Optimistic Shared Memory Dependence Tracing

Yanyan Jiang[1], Du Li[2], Chang Xu[1], Xiaoxing Ma[1] and Jian Lu[1]

[1]Nanjing University
[2]Carnegie Mellon University

Understanding Non-determinism

- Concurrent programs are non-deterministic
  - transient behavior on specific interleaving/schedule[1]

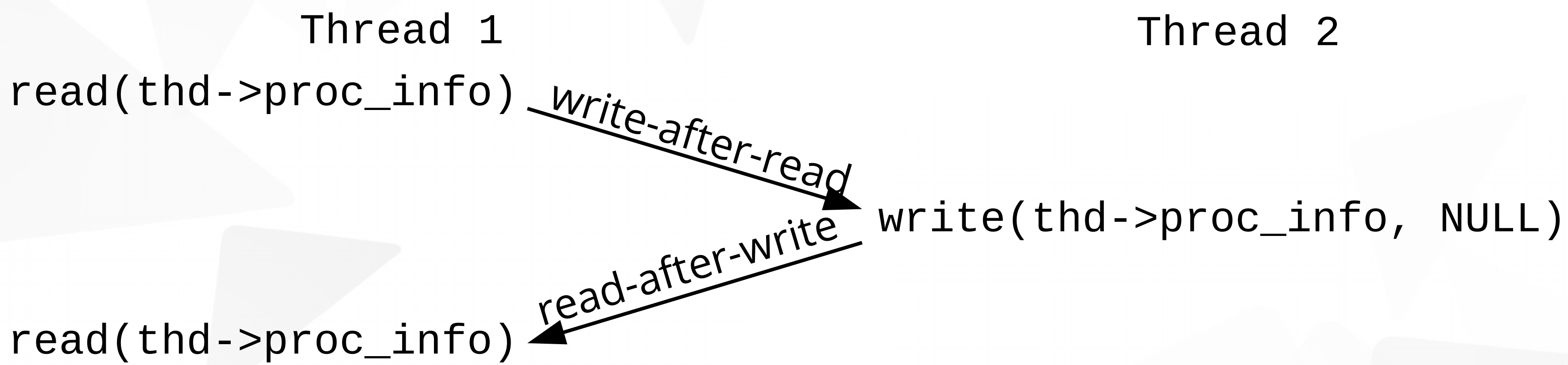
```
              Thread 1                              Thread 2
if (thd->proc_info) {
                                           thd->proc_info = NULL;
  fputs(thd->proc_info, ...);
```

Understanding order of shared memory accesses

[1] MySQL crashes due to null pointer dereferencing.

Shared Memory Dependences

- The order between consecutive accesses of a shared location



- Four types of shared memory dependences
  - read-after-read (RAR), read-after-write (RAW)
  - write-after-read (WAR), write-after-write (WAW)

Using Shared Memory Dependences

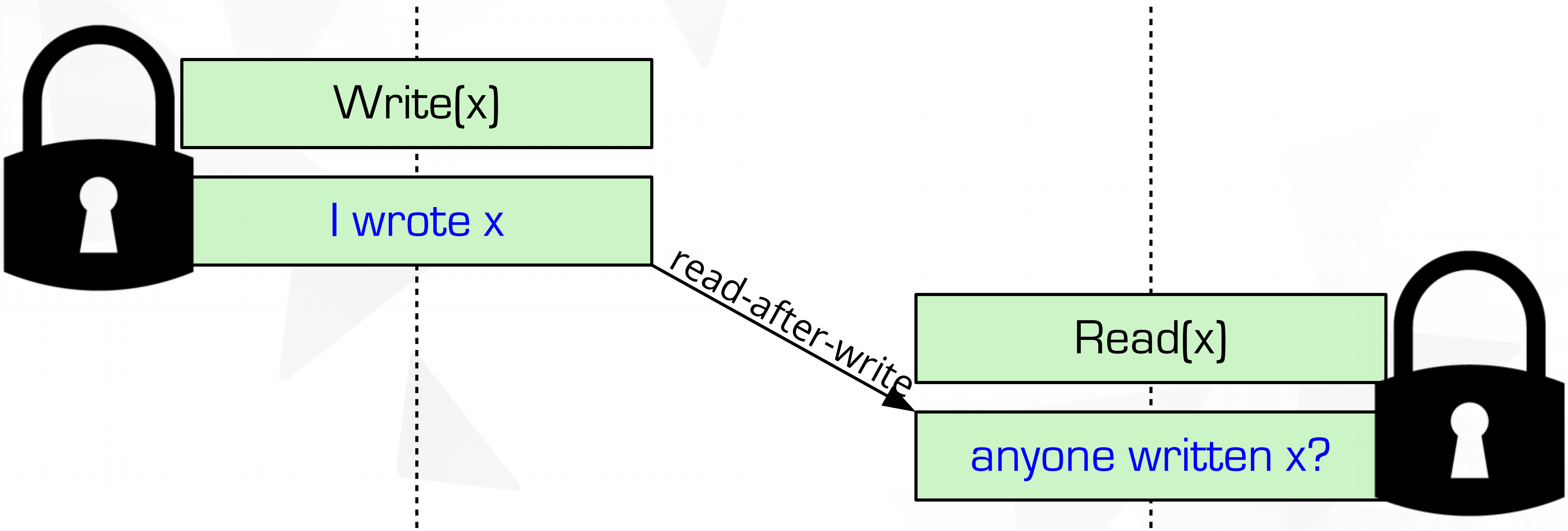
- Shared memory dependence in terms of replaying
  - RAW + WAR + WAW → trivial deterministic replay
  - RAW + WAW → $O(n)$ deterministic replay[1]
  - RAW only → $O(exp(n))$ deterministic replay[2]
- Predictive trace analyses
  - data race / atomicity violation detection

[1] Y Jiang, et al. CARE: Cache guided deterministic replay for concurrent Java programs. In *ICSE*, 2014.

[2] P Liu, et al. Light: Replay via tightly bounded recording. In *PLDI*, 2015.

Capturing Shared Memory Dependences

- Generally, we update metadata at shared memory accesses, and check it afterwards



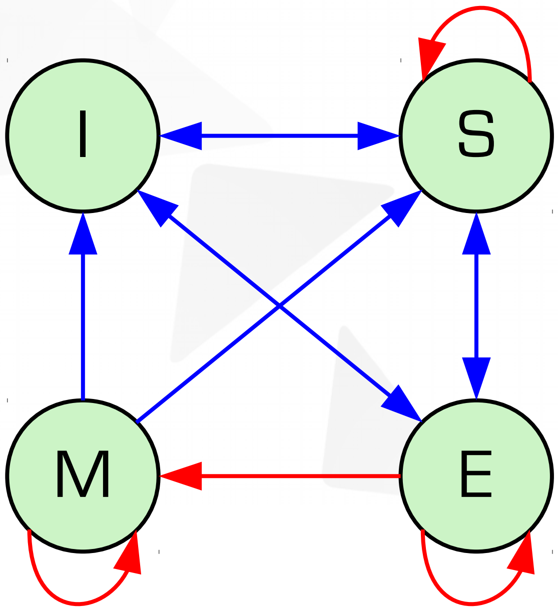
We must force atomic access!

