Lock Shielding:
A General Technique for Misuse-Resilient Locks

Vivek Shahare [1] Milind Chabbi [2,1] Nikhil Hegde[1]

[1] Indian Institute of Technology Dharwad, India

[2] Programming Systems Group, Uber Technologies Inc., USA

ICS'26 Belfast, UK July 7, 2026

Locks

- Provide mutual exclusion for shared data
- Most popular mutual-exclusion primitive
- Common usage:

```
pthread_mutex_t m;

m.Lock()

Critical Section (CS)

m.Unlock()
```

Our Focus: Lock Misuse

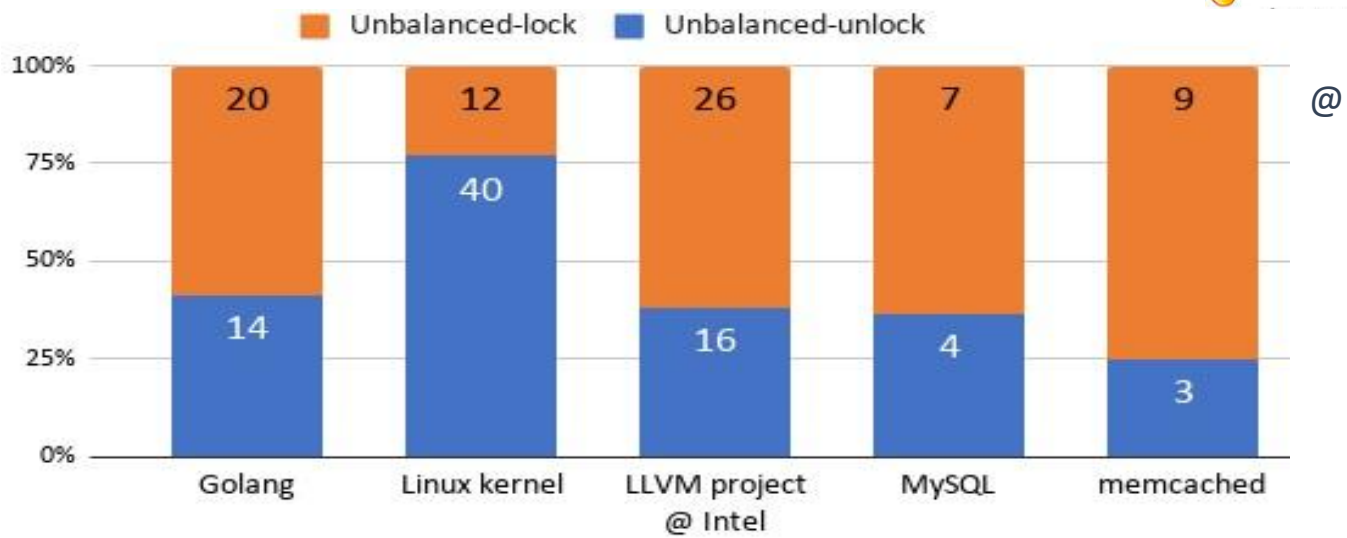
Unbalanced-lock:
forgetting to call `unlock`

```
if cond {
        m.Lock()
        ...
    return;
}
m.unlock()
```

Unbalanced-unlock:
call `unlock` without `lock`

```
if cond {
        m.Lock()
}


m.Unlock()
```



@

@Vivek Shahare, Milind Chabbi, and Nikhil Hegde. Protecting Locks Against Unbalanced Unlock(). *SPAA '23*

Lock Misuses - Impact

Unbalanced-Lock

Potential Impact:

- Starvation?
- Deadlock?
- Benign? - E.g. reentrant locks

Unbalanced-Unlock

Potential Impact:

- Mutex violation?
- Starvation?
- Corruption of lock internals?
- Program corruption?
- Benign?

Can we:

- detect both misuses?
- devise algorithms to avoid these situations?

Support for Handling Misuse in Production Codes

Language Construct	Handles unbalanced-lock?	Handles unbalanced-unlock?
pthread `pthread_mutex_t`	✗ (deadlocks)	✗ (undefined behavior)
pthread lock with `PTHREAD_MUTEX_RECURSIVE`	✓	✗ (undefined behavior)
pthread lock with `PTHREAD_MUTEX_ERRORCHECK`	✓	✓ (avoids)
C++11 `lock_guard <std::mutex>`	✗ (deadlocks)	✓ (avoids)
C++11 `lock_guard <std::recursive_mutex>`	✓	✓ (avoids)
C++11 `std::mutex` with explicit `lock()` / `unlock()`	✗ (deadlocks)	✗ (undefined behavior)
C++11 `std::recursive_mutex` with lock()/unlock()	✓	✗ (undefined behavior)

Baseline version **Recursive Version** **RAII Model/Error Check**

- OMP, Java Synchronized, Boost Mutex

Key Contributions - Lock Shielding

Provide Lock Misuse Resiliency

Lock Shielding (LS): a general technique for lock misuse resiliency handling both types of misuses.

Make Locks Reentrant

LS turns any non-reentrant lock into reentrant (when desired).

Portability

LS is a library and a couple of APIs, which are wrappers over lock-unlock operations of any kind of lock.

High Performance

Lightweight design with minimal overhead compared to alternative solutions.

Real-World Impact

Successfully uncovers new bugs in Git software suites (e.g. Tensorflow, MPL, and Mempool).

Alternative Approaches

PID (process ID)-based solution

Mechanism:

Adds thread ID field to lock object to track lock ownership

Drawbacks:

- High runtime overhead - >15% over baseline[$]
- Not portable across systems

Lightweight Lock-specific changes

Mechanism:

Repurpose existing fields of lock object to store ID of lock owner. Modify acquire()-release() methods of **every lock algorithm.**

Drawbacks:

- Lacks generality
- Significantly increases complexity

The Need: A Generic, Portable, and Efficient Solution

[$] Vivek Shahare, Milind Chabbi, and Nikhil Hegde. Protecting Locks Against Unbalanced Unlock(). *SPAA '23*

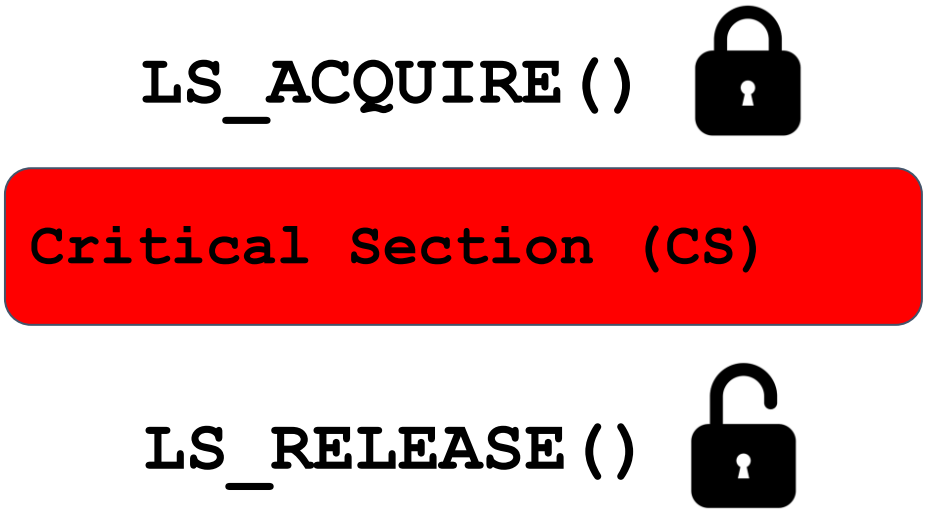
Lock Shielding - Working

- Lock Shielding consists of 2 APIs:
  - `LS_ACQUIRE(L, false, Lock)`
  - `LS_RELEASE(L, false, Unlock)`
  - Supports arbitrary lock function signatures using **variadic templates**.
- `Lock(ref L)`: underlying lock implementation
- `Unlock(ref L)`: underlying unlock implementation
- Works with legacy applications through **LD_PRELOAD**

