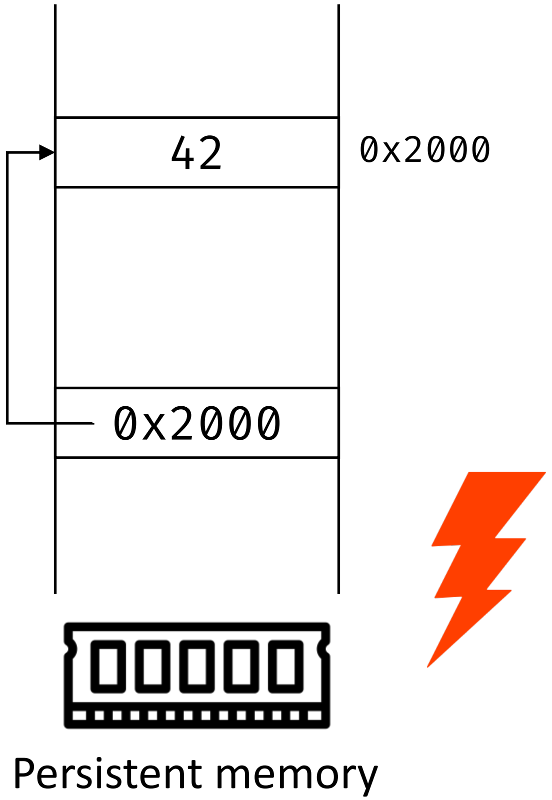
intel OPTANE™ DC
PERSISTENT MEMORY

≠

Persistent Redis

Challenge: crash consistency for PM

Definition: Operations must persist **all-or-nothing**.



Challenge: crash consistency for PM

Definition: Operations must persist **all-or-nothing**.

Applications typically use logging for atomicity & recoverability.

- Requires complex code.
- Can incur high overhead.

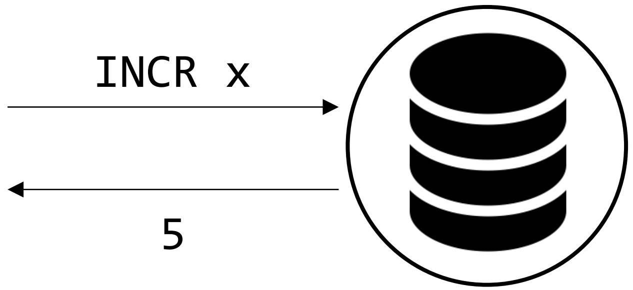
> Although Redis is highly-optimized for DRAM, porting it to NVMM is not straightforward and requires large engineer- ing e
> with

> (§ 3). Our findings were interesting, and in some cases, quite surprising. A big takeaway was that this exercise can be surprisingly non-trivial. The required lower level changes were contagious and quickly became pervasive.

How to use PM to provide persistence with minimal **programming effort** and **performance overhead**?

Key insight

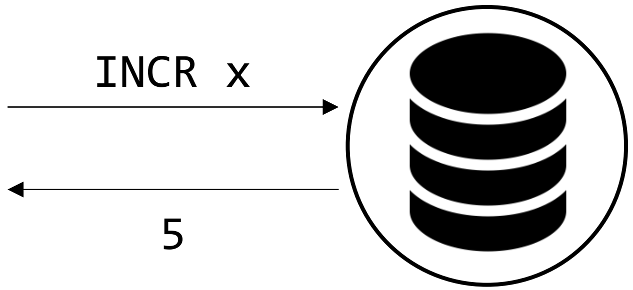
In-memory storage systems are state machines



State machine properties:

- Encapsulate state.
- Have atomic operations.
- Execute operations deterministically.

Solution: State machines as PM abstraction



☺ Encapsulates persistent state for recovery.

☺ State machine operation = units of persistence.

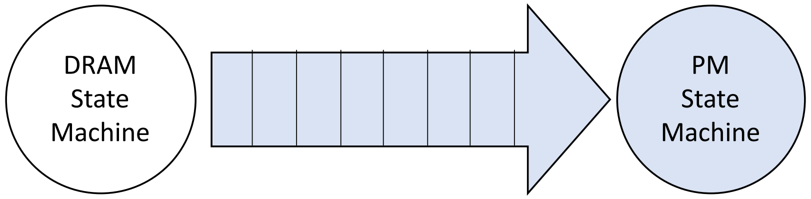
☺ Determinism ➔ persistence via operation logging.

Outline

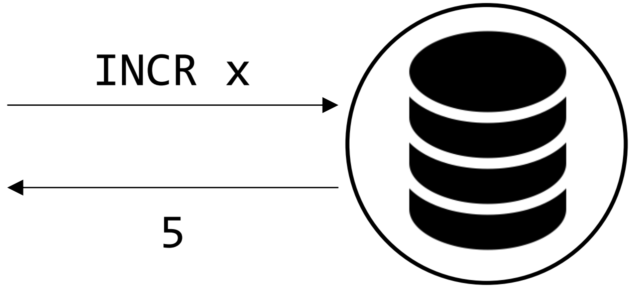
- **Background**: Challenges and key insight
- **Persimmon overview**: API and guarantees
- **Persimmon runtime**: Design and implementation
- **Evaluation**: Programming experience and performance

The Persimmon system

- A user-level runtime system that provides persistence to in-memory state machine applications.
- Keep **2 copies of state machine**: one in DRAM, one in PM.
- On RPC-handling path: state machine **operation logging**.
  - Persistence with low latency overhead!
- In the background: **shadow execution** on **PM state machine**.
  - For crash consistency: Dynamically instrumented for undo logging.



Application model: State machine



State machine operations are arbitrary application code that:

- Do not have external dependencies.
- Execute deterministically.
- Have no external side-effects.

Assumption: operations are **applied sequentially**.