Capturing Shared Memory Dependences: Overhead

- Program instrumentation inevitably brings overhead
  - scalable overhead: thread-local operations
    - metadata bookkeeping
    - extra call and branch instructions
  - non-scalable overhead: serialization
    - forcing atomicity is much more costly

Reducing Overhead: Exploiting Thread Locality

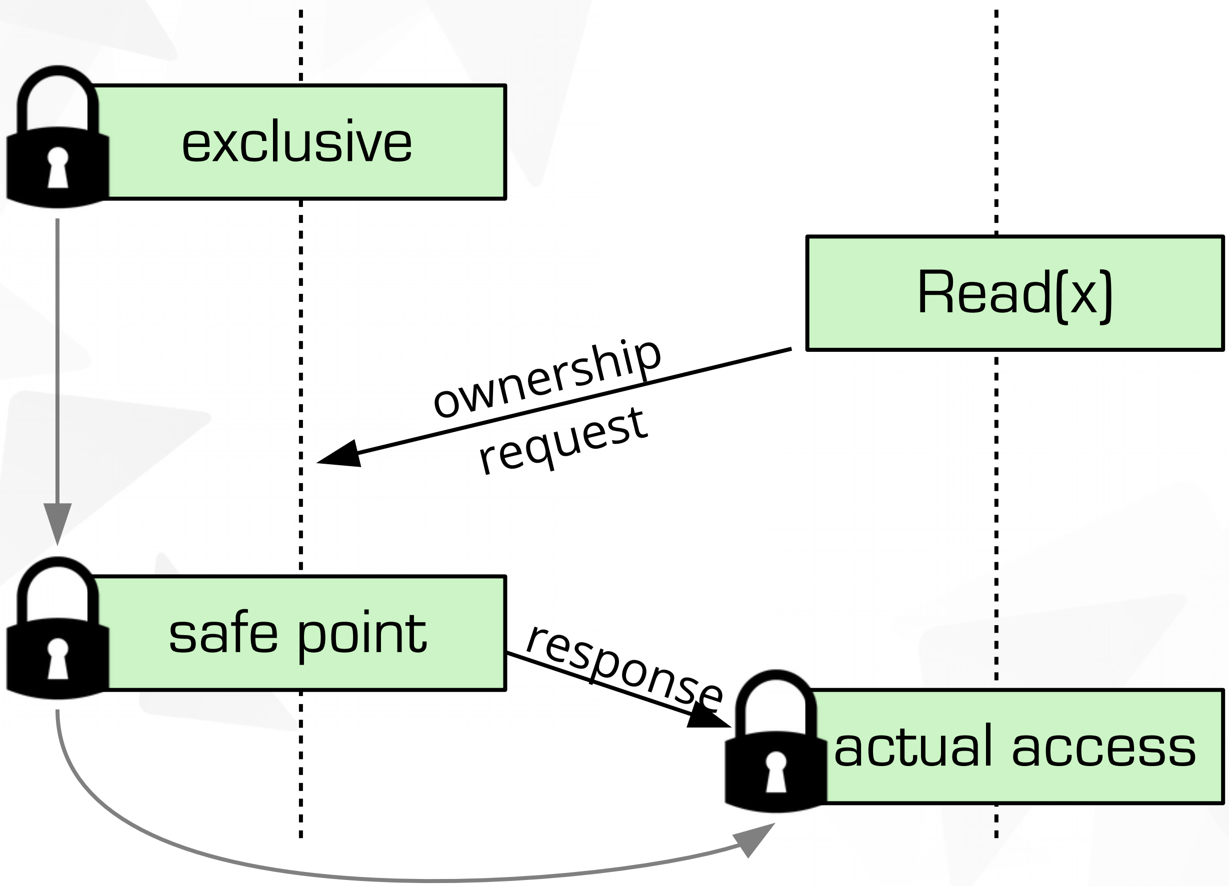
- A thread tends to have exclusive/shared access of a shared location for a consecutive time period
  - example: classical MESI cache coherence protocol



Fast path: exclusive/shared (CHEAP)
Slow path: bus traffic (EXPENSIVE)

Octet: Optimistic Tracing via Biased Locking[1]

- Eager acquisition, lazy release



[1] M D Bond, et al. Octet: Capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, 2013.

Octet: Drawback

- Biased lock is too optimistic
  - thread-local accesses are indeed fast, but
  - inter-thread coordination is too costly
    - under write-heavy workloads it is even worse than simple locking[1]

Can we simultaneously achieve
FAST fast paths and NOT-SLOW slow paths?

[1] M Cao, et al. Drinking from both glasses: adaptively combining pessimistic and optimistic synchronization for efficient parallel runtime support. In *WoDet*, 2014.

RWTrace: Overview of the Trade-off

- Not only thread locality, but also *reads dominate writes*
  - slower thread-local writes
  - faster cross-thread coordinations

	LEAP[1]	Octet	RWTrace
Thread-local Read	Slow	Fast	Fast
Shared Read	Slow	Fast	Fast
Inter-thread Read	Slow	Very Slow →	Slow
Thread-local Write	Slow	Fast →	Slow
Inter-thread Write	Slow	Very Slow →	Slow

[1] J Huang, et al. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

1 – Serializing Writes

- Use simple mutex lock to protect writes
  - each address is associated with a lock
  - write-after-write dependences can be captured

	Octet	RWTrace
Thread-local Read	Fast	Fast
Shared Read	Fast	Fast
Inter-thread Read	Very Slow	Slow
Thread-local Write	Fast	→ Slow
Inter-thread Write	Very Slow	→ Slow

1 – Serializing Writes: Performance

- Write-write data race is NOT expected
  - often leads to unexpected behaviors
  - developers eliminate them by synchronization
- Corollary: write-time locking scales as the non-instrumented program scales!
  - serializing all writes by a global lock does not hurt performance too much[1]

[1] J Zhou, et al. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, 2012.