Ref to Lock object

Reentrancy Flag

Function Pointer

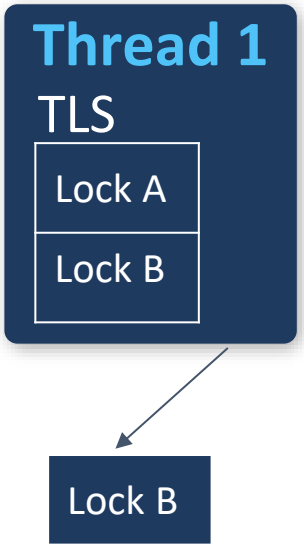


Lock Shielding – Data Structure (TLS)

- Each thread stores references of lock it holds in **thread local storage (TLS).**
  - e.g., `_Thread_local int counter = 0;`
  - No synchronization needed — TLS is private to each thread
- Implemented using:
  - **Arrays:** Default option. 4 entries, fits 1 cache line
  - **Hash Table** (`uthash`)*: necessary for handling applications that simultaneously hold many locks (nesting depth). O(1) insert / lookup / delete
- Memory Complexity:
  - **`O(T × D),`** where T = # threads, D = max lock nesting depth

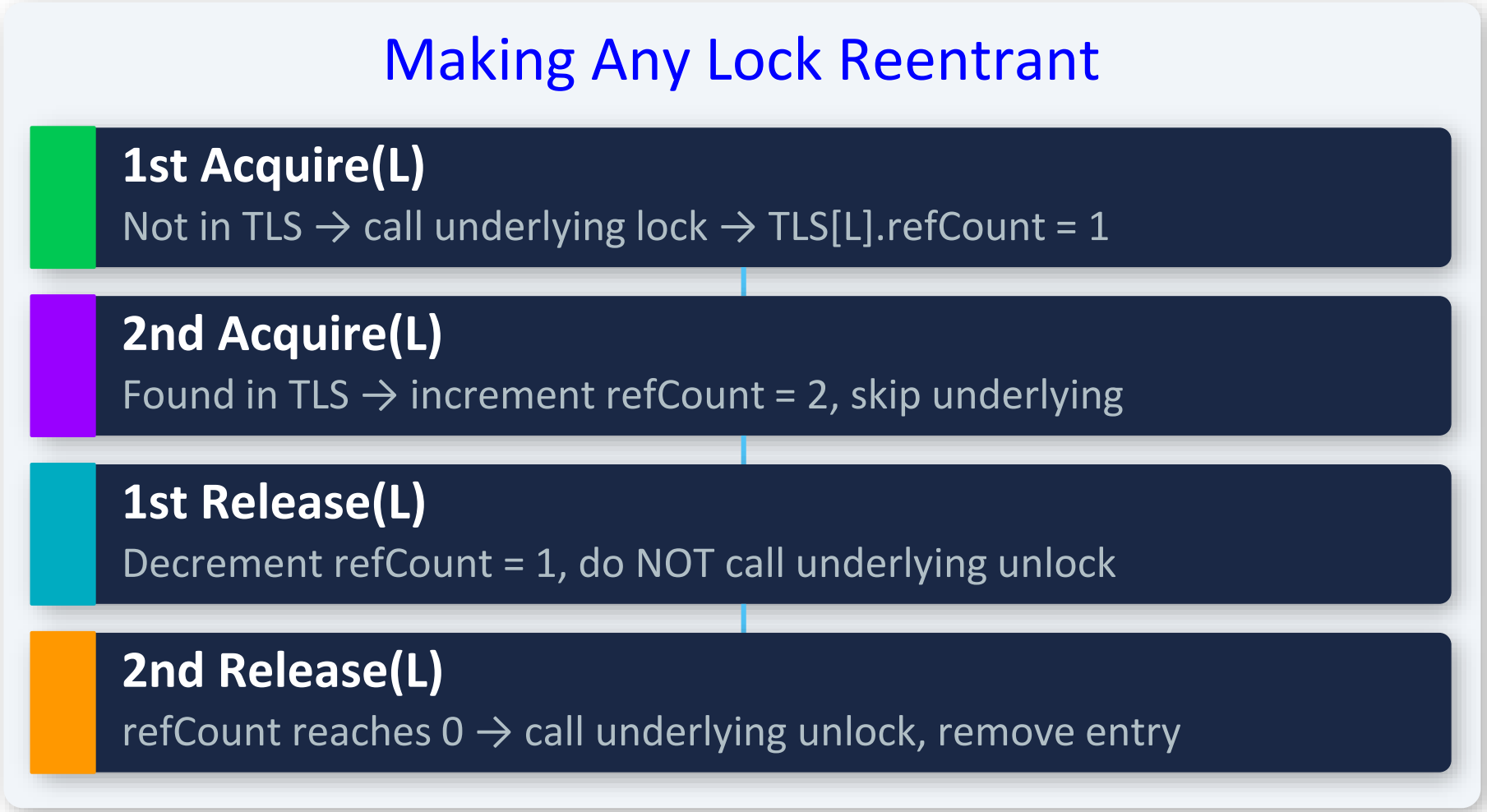
*Troy D. Hanson. 2021. uthash: a hash table for C structures. https://troydhanson.github.io/uthash/

Lock Shielding – Event Sequence



1. **`LS_Acquire` called:**
   Checks TLS for the lock pointer
2. **Lock pointer already in TLS?**
   If Non-reentrant→UNBALANCED_LOCK
3. **Otherwise update TLS:**
   Adds lock reference to thread's TLS (or increments ref count)
4. **Call underlying**
   `lock.Acquire()`
5. **`LS_Release` called**
   Checks TLS: lock reference not present→ UNBALANCED_UNLOCK; else decrement ref count

Lock Shielding – Use Case



Underlying lock APIs are never invoked recursively — correctness preserved.