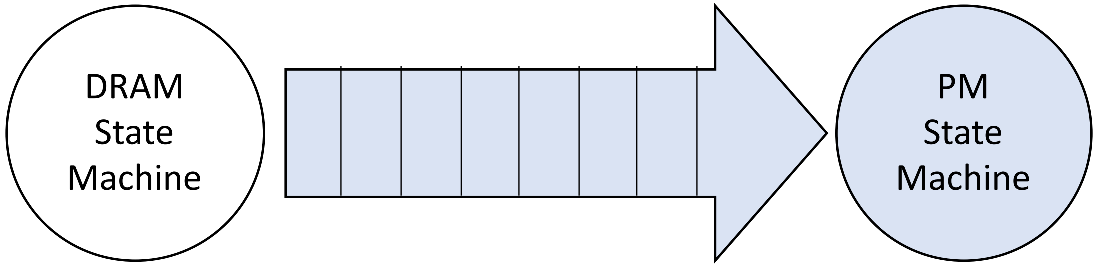
Persistent State Machine (PSM) API

- `psm_init() → bool` – Initialize; return true if in recovery.
- `psm_invoke_rw(op)` – Invoke read-write op with persistence.
- `psm_invoke_ro(op)` – Invoke read-only op without persistence.

Persistent State Machine (PSM) guarantees

- **Linearizability**: All PSM operations are run in order submitted.
- **Durability**: PSM operations are never lost once they return.
- **Failure atomicity**: If crash before PSM operations, recover to state either before or after.

Persimmon design: Pros and cons



☺ Low programming effort

☺ Low latency overhead

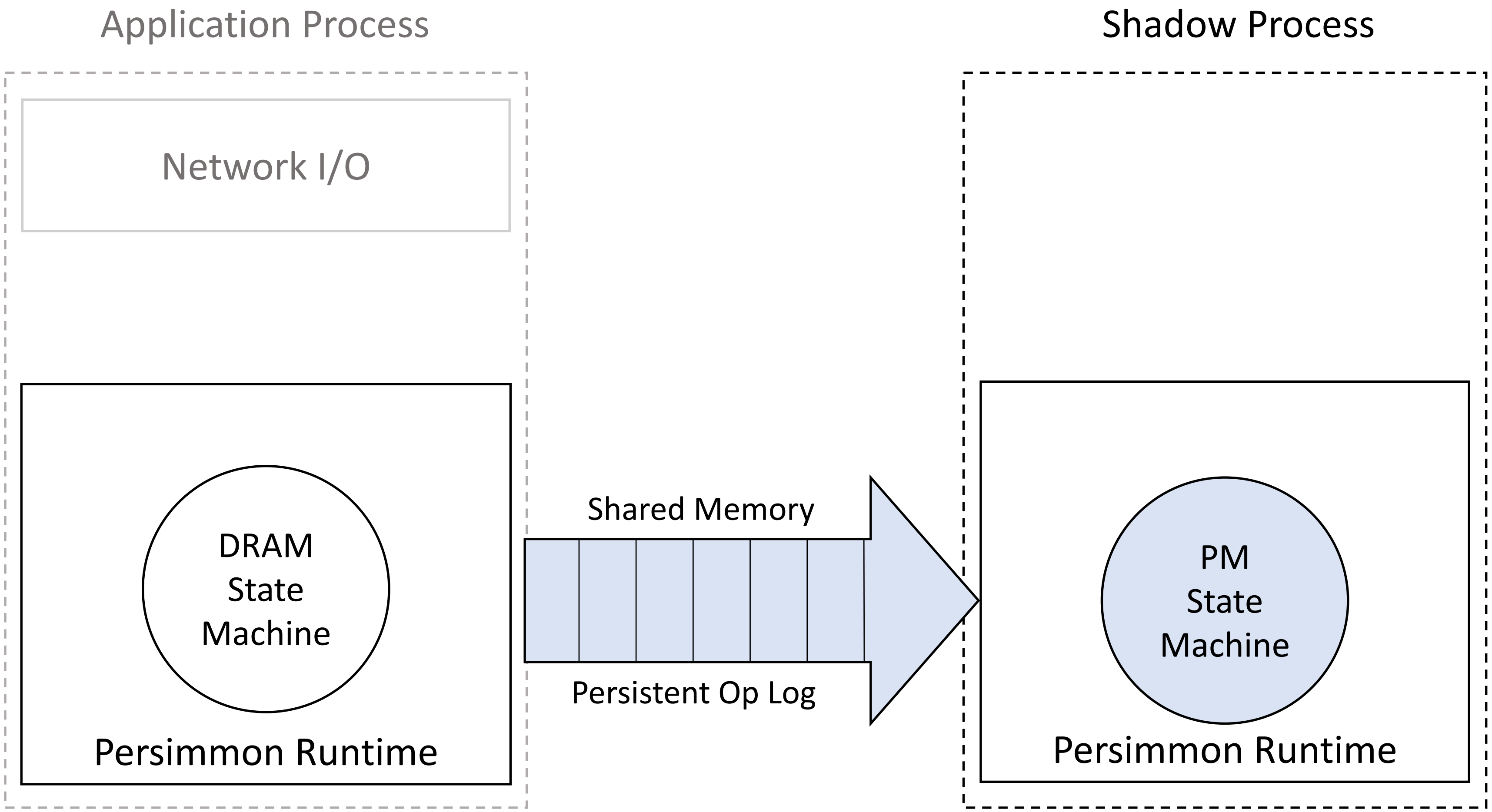
☹ Requires two CPU cores, 2x space.

☹ Shadow execution: throughput bottleneck?