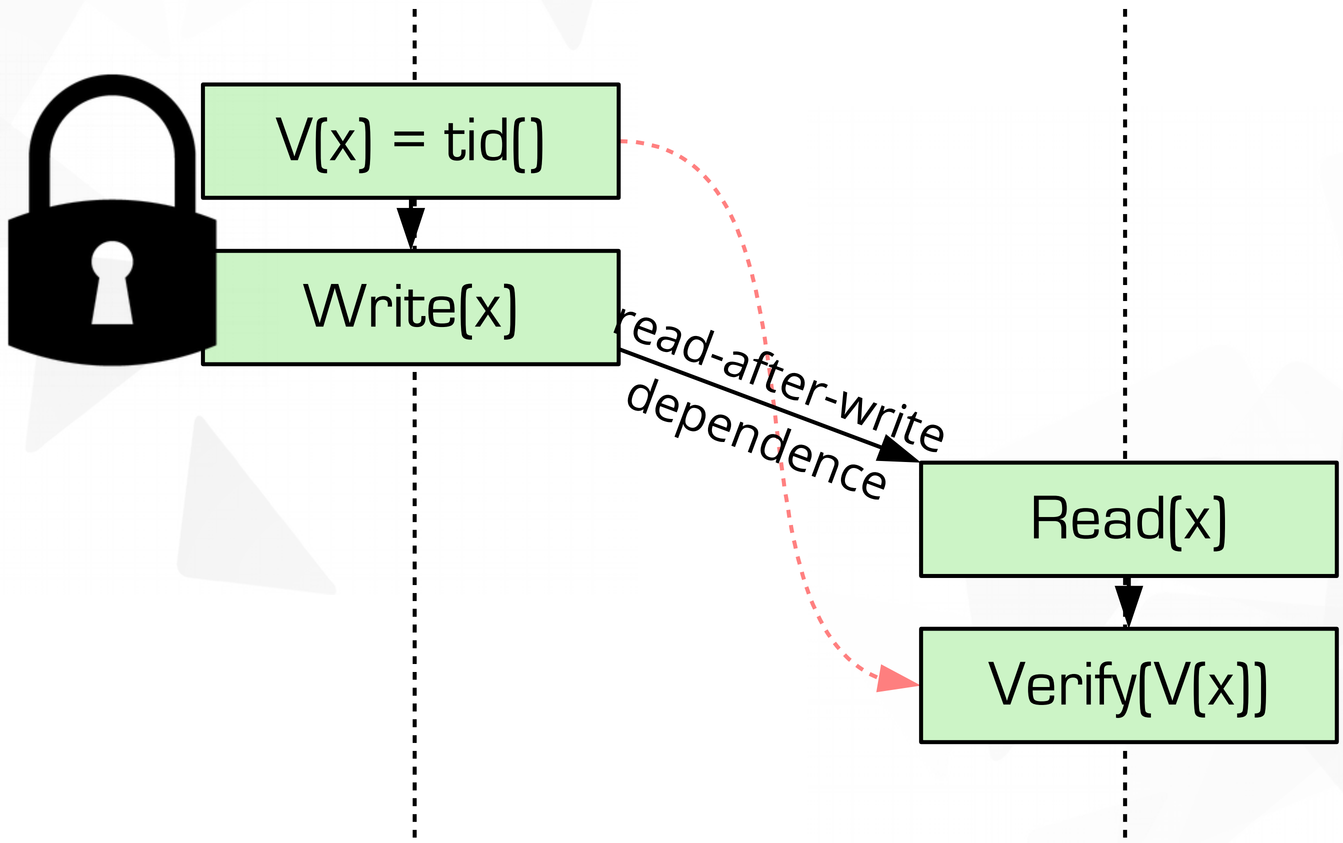
2 – Read Fast Path

- The vast majority, as fast as possible
  - $O(1)$, zero synchronization (wait-free), scalable
  - tests whether a thread is reading a value from previously unknown source

	Octet	RWTrace
Thread-local Read	Fast	Fast
Shared Read	Fast	Fast
Inter-thread Read	Very Slow	Slow
Thread-local Write	Fast	Slow
Inter-thread Write	Very Slow	Slow

2 – Read Fast Path: Am I Thread-local?

- Single-sided error: may unnecessarily fall back to slow path, but never miss any read-after-write dependence