Lock Shielding Variants

	LS_Array Fixed-size array	**LS_Hybrid** Adaptive TLS (array + hash table)
TLS Data Structure	TLS array	TLS array-> hashtable on overflow
Memory Complexity	`O(T × MAX_LOCKS)` fixed	`O(T × D)`, D = actual nesting depth
Lookup Cost	Linear scan on array	Linear scan ≤ 4 then O(1) hash table
Deep Nesting	Cap at MAX_LOCKS limit	Unlimited hash table auto-grows
Dynamic Allocation	None (fixed array)	Hash table pool; no runtime trimming

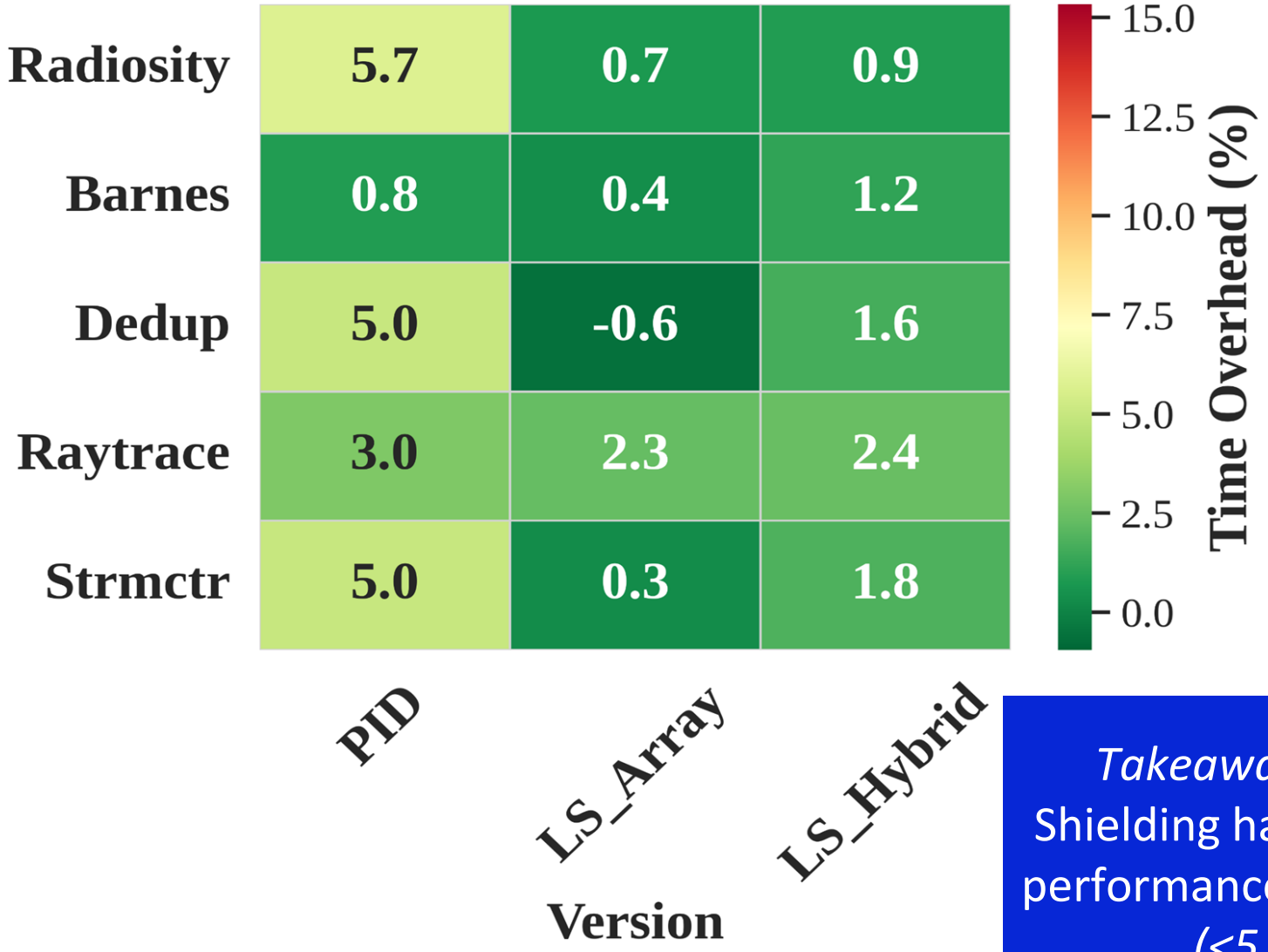
Experimental Setup

- System Configuration
  - Dual-socket, AMD EPYC 9534 @ 2.4GHz, 64-core processor per socket, 512 MB unified L3 cache, 251 GB main memory
- Benchmarks and Locks Evaluated
  - Synchrobench, SPLASH-2x [6] and PARSEC 3.0 [5], Intention Lock [3]
    - Hashtable, Skiplist, Hoh-list, Lazy-list, and RCU-tree
    - Barnes, Dedup, Radiosity Raytrace and Streamcluster
  - TAS, Ticket, ABQL, MCS, CLH, HMCS, K42, Hemlock, CNA

Characterization of Benchmark Applications

Application	#of locks	Depth	# of lock() ops
PARSEC 3.0 and SPLASH-2x			
Radiosity	3914	1	53 M
Barnes	2051	1	17 M
Dedup	98322	1	1 M
Raytrace	5	1	9 M
Streamcluster	191	1	79 K
Synchrobench			
Hashtable	780	2	296 M
RCU-Tree	265	5	337
Skiplist	263	7	438
Hoh-List	262	2	1.3 B
Lazy-List	280	2	247 M

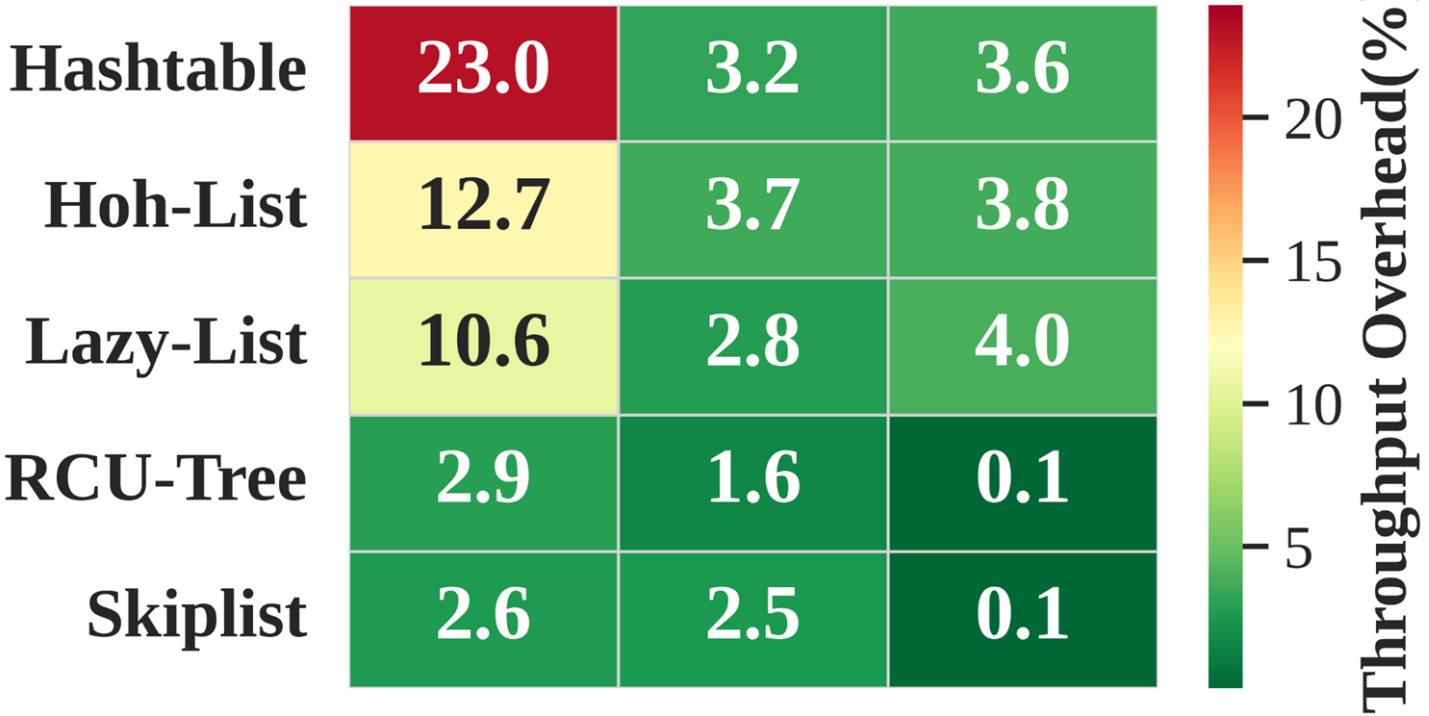
Results : **Parsec 3.0** and **Splash-2x** (MCS)