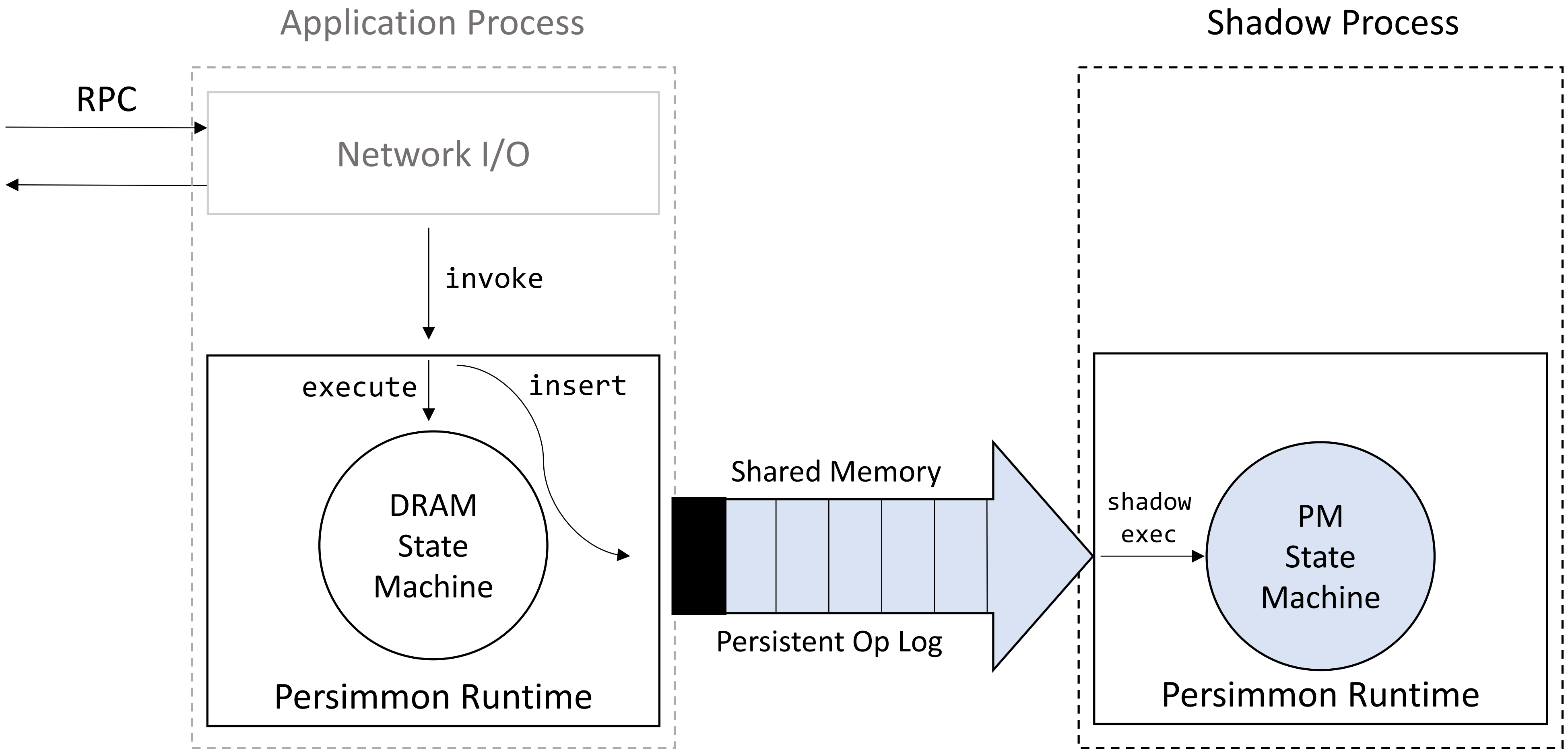
Outline

- **Background**: Challenges and key insight
- **Persimmon overview**: API and guarantees
- **Persimmon runtime**: Design and implementation
- **Evaluation**: Programming experience and performance

