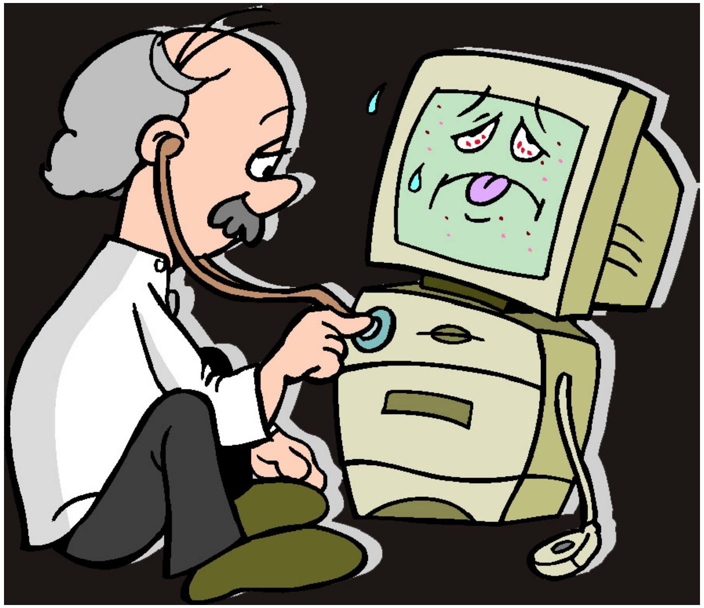
ERASER : A DYNAMIC DATA RACE DETECTOR FOR MULTITHREADED PROGRAMS

BY STEFAN SAVAGE ET AL.



KRISH

9/1/2011

MOTIVATION

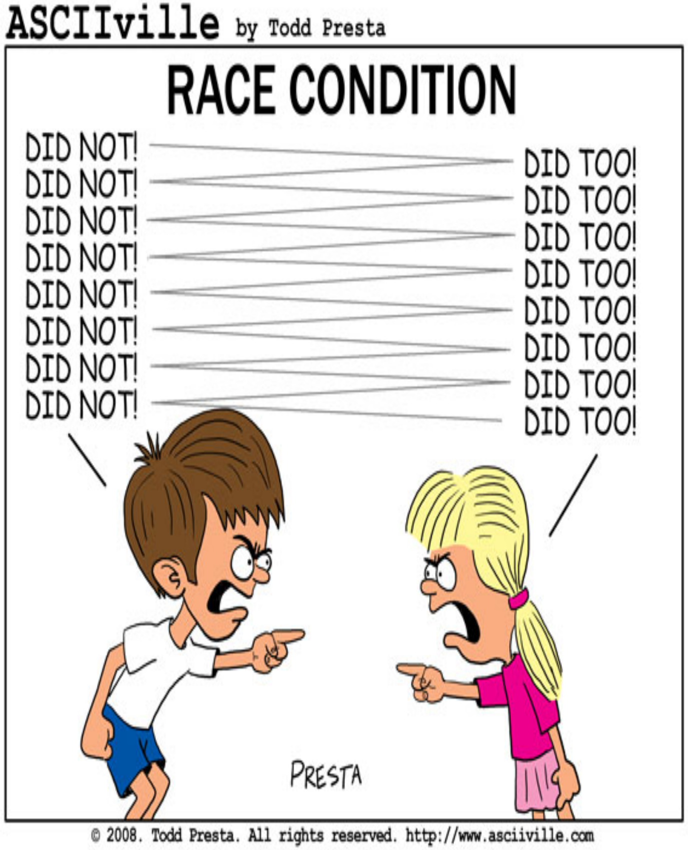
- Only parallel programs can benefit from today’s multi-core processors
- Multithreading has become a common(important) programming technique
- Synchronization between threads is a challenge
  - Synchronization errors are easy to cause and hard to debug
    - DATA RACE
  - Tracking errors take weeks and months

DATA RACE : NON-DETERMINISM

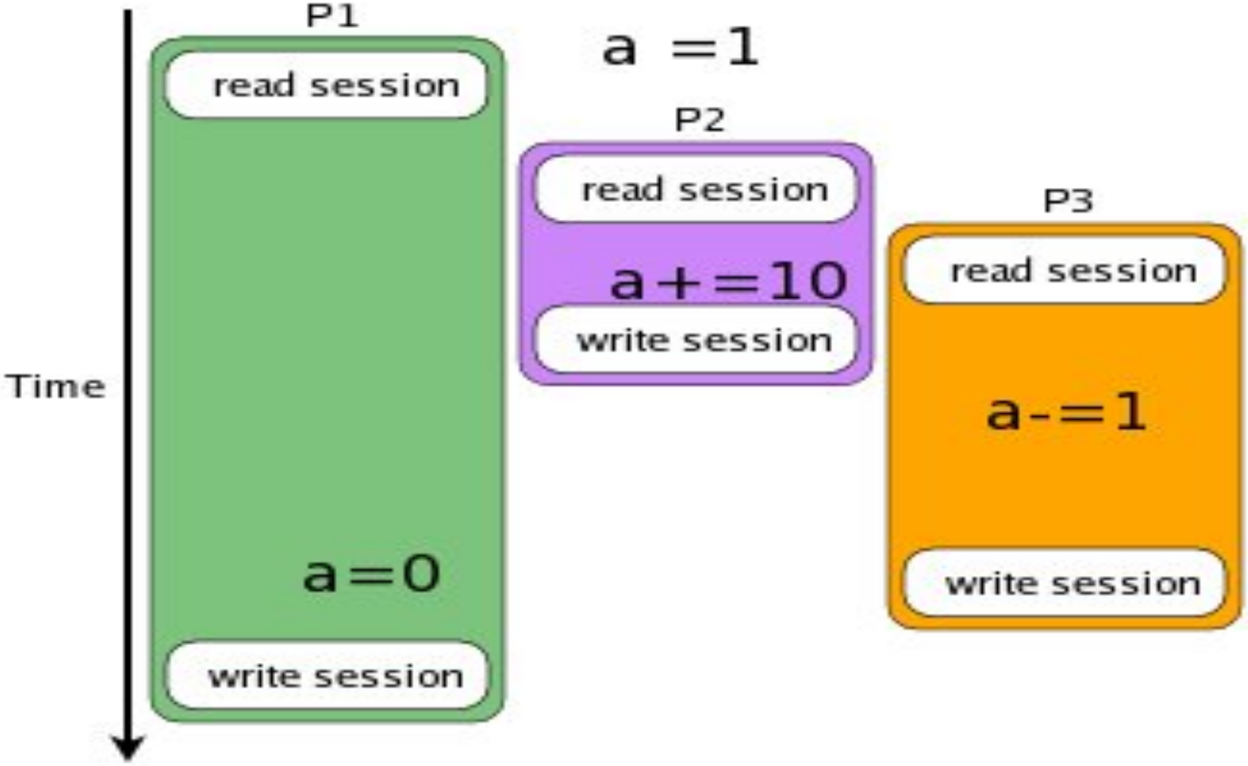
When two concurrent threads access a shared variable

1. at least one is a write and
2. the threads use no explicit synchronization to prevent simultaneous access

then the execution will depend on interleaving