Takeaway: Lock Shielding has minimal performance overhead *(<5 %)*

Numbers indicate overhead percentage at 64 threads

Results : **Synchrobench** - (MCS Lock)

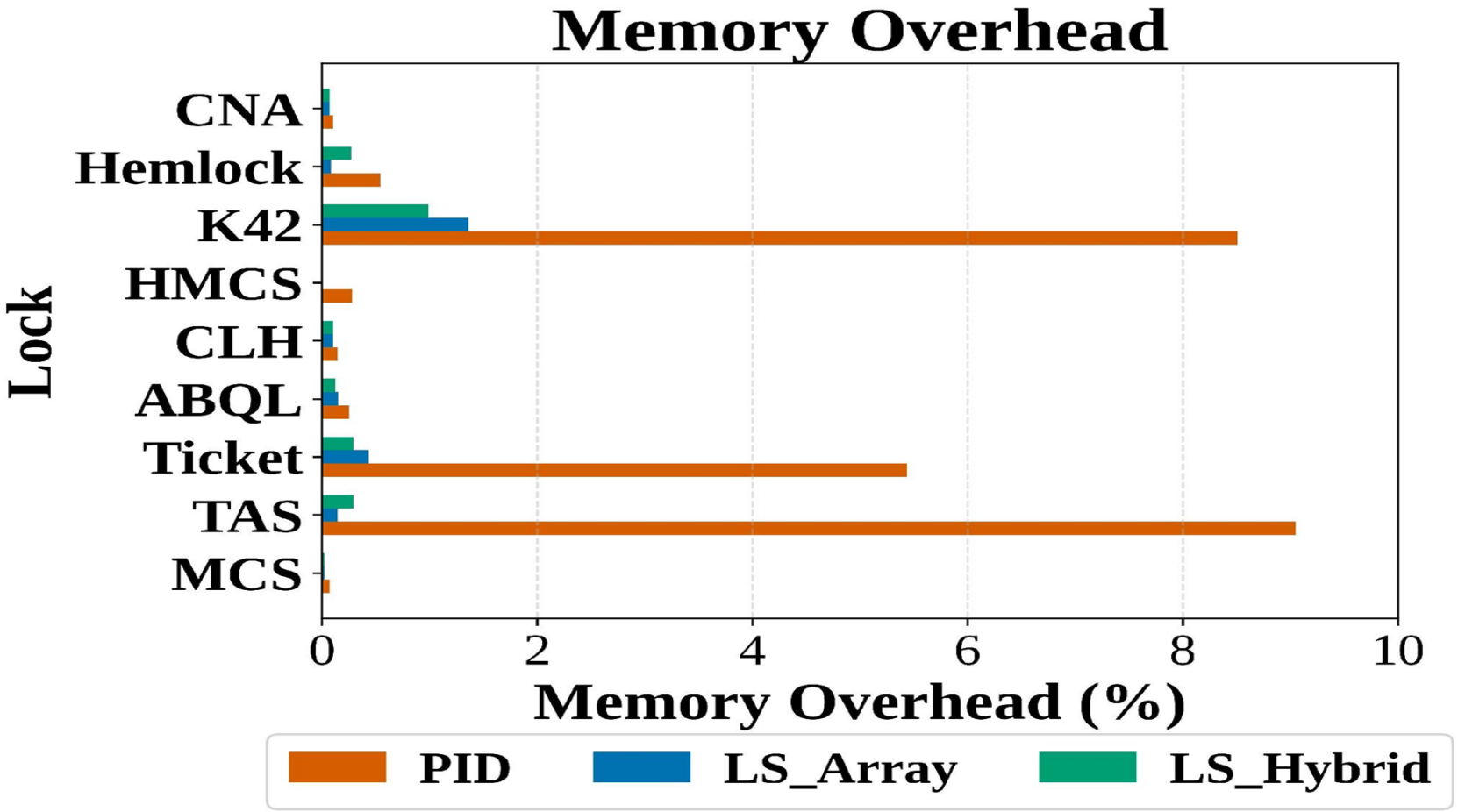


Takeaway 1: Overhead of LS_Array < LS_Hybrid < PID

Takeaway 2: Overhead of LS_Hybrid< LS_Array < PID for skip list and RCU

Numbers indicate overhead percentage at 64 threads

Memory Usage Statistics for *Intention Lock* Benchmark



Takeaway : Lock Shielding has least memory overhead

Numbers indicate memory usage at 64 threads

Lock Shielding: **Real-World Impact**

Evaluating 60 open-source repositories — uncovering hidden concurrency bugs

Repository Selection Criteria

- Frequent lock-based synchronization as primary primitive

Notable targets: LLVM, LevelDB, OpenSSL, OpenMPI and 56 more

Bug #1 — TensorFlow Machine Learning Framework

TensorFlow 2.20.0

- **Component:** Tensor Slice Reader Cache
- **Bug:** **Unbalanced Unlock** (RAII + Manual Unlock)

What Happened

- RAII lock mixed with manual `lock()-unlock()`
- Exception during object construction triggers double unlock

Buggy Code (tensor_slice_reader_cache.cc)

```
const TensorSliceReader* TensorSliceReaderCache::GetReader(..

mutex_lock l(mu_); // RAII: auto-acquires mu_

if (readers_.find(filepattern) == readers_.end()) {
still_opening_.insert(filepattern);

mu_.unlock(); // Manual unlock — BREAKS RAII contract!

TensorSliceReader* tmp_reader(
new TensorSliceReader(...)); // ← may throw!

mu_.lock(); // Manual relock
// ... analyse constructor results ...
}
return;
// BUG: if exception thrown above,
// RAII destructor calls unlock() again → double unlock!
}
```

Bug #2 — MPL/MaPLe Functional Programming Runtime

MaPLe v0.5.3

- **Component:** Concurrent Garbage Collector
- **Bug:** Missing Unlock (**Unbalanced Lock**)

What Happened

- Loop acquires **i** mutexes but releases only **i−1**, leaving one mutex locked.
- **Lock Shielding** detects the subsequent re-acquire as an **unbalanced-lock**