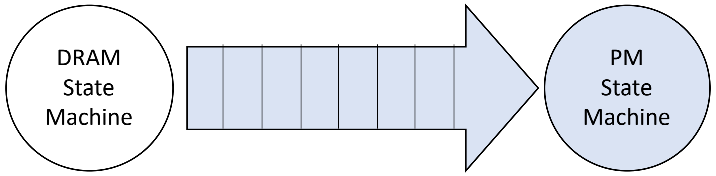
Persimmon runtime



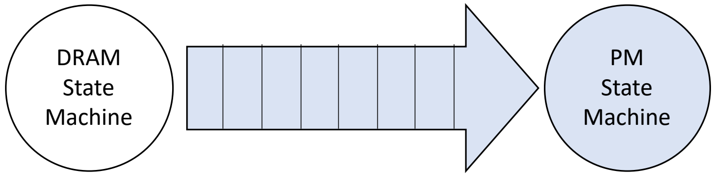
Persimmon runtime



Shadow execution

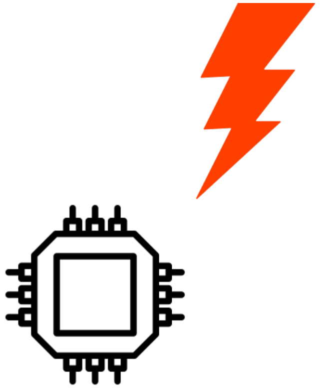


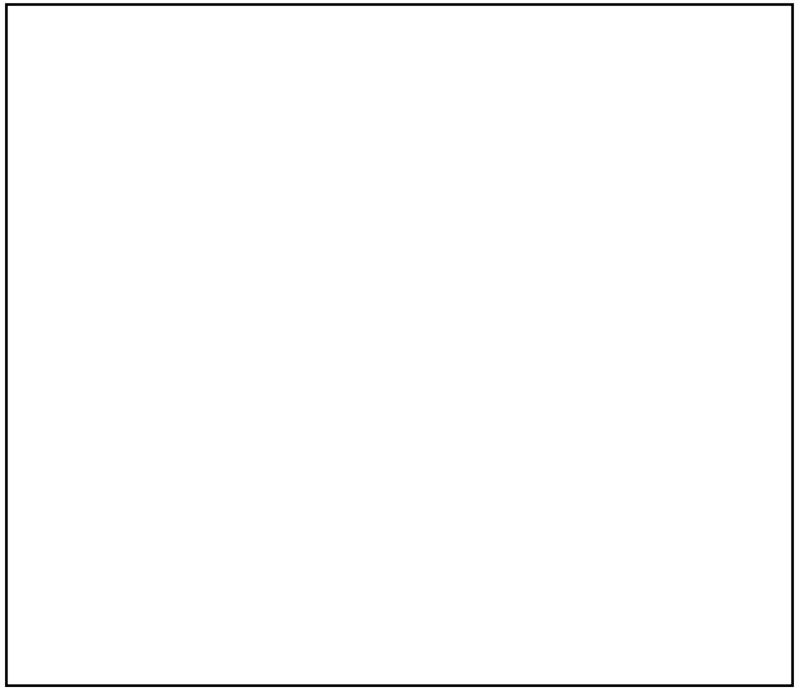
Shadow execution



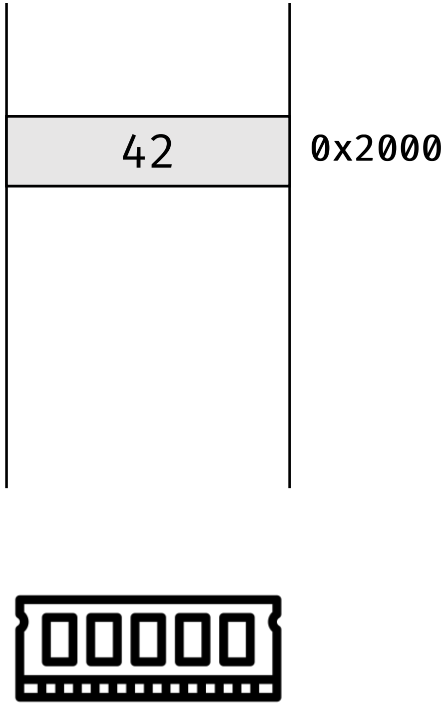
State machine operation

```
   ⋮
mov $42, 0x2000
   ⋮
```