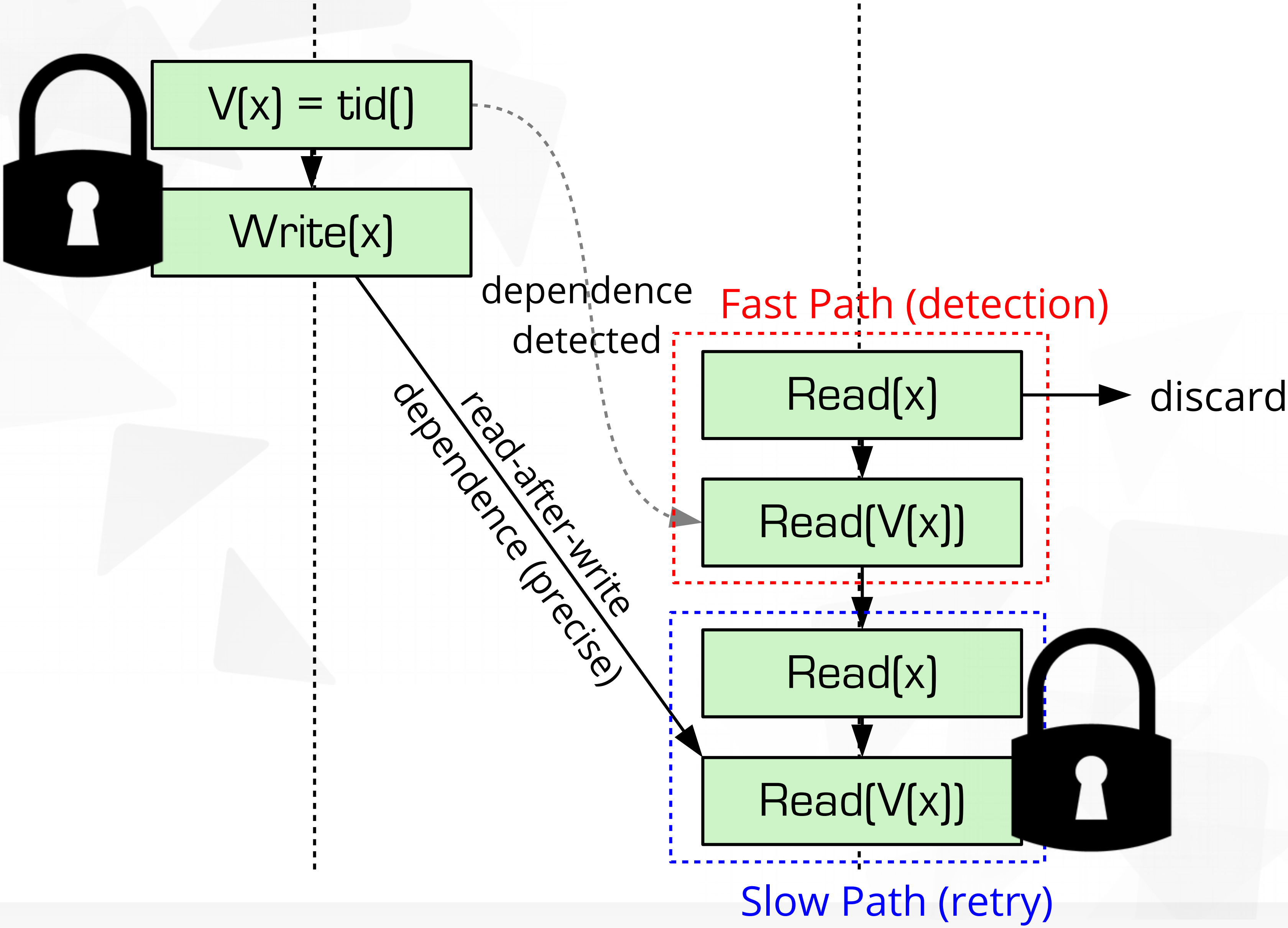


3 – Read Slow Path

- Read fast path fails → fall back to slow path
  - ignore the previously read value
  - read again with lock (exact read-after-write dependence)

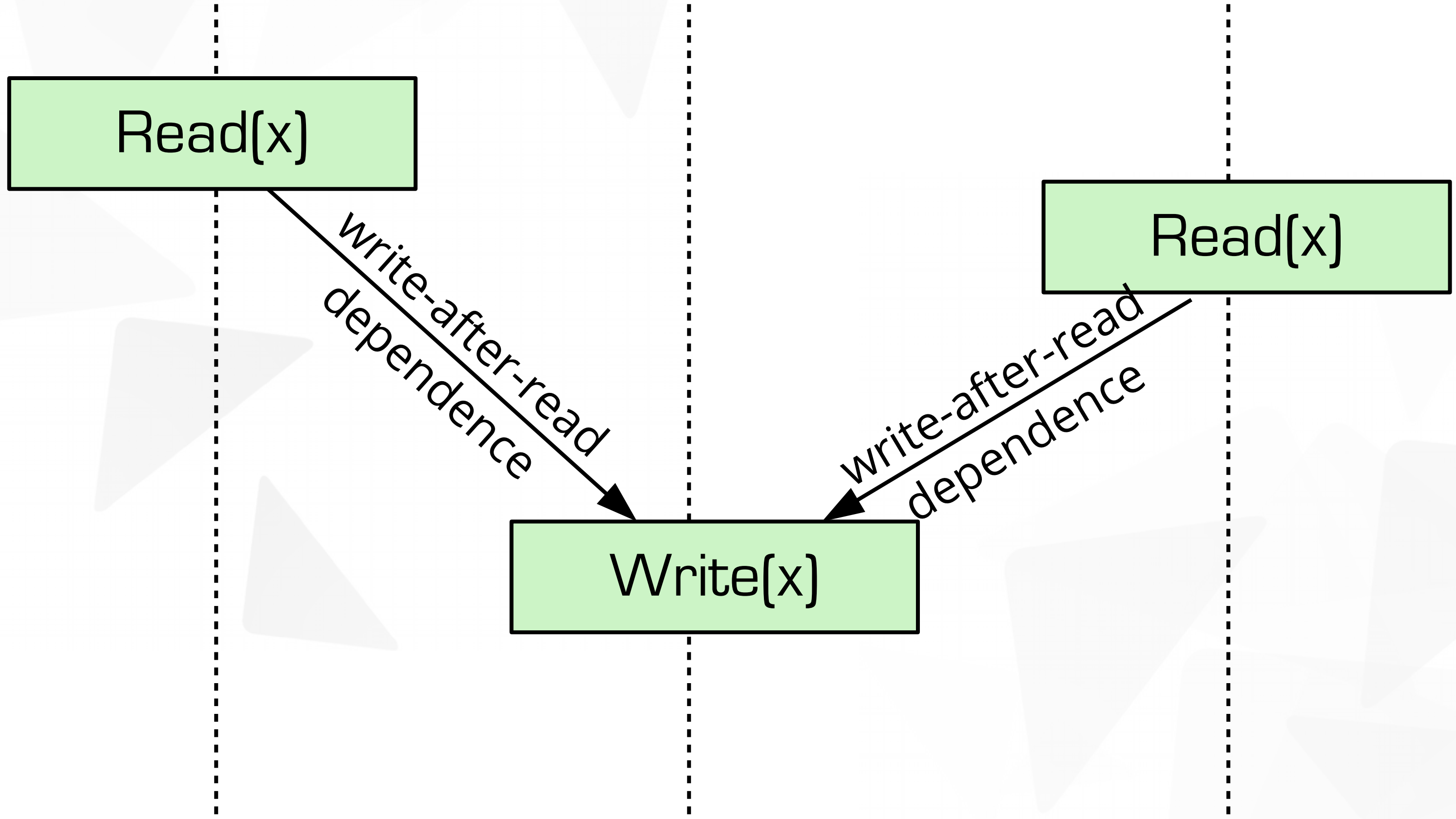
	Octet	RWTrace
Thread-local Read	Fast	Fast
Shared Read	Fast	Fast
Inter-thread Read	Very Slow →	Slow
Thread-local Write	Fast	Slow
Inter-thread Write	Very Slow	Slow