Buggy Code (runtime/gc/concurrent-stack.c)

```
bool CC_stack_try_close(CC_stack* stack, HM_chunkList removed
// --- initial checks ---

for (i = 0; i < stack->numStacks; i++) {
pthread_mutex_lock(&(stack->stacks[i].mutex));
HM_chunkList thisBag = &(stack->stacks[i].storage);
if (NULL != thisBag->firstChunk)
break; // ← exits loop early
stack->stacks[i].isClosed = TRUE;
}
if (i == stack->numStacks) {
stack->allClosed = TRUE;
for (size_t j = 0; j < i; j++) {
pthread_mutex_unlock(&(stack->stacks[j].mutex));
} // BUG: i-th mutex never unlocked!
return TRUE;
}
for (size_t j = 0; j < i; j++) {
pthread_mutex_unlock(&(stack->stacks[j].mutex));
} // BUG: same off-by-one here
```

Bug #3 — mem-pool Memory Management Library

Mem-pool 1.1.2

- **Component:** Variable-size memory pool management
- **Bug:** Missing Unlock (**Unbalanced Lock**)

What Happened

- `pool_variable_aligned_sizeof()` returns early before calling `unlock()`.
- The mutex remains locked, causing subsequent calls to **deadlock**.

Buggy Code (src/variable.c)

```
MemPoolError pool_variable_aligned_sizeof(
VariableMemPool *pool, void *ptr, size_t *size) {

lock(pool); // ← acquire mutex

if (!buffer_list_find(pool->buff_head, ptr)) {
return MEM_POOL_ERR_UNKNOWN_BLOCK;
// BUG: early return — unlock() never called!
}

SizedBlock *block = (SizedBlock *)(
(char *)ptr - pool->header_size);
*size = block->header.size;

unlock(pool); // ← never reached on error path
return MEM_POOL_ERR_OK;
}
```

Conclusions

Lock Shielding is generic , portable and efficient design that works across systems without modifying lock internals.

Lock Shielding protects against both unbalanced-lock and unbalanced-unlock misuse and enable any non-reentrant lock reentrant.

Lock Shielding has low overhead and highly scalable for high-performance concurrent applications.

Real-World Impact:

Uncovered critical bugs in popular git repositories like **TensorFlow, MaPLe,** and **mem-pool**. Reported them and fixes were upstreamed.

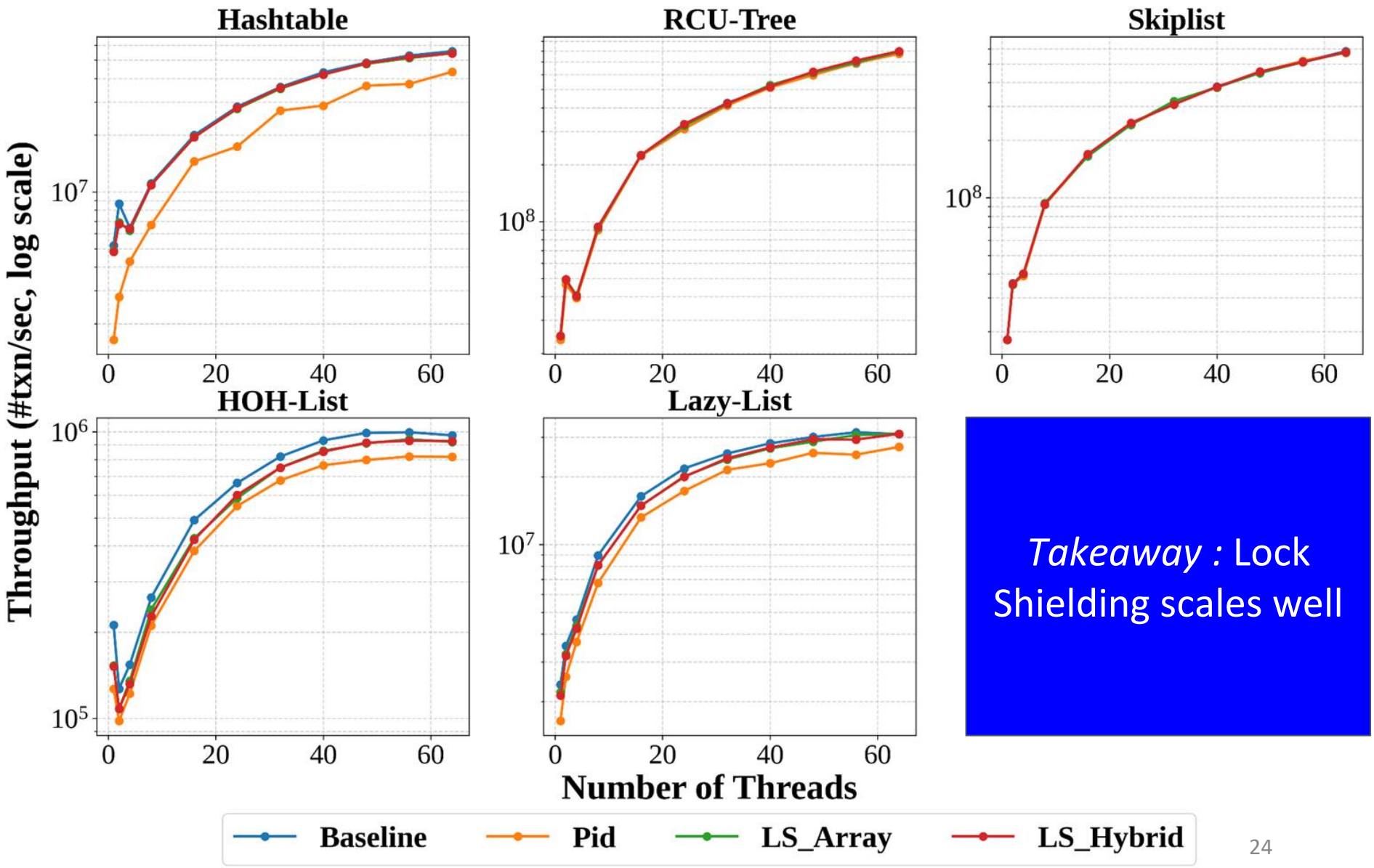
References

1. Vivek Shahare, Milind Chabbi, and Nikhil Hegde. 2023. Protecting Locks Against Unbalanced Unlock(). In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23)*: 199–211, 2023
2. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991
3. Saurabh Kalikar and Rupesh Nasre. DomLock: A New Multi-Granularity Locking Technique for Hierarchies. *ACM Trans. Parallel Comput.* 4(2), 2017
4. Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.
5. PARSEC Group et al. 2011. A memo on exploration of SPLASH-2 input sets. Princeton University.
6. Synchronization Constructs - OMSCS Notes; www.omscs-notes.com. Retrieved April 12, 2022, from https://www.omscs-notes.com/operating-systems/synchronization-constructs/
7. Spinlocks. (n.d.). www.cs.rochester.edu. Retrieved April 14, 2022, from https://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html