CPU Caches

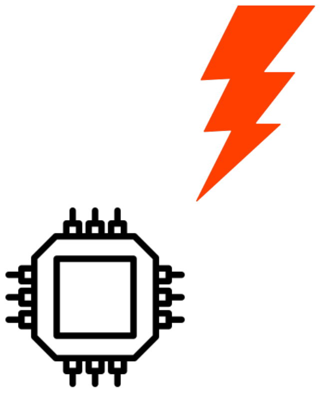


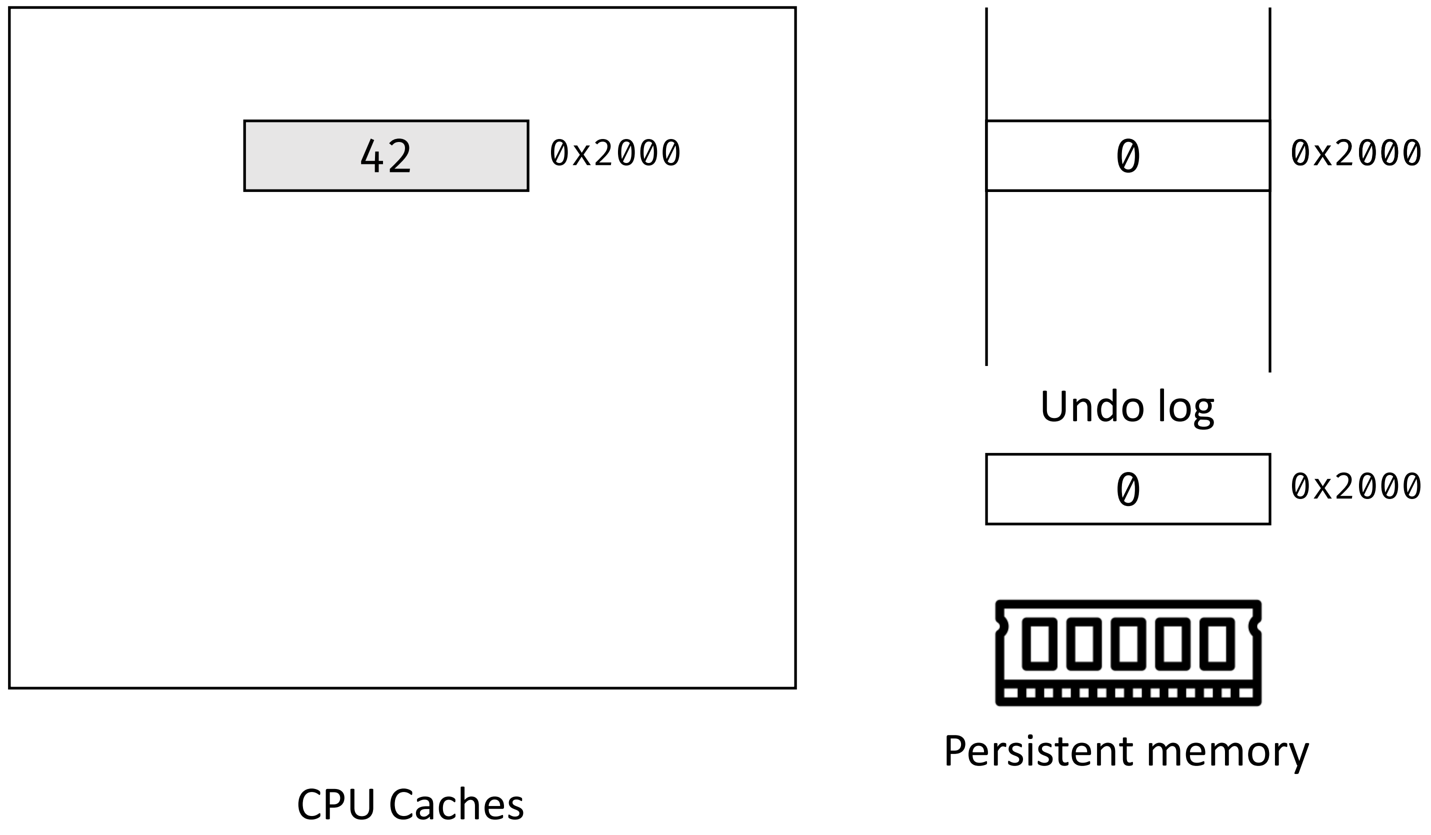
Persistent memory

Dynamic instrumentation for undo logging

State machine operation

```
⋮
(log 0x2000)
→ mov $42, 0x2000
⋮
```

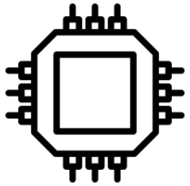


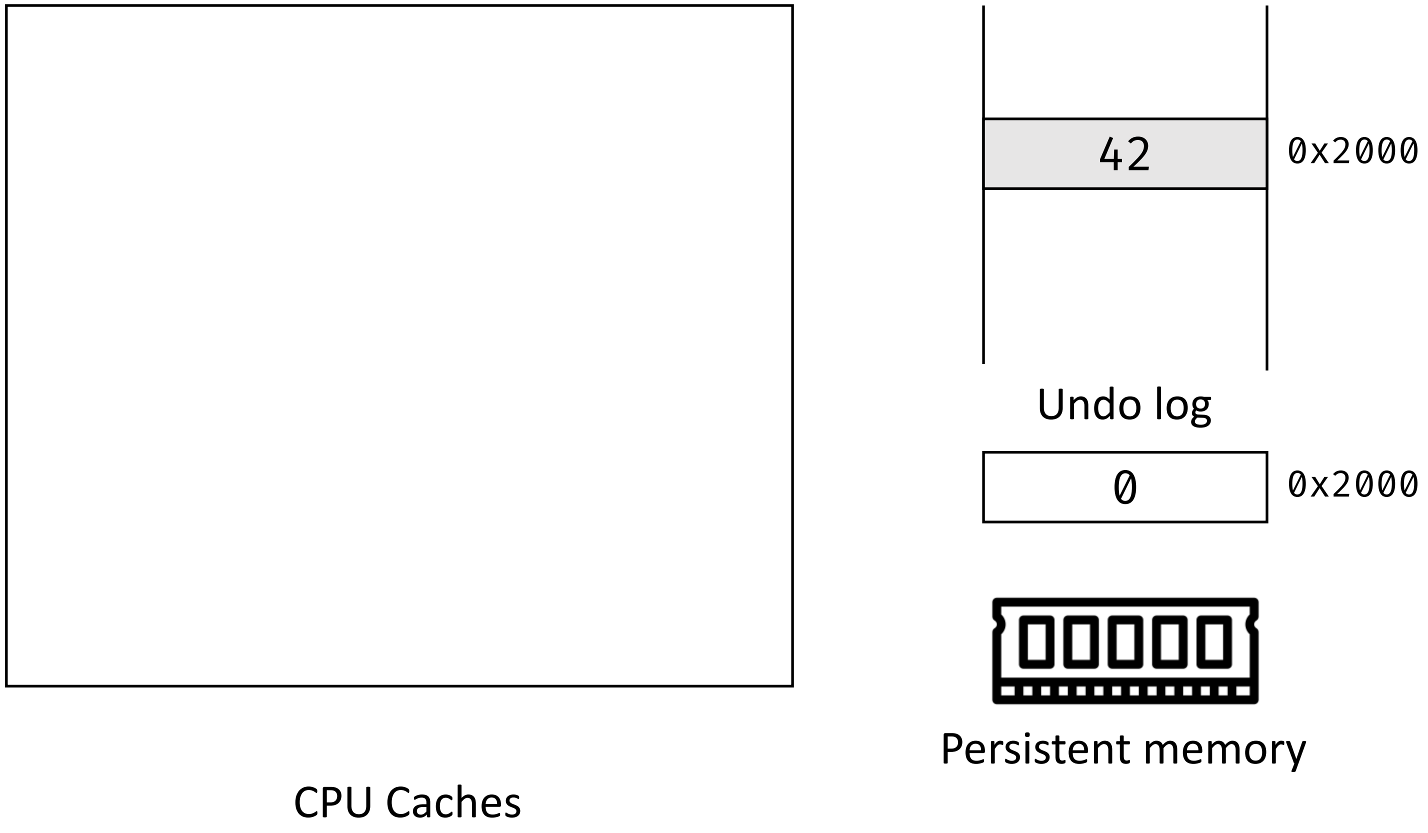


Recovery using the undo log

State machine operation

```
⋮
(log 0x2000)
mov $42, 0x2000
⋮
```

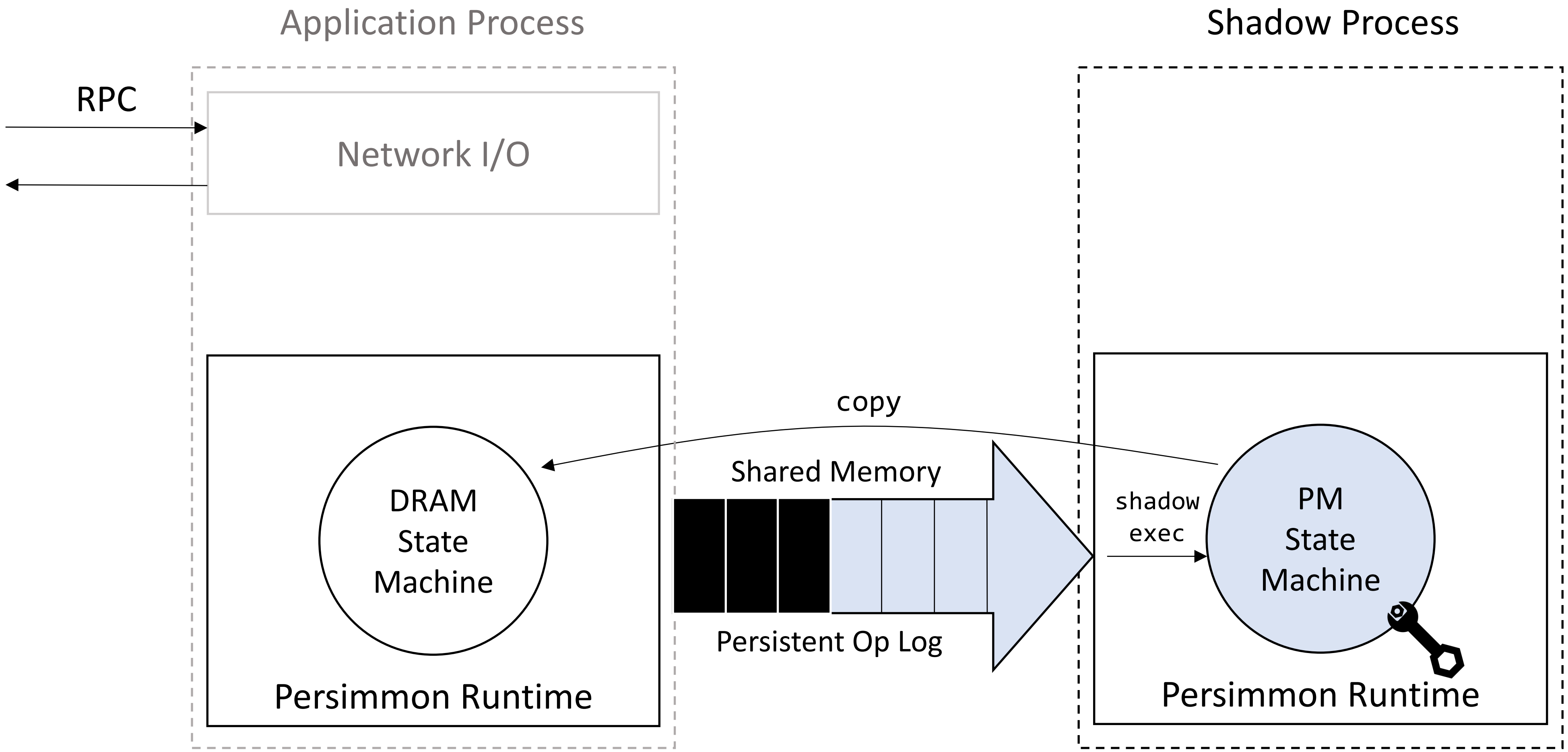




Optimizations for undo logging

- Undo-log in 32B blocks.
- De-duplication: log each block only once.
- …

Application crash recovery



Outline

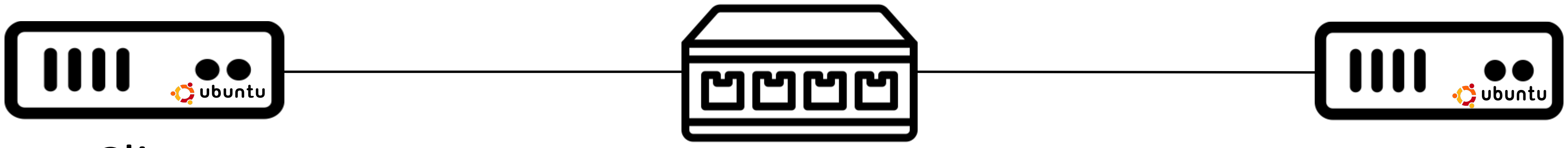
- **Background**: Challenges and key insight
- **Persimmon overview**: API and guarantees