3 – Read Slow Path: Retry



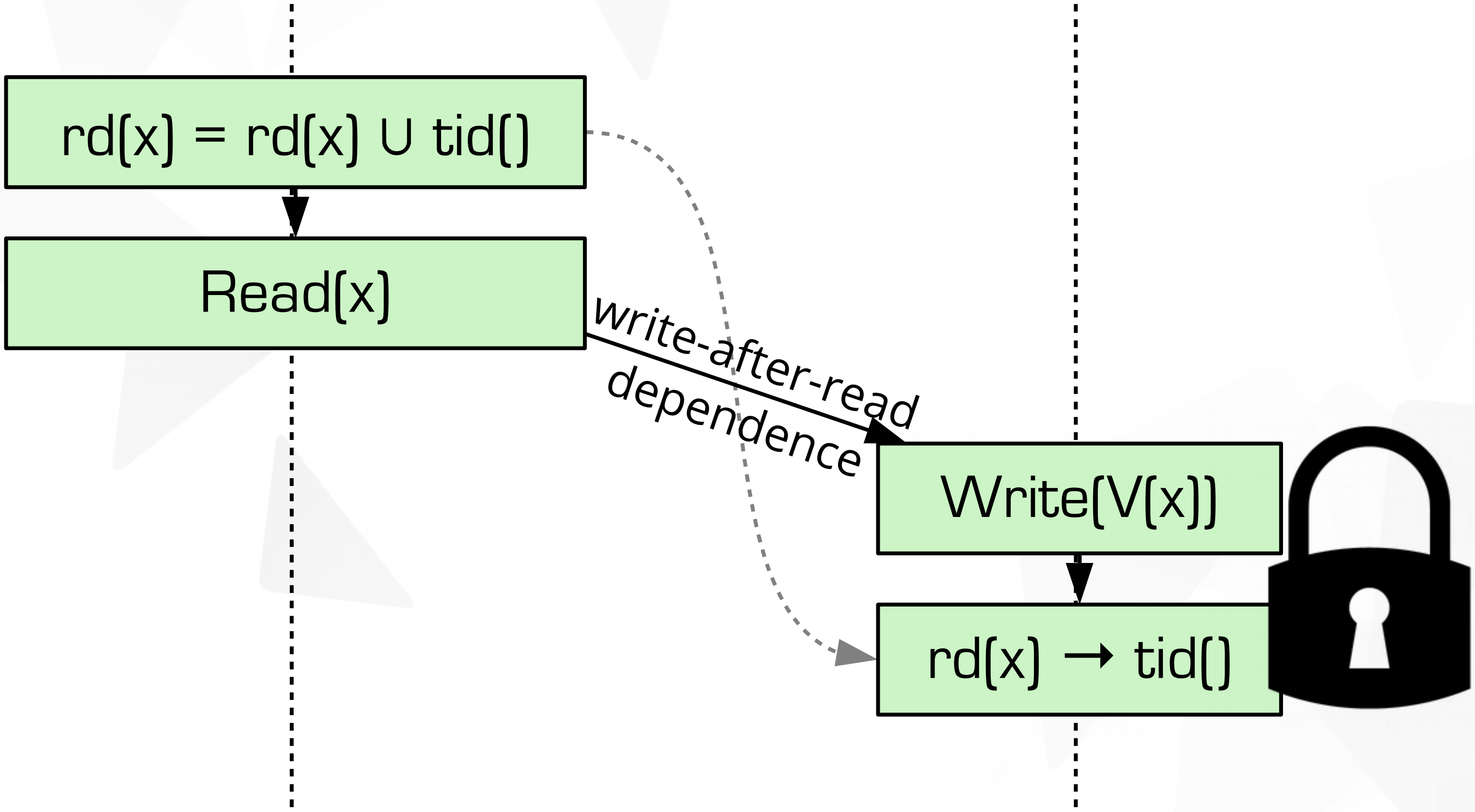
4 - Write-after-read Dependences

- Write-after-read dependences are many-to-one



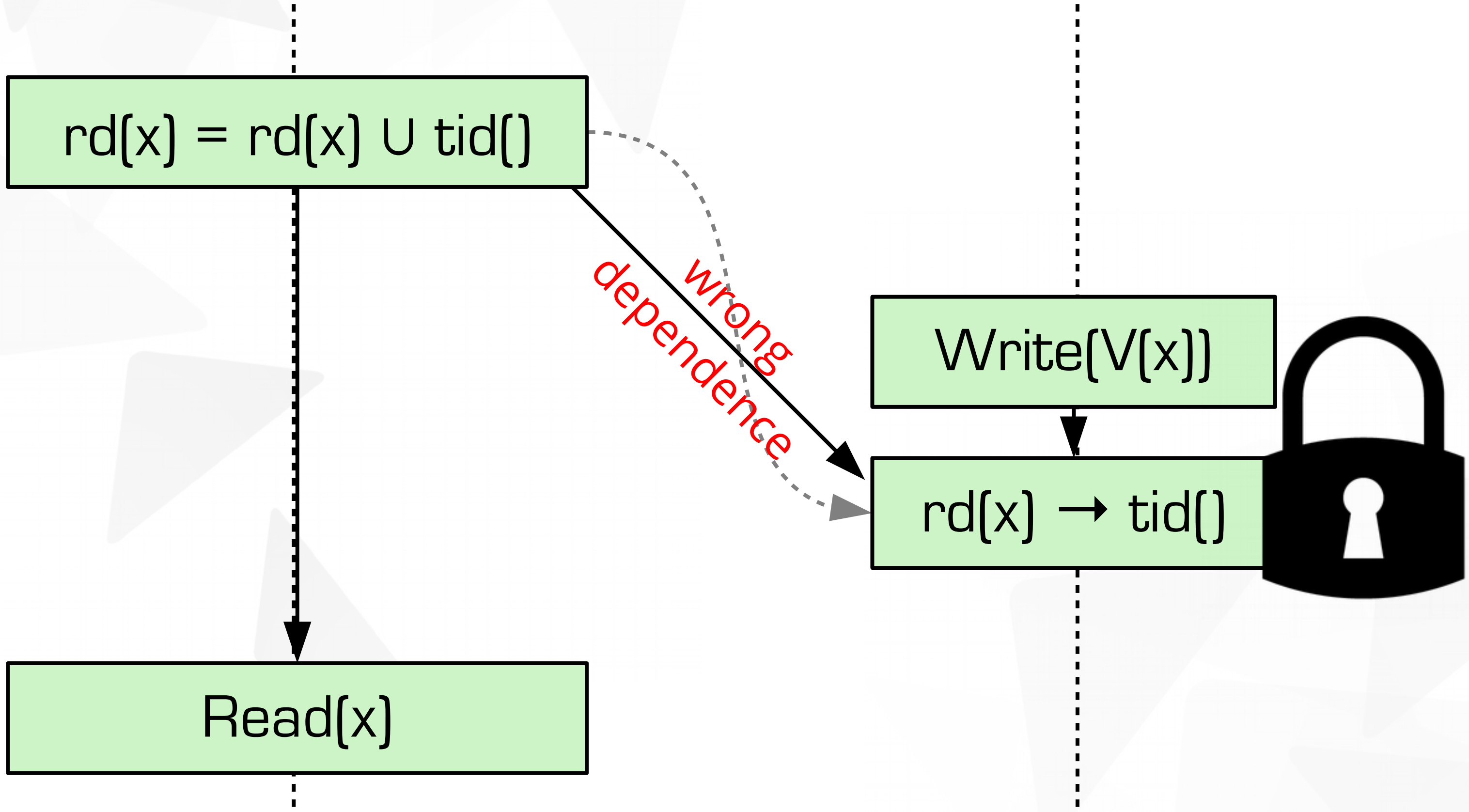
4 - Write-after-read Dependences: The Memory Order Trick Again, Inversely