and more…

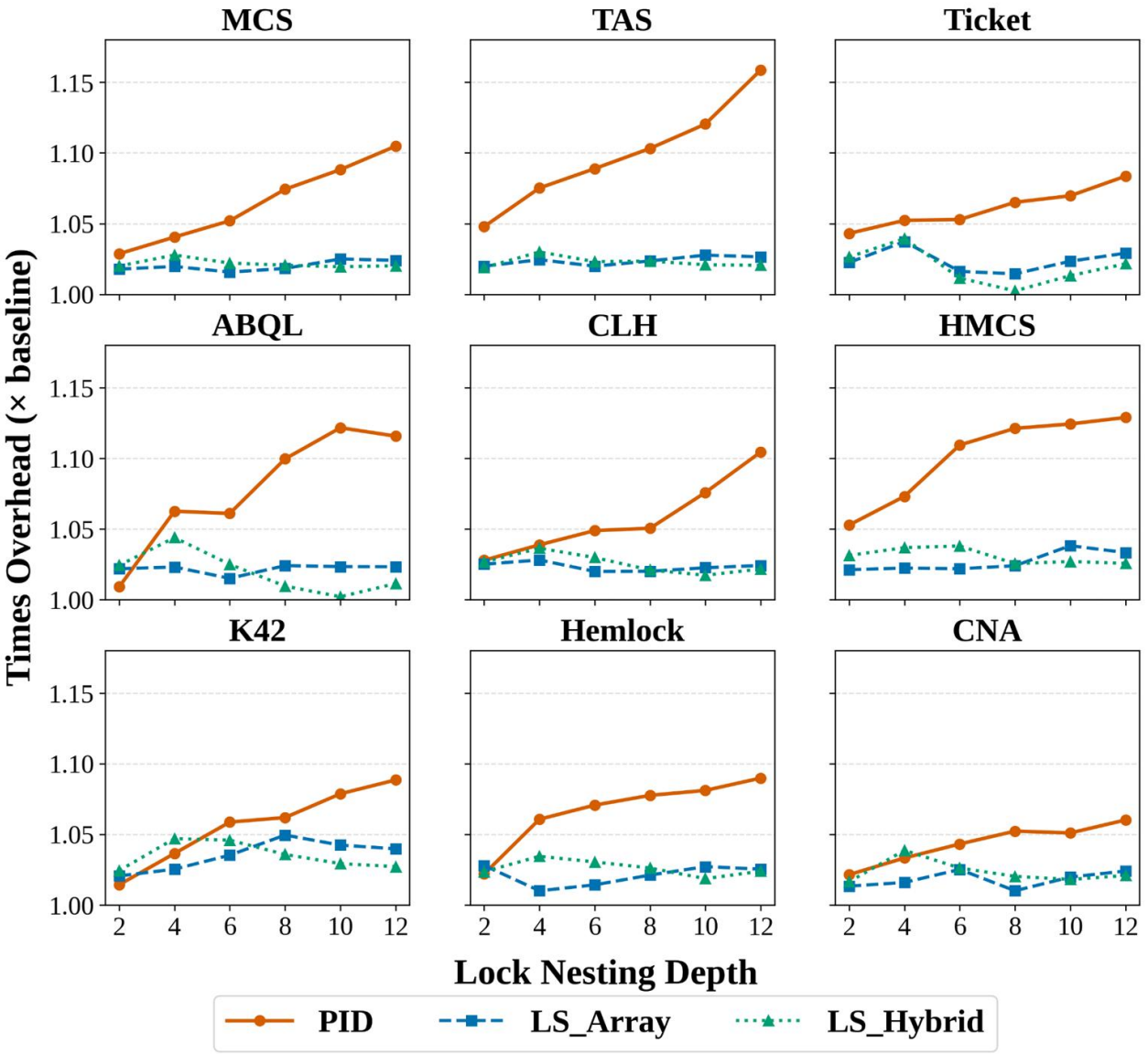
Thank You! Questions?