- **Persimmon runtime**: Design and implementation
- **Evaluation**: Programming experience and performance

Persimmon requires little code modification

	Lines added / changed	
	Redis	TAPIR
Initialize Persimmon	7	10
Factor out state machine init.	36	34
Serialize state machine operation	26	12
Deserialize & execute operation	45	25
Check for read-only operations	1	1
Refactor for better performance	N/A	57
Total	115	139

Redis performance experiment setup

Arista 7060CX 100 Gbps



Client

- Mellanox CX-5 100 Gbps NICs
- 20-core dual-socket Xeon Silver

Server

- Mellanox CX-5 100 Gbps NICs
- 52-core dual-socket Intel Xeon Platinum
- 3TB of Intel Optane DC PMM (app direct)
- 768 GB of DRAM

redis

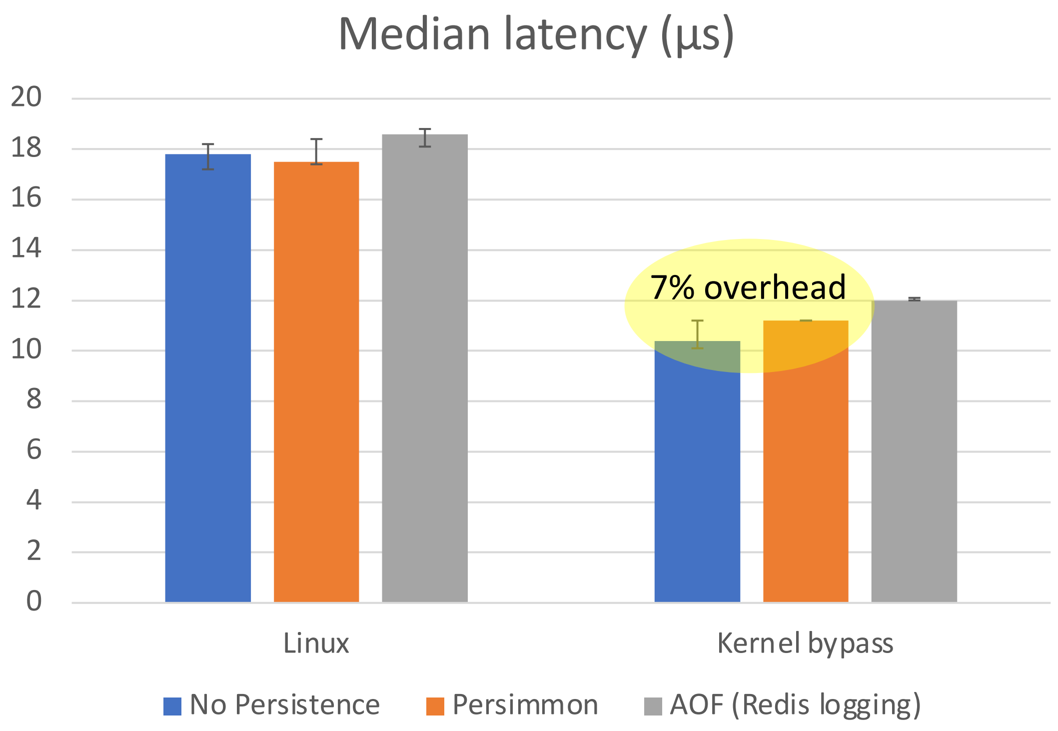
“Vanilla”