- Maintain a read set to preserve wait-free read fast path
  - no extra synchronization, still scalable

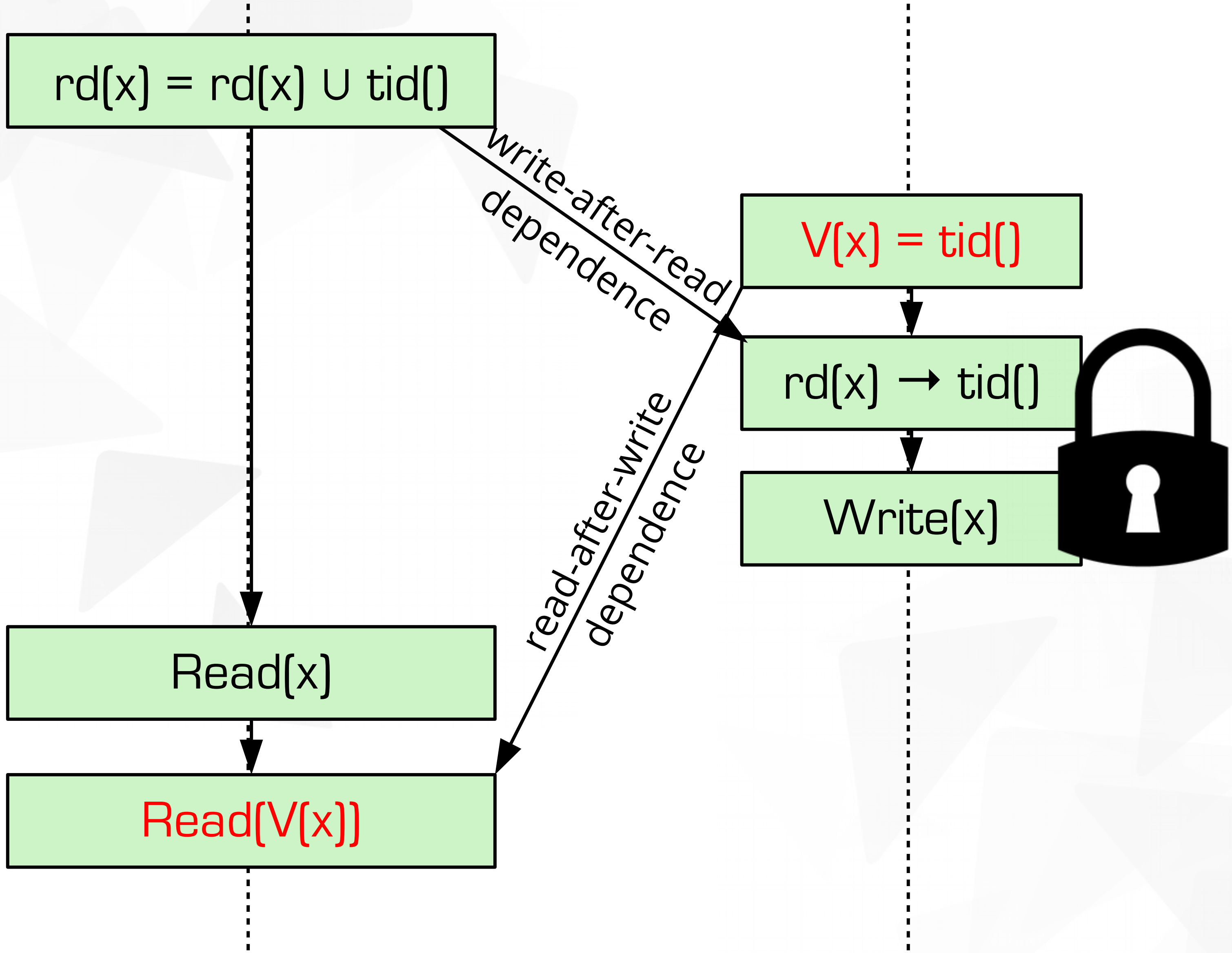


4 - Write-after-read Dependences: Wait-free Thread-local Read Fast Path

- There can be tricky memory orderings

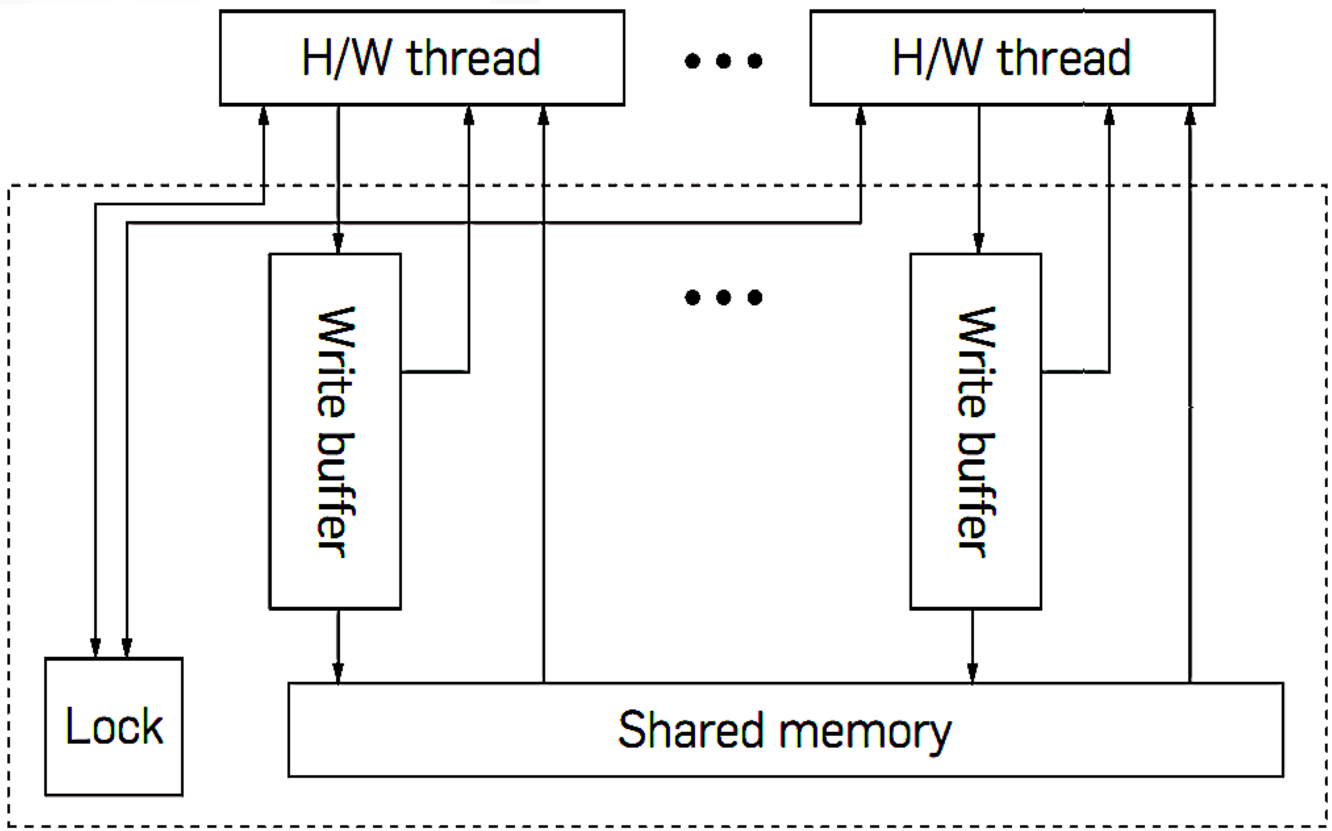


4 - Write-after-read Dependences: Handling Self-loop