Scaling - Synchrobench-MCS Lock



Takeaway : Lock Shielding scales well

Results : Synthetic

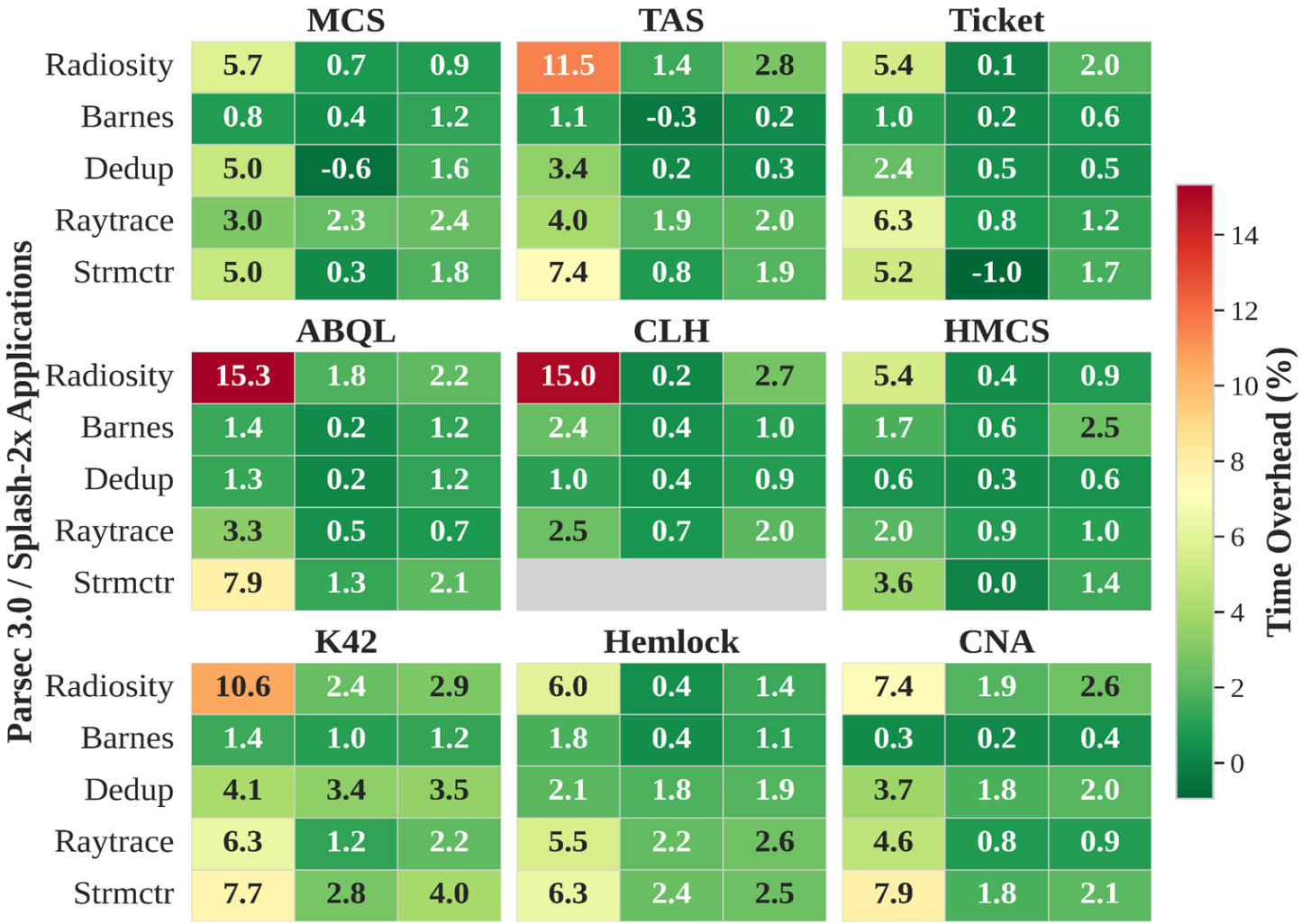


Takeaway 1:
*LS_*Array: best for shallow nesting workloads

Takeaway 2:
LS_Hybrid: best for deep nesting workloads

Numbers indicate overhead percentage at various lock nesting depth at 64 threads

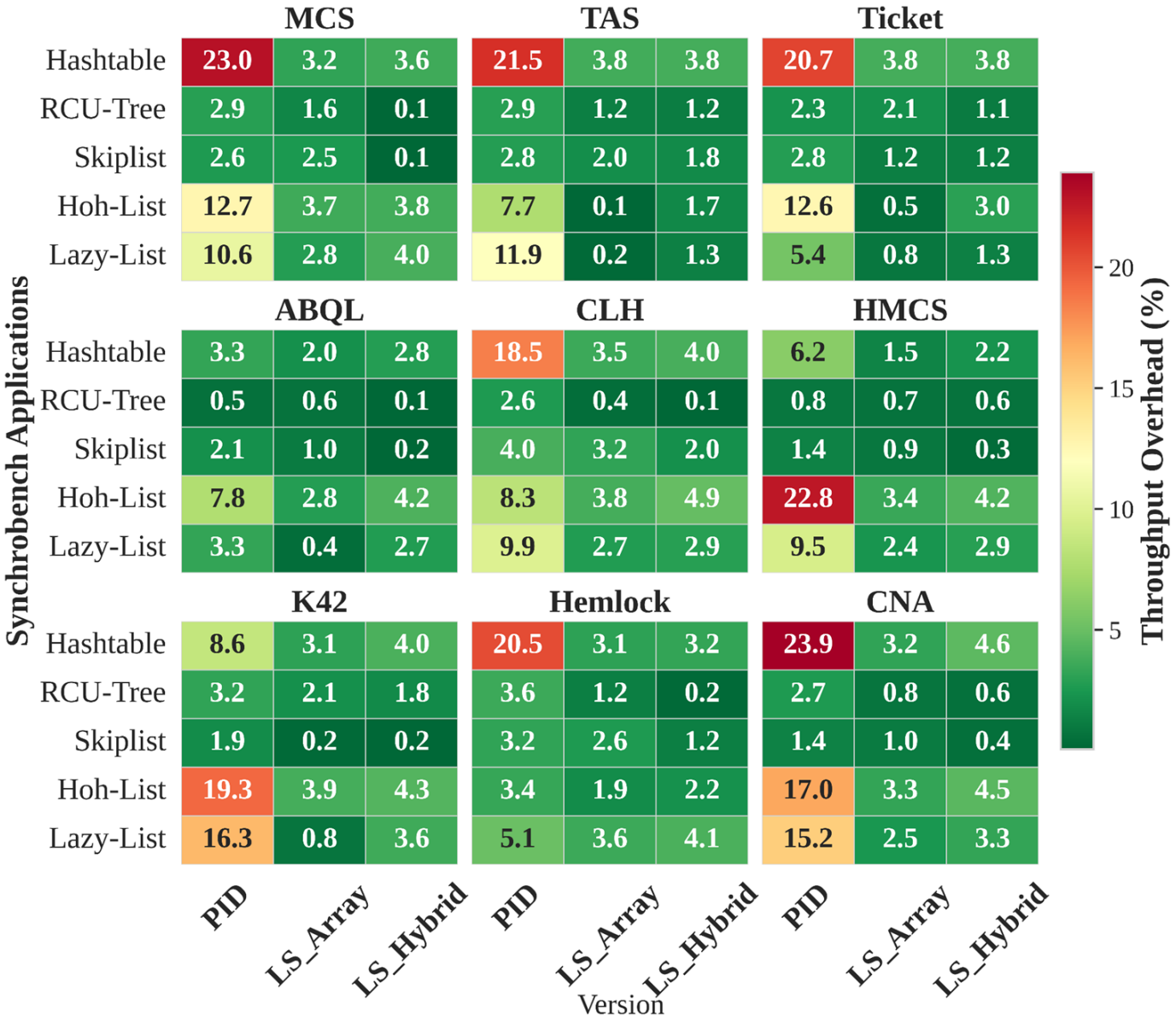
Lock Shielding has minimal performance overhead *(<5 %)*



Takeaway: Lock Shielding Outperforms PID-based solutions significantly

Numbers indicate overhead percentage at 64 threads

Results : Synchrobench



Numbers indicate overhead percentage at 64 threads

Adoption cost: pure overhead study

Locking Mechanism	Variant	Throughput (1T / 64T)	% Overhead (1T / 64T)
Pthread	**pthread_mutex_t**	1.75E+08 / 4.30E+07	**baseline / baseline**
	RECURSIVE	1.47E+08 / 2.51E+07	15.98 / 41.53
	ERRORCHECK	1.47E+08 / 3.16E+07	15.97 / 26.41
	LS(pthread_mutex_t)	1.14E+08 / 1.66E+07	35.08 / 61.44
	pthread_rwlock_t	1.71E+08 / 2.34E+07	**baseline / baseline**
	LS(pthread_rwlock_t)	1.18E+08 / 1.71E+07	31.18 / 26.80
OMP	**omp_lock_t**	2.05E+08 / 2.09E+06	**baseline / baseline**
	omp_nest_lock_t	1.94E+08 / 3.17E+06	5.29 / -59.68
	LS(omp_lock_t)	1.30E+08 / 2.09E+06	36.66 / 1.97
C++	LS(std::mutex)	1.16E+08 / 1.92E+07	33.68 / 33.70
	std::shared_mutex	1.68E+08 / 2.27E+07	**baseline / baseline**
	LS(std::shared_mutex)	1.16E+08 / 1.83E+07	32.69 / 19.62

Takeaway: Lock Shielding incur higher overhead than Production libraries with empty CS.

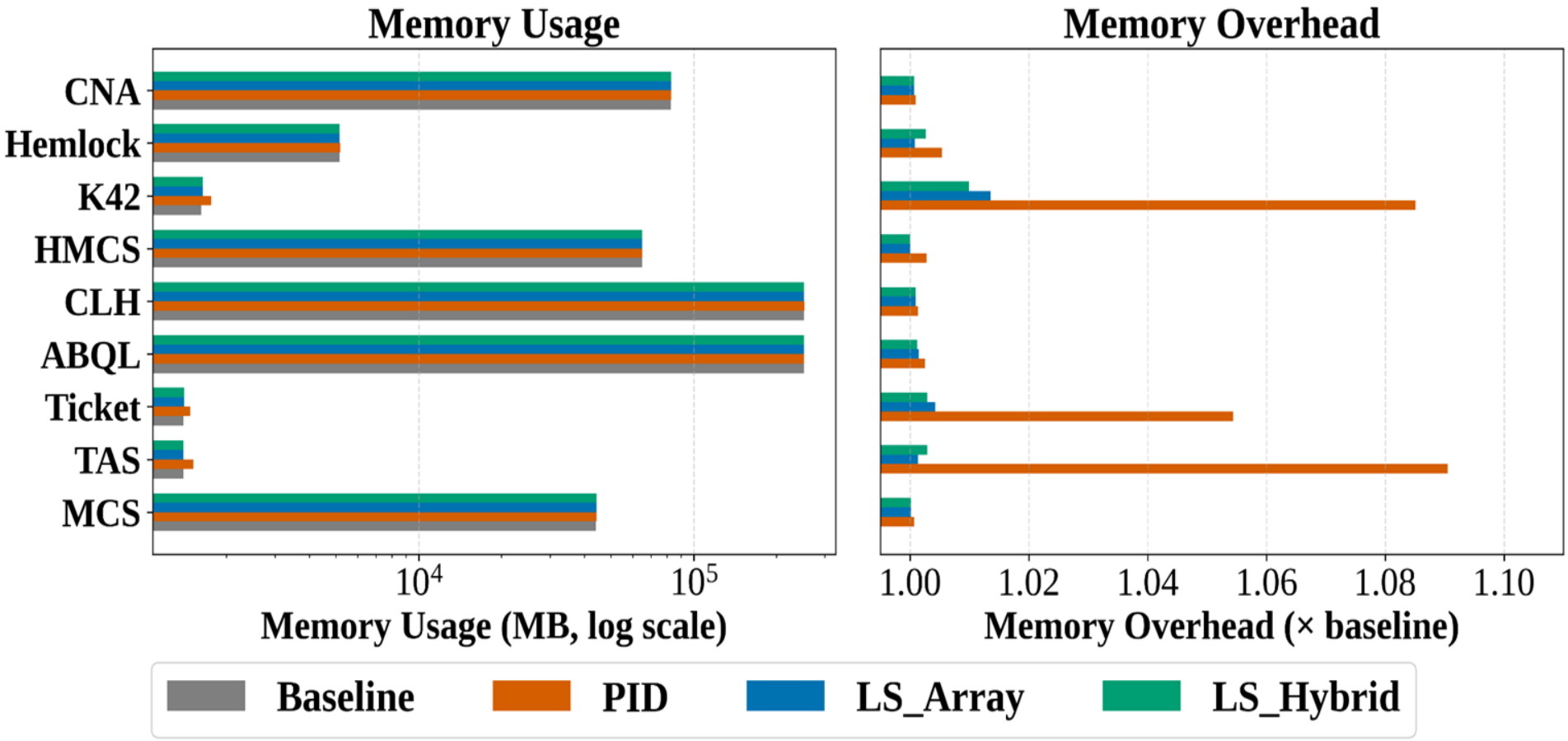
Overhead (%) of Pthread variants for Synchrobench

Application	RECURSIVE	ERRORCHECK	LS_Hybrid
Hashtable	2.58	**3.25**	2.79
RCU-Tree	**1.49**	-0.65	0.37
Skiplist	0.86	**1.64**	0.60
Hoh-List	**9.92**	4.77	4.44
Lazy-List	**4.53**	1.75	3.95

% overhead relative to pthread_mutex_t baseline - RECURSIVE / ERRORCHECK / LS_Hybrid

Takeaway: Lock Shielding is highly competitive with existing Pthread options

Memory Usage Statistics for *Intention Lock* Benchmark



Takeaway : Lock Shielding has least memory overhead

Numbers indicate memory usage at 64 threads

Support for Handling Misuse in Production Codes

Language Construct	Handles unbalanced-lock?	Handles unbalanced-unlock?
OMP `omp_lock_t`	✗ (deadlocks)	✗ (undefined behavior)
OMP `omp_nest_lock_t`	✓	✗ (undefined behavior)
pthread `pthread_mutex_t`	✗ (deadlocks)	✗ (undefined behavior)
pthread lock with `PTHREAD_MUTEX_RECURSIVE`	✓	✗ (undefined behavior)
pthread lock with `PTHREAD_MUTEX_ERRORCHECK`	✓	✓ (avoids)
C++11 `lock_guard<std::mutex>`	✗ (deadlocks)	✓ (avoids)
C++11 `lock_guard<std::recursive_mutex>`	✓	✓ (avoids)
C++11 `std::mutex` with explicit `lock()/unlock()`	✗ (deadlocks)	✗ (undefined behavior)
C++11 `std::recursive_mutex` with lock()/unlock()	✓	✗ (undefined behavior)
Java `synchronized` construct	✓	✓ (avoids)
Java ReentrantLock	✓	✗
`Boost::Mutex`	✗ (deadlocks)	✗
`Boost::recursive_mutex`	✓	✗ (undefined behavior)
`Boost::Mutex` (Checked)	✓	✓

Baseline version **Reentrant Version** **RAII Model/Error Check**

Lock Shielding - Working

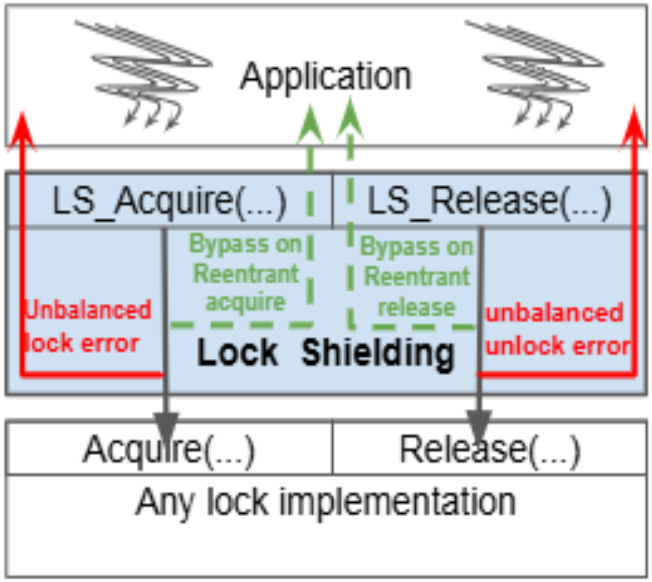
- Lock Shielding consists of 2 APIs:
  - `LS_ACQUIRE(L, false, Lock)`
  - `LS_RELEASE(L, false, Unlock)`
  - Supports arbitrary lock function signatures using **variadic templates**.

Ref to Lock object

Reentrancy Flag

Function Pointer

- `Lock(ref L)`: underlying lock implementation
- `Unlock(ref L)`: underlying unlock implementation
- Works with legacy applications through **LD_PRELOAD**



LS_ACQUIRE()
m.Lock()

Critical Section (CS)

LS_RELEASE()
m.Unlock()

Lock Shielding – Data Structure (TLS)

- Each thread stores references of lock it holds in **thread local storage (TLS).**
  - e.g., `_Thread_local int counter = 0;`
  - No synchronization needed — TLS is private to each thread
- Implemented using:
  - **Arrays:** Default option. 4 entries, fits 1 cache line
  - **Hash Table** (`uthash`)*: necessary for handling applications that simultaneously hold many locks (nesting depth). O(1) insert / lookup / delete

  Summary: complexity of ownership tracking is linear scan on array; expected constant-time on hash table

- Memory Complexity:
  - **`O(T × D),`** where T = # threads, D = max lock nesting depth

*Troy D. Hanson. 2021. uthash: a hash table for C structures. https://troydhanson.github.io/uthash/