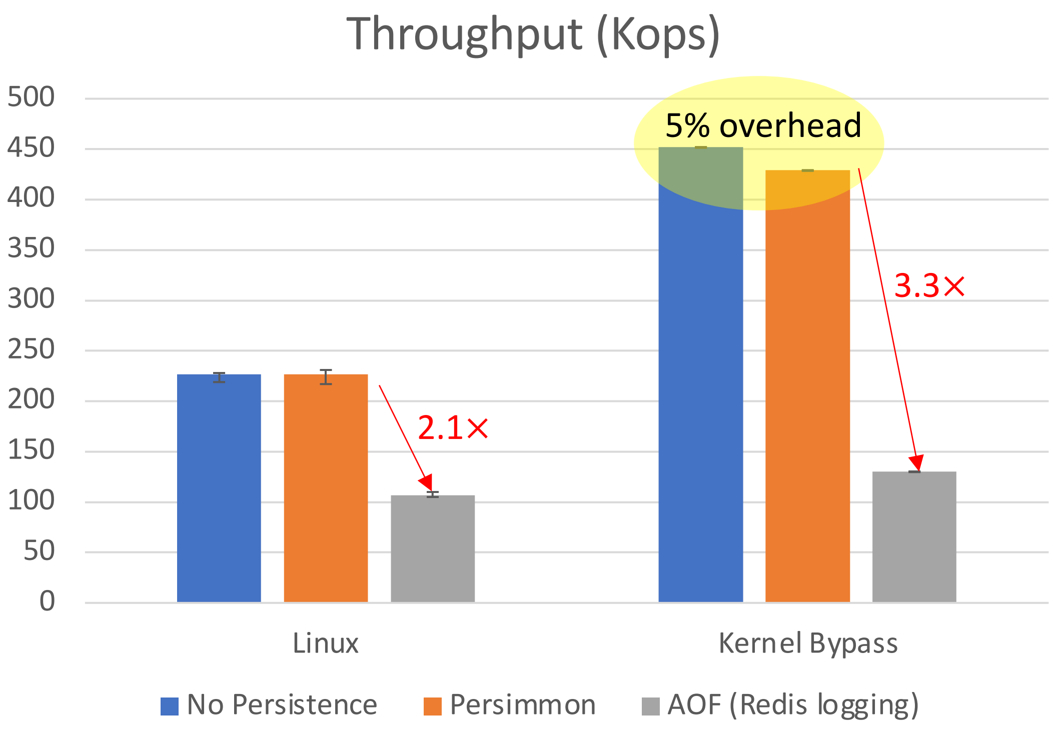
- Networking: Linux TCP
- Memory allocator: jemalloc

Kernel bypass

- Networking: DPDK UDP
- Memory allocator: Hoard

Redis is fast (and persistent) under Persimmon

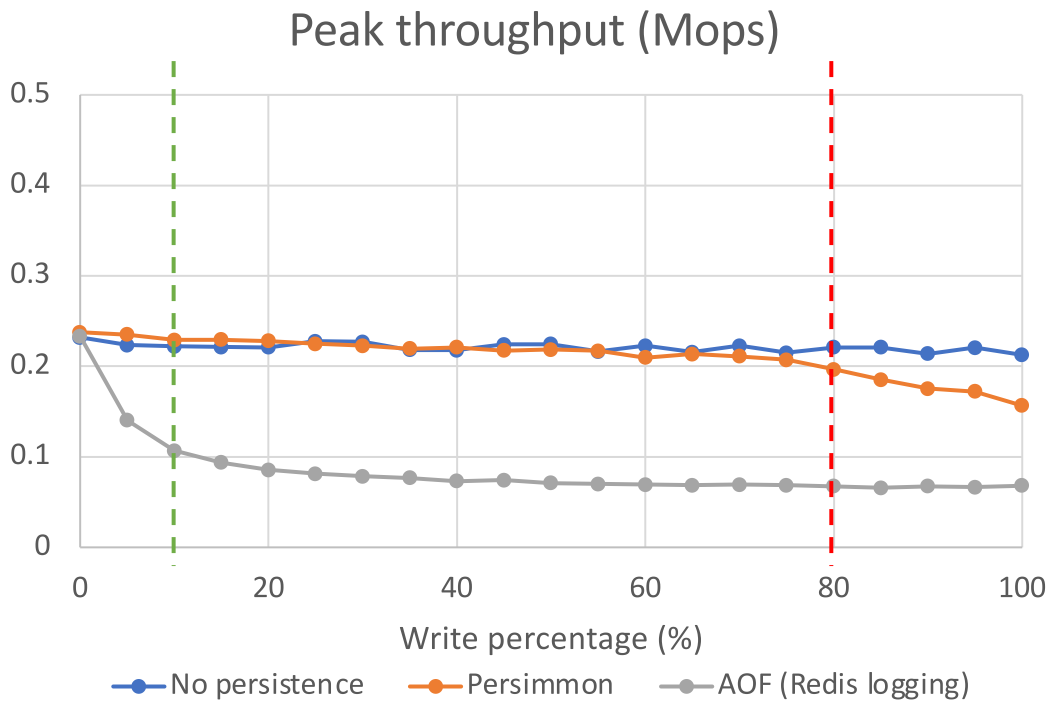




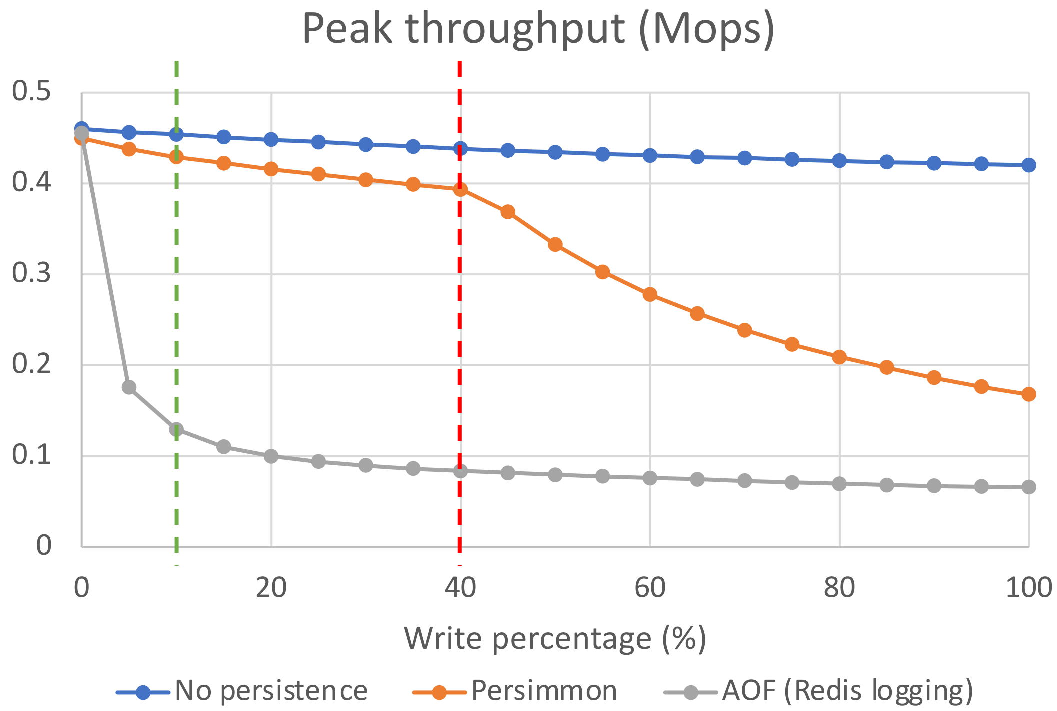
(Read/write workload, 10% writes, Zipf constant = 0.75, 130 million key-value pairs)

Persimmon performance depends on write percentage

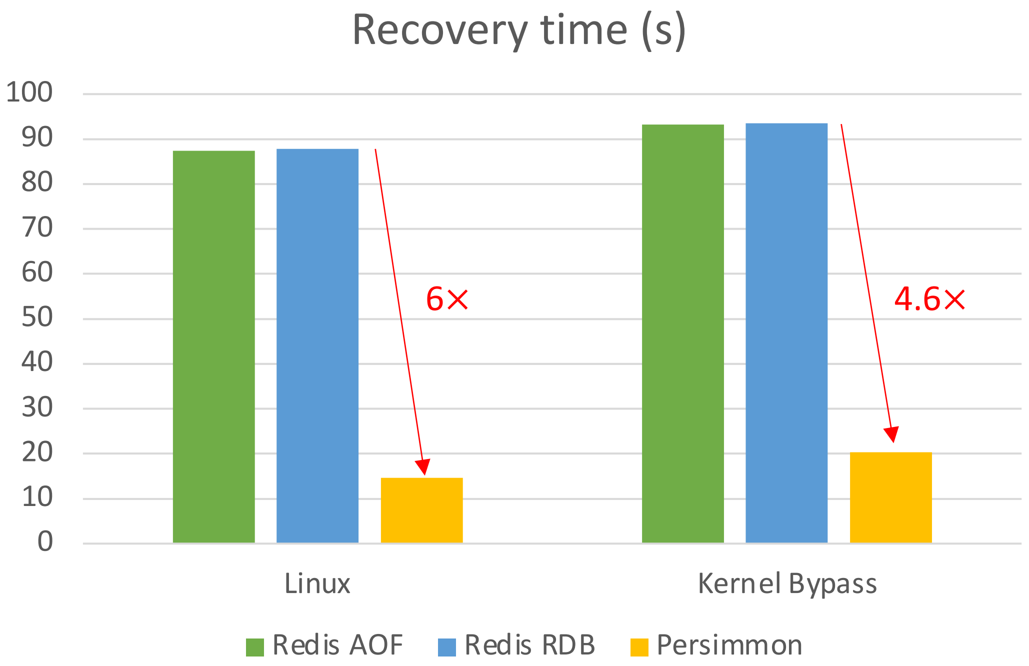
Redis on Linux

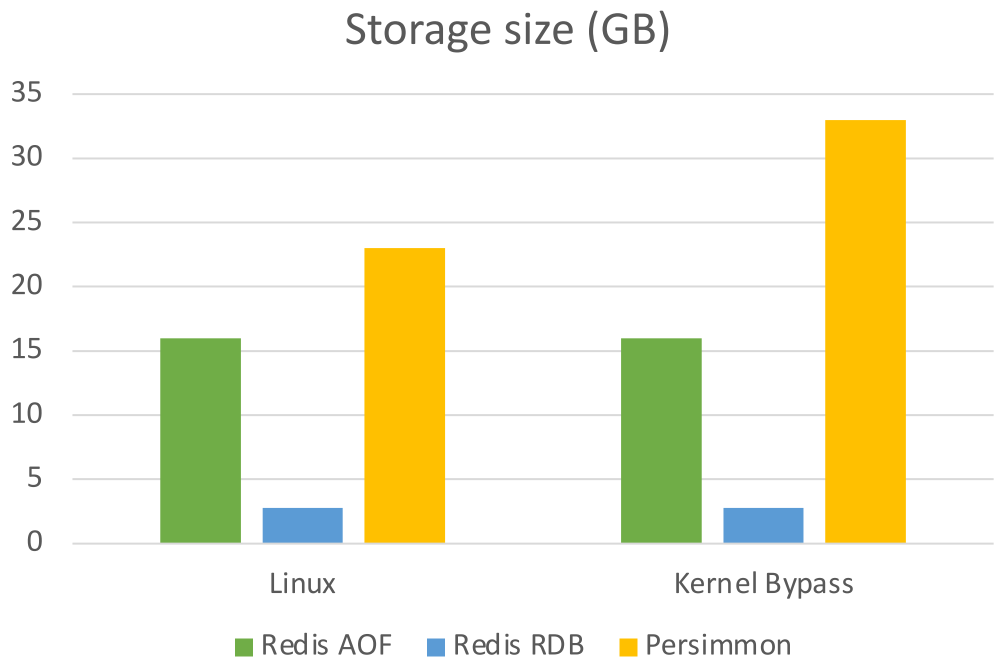


Redis with kernel bypass



Persimmon recovers quickly





(Read/write workload, 130 million key-value pairs)

Conclusion

- Persistent State Machines (PSM): a useful persistent memory abstraction for in-memory applications.
- Persimmon uses operation logging + shadow execution to achieve fast, low-effort persistence.
- Persimmon can persist Redis with ~100 LoC change and 5–7% performance overhead on a typical workload.

Thank you!

Wen Zhang <zhangwen@cs.berkeley.edu>