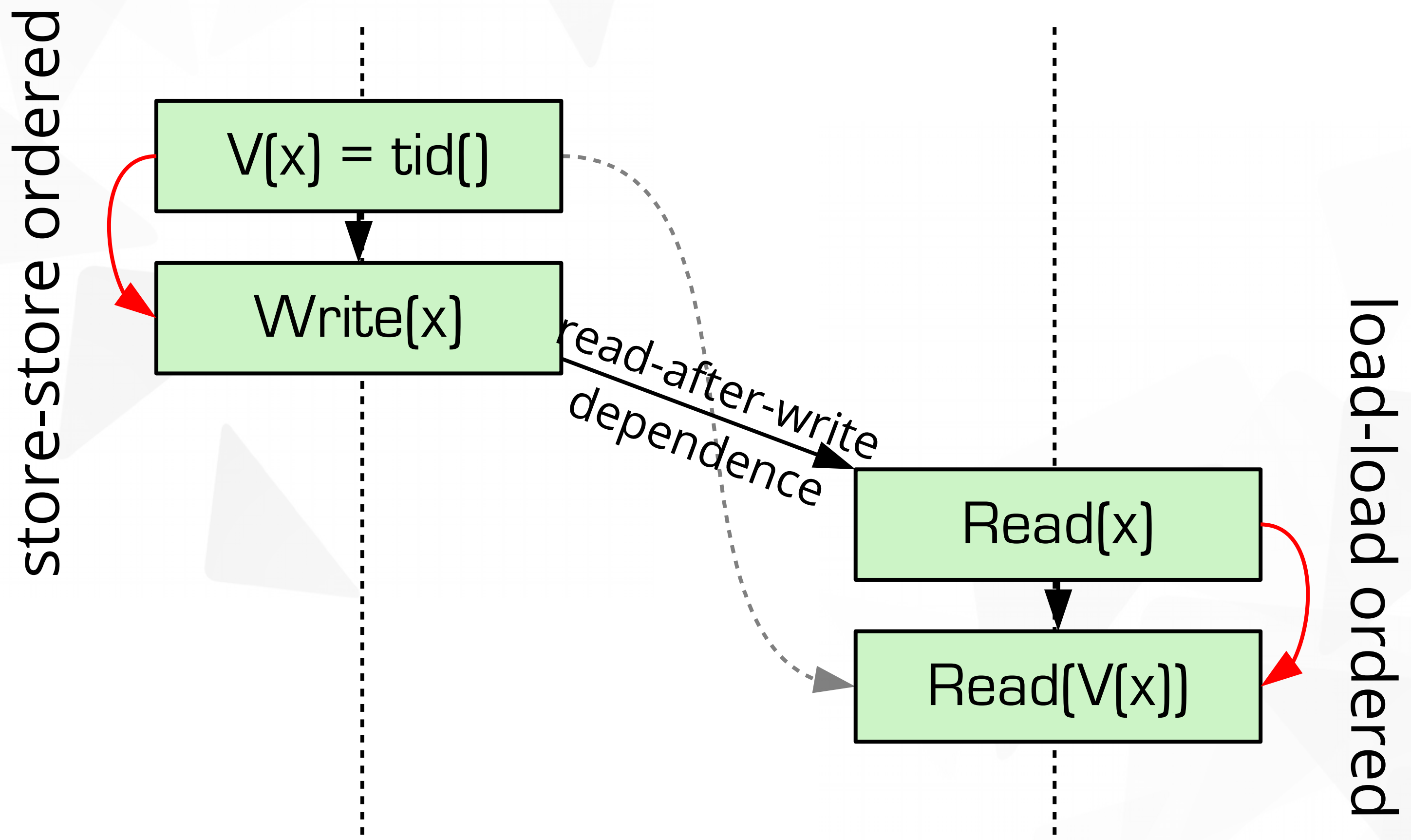
5 - Relaxed Memory Model

- RAW dependence tracing is barrier-free for x86-TSO
  - fences required for weaker memory models



5 – Relaxed Memory Mode: Memory Ordering on x86-TSO

- Barrier-free for read-after-write dependences
- MFENCE for write-after-read dependences

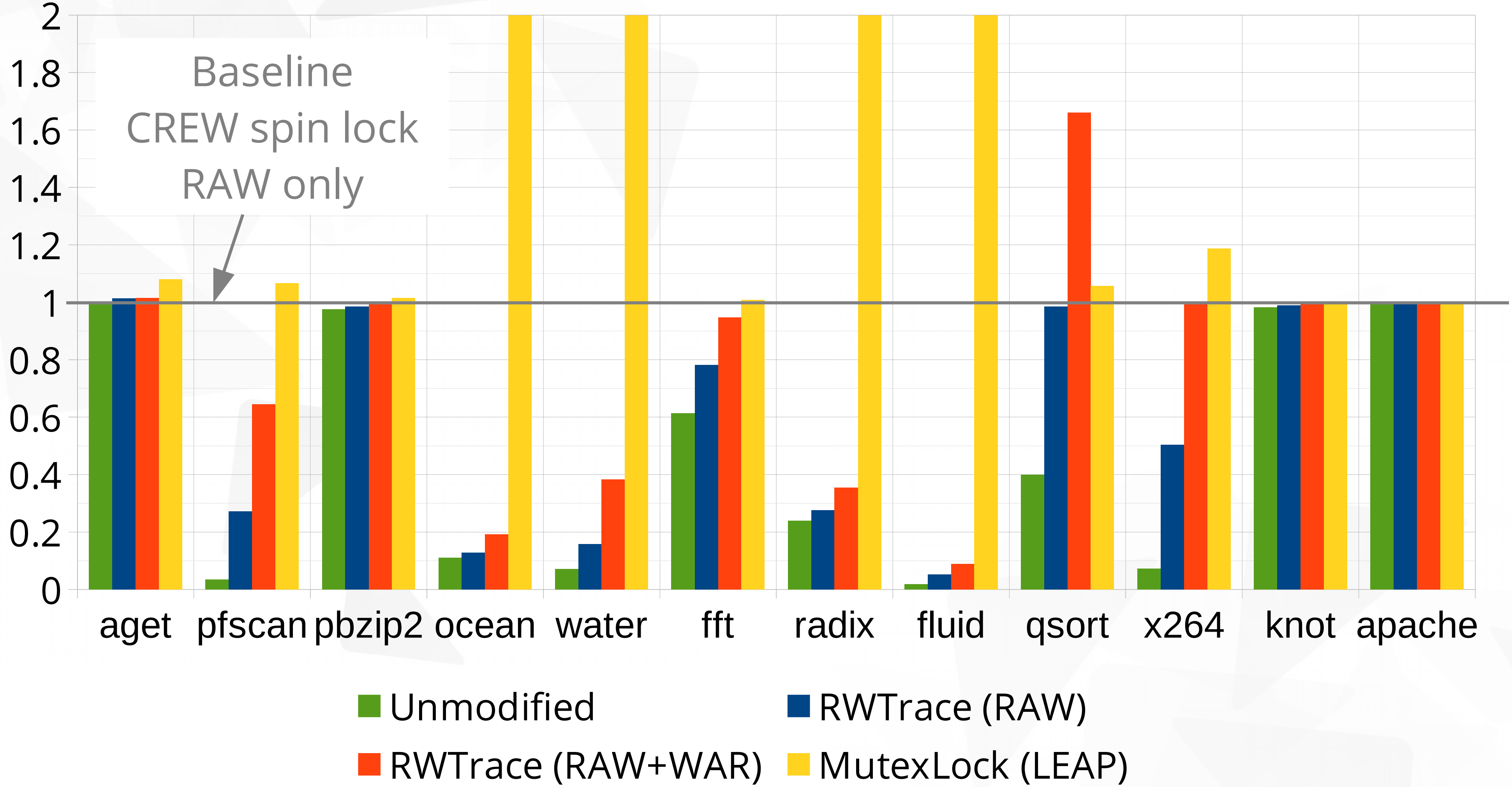


RWTrace: Optimistic Shared Memory Dependence Tracing

- Technical highlights
  - precise WAW, RAW, WAR dependences tracing
  - wait-free thread-local read fast path
    - barrier-free on x86-TSO
  - scalable program instrumentation
- Implementation
  - upon LLVM, open source[1]
  - as our experimental platform (like Octet)

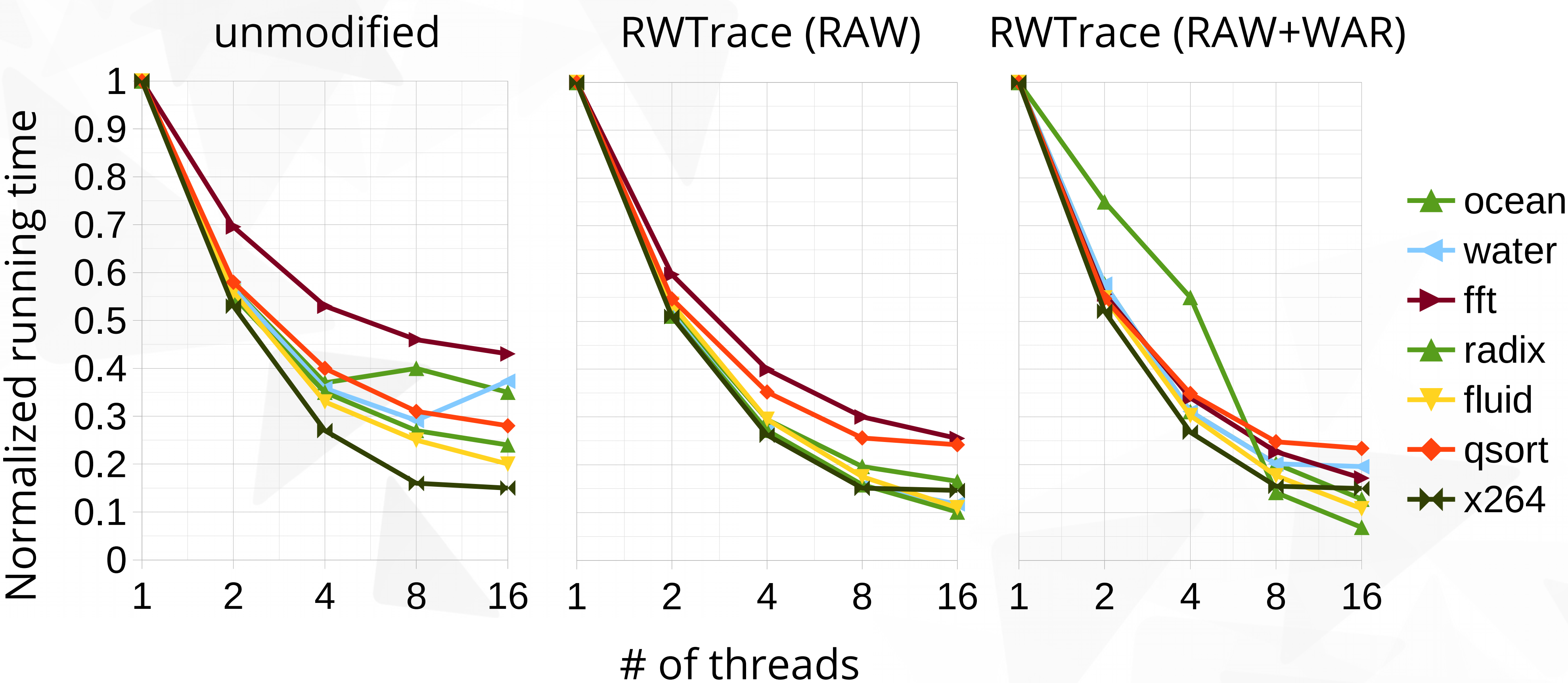
[1] http://github.com/jiangyy/rwtrace

Evaluation Results[1]: Overhead



[1] All experiments conducted on a 4×6-core Xeon machine.

Evaluation Results: Scalability



Shared Memory Dependences: In Pursuit of Determinism

- Dynamic analysis of concurrent systems
  - deterministic replay, data race / atomicity violation / false sharing detection, etc.
  - shared memory dependence reduction
  - theoretical aspects
- Non-determinism control
  - software transactional memory
  - deterministic multi-threading

Thank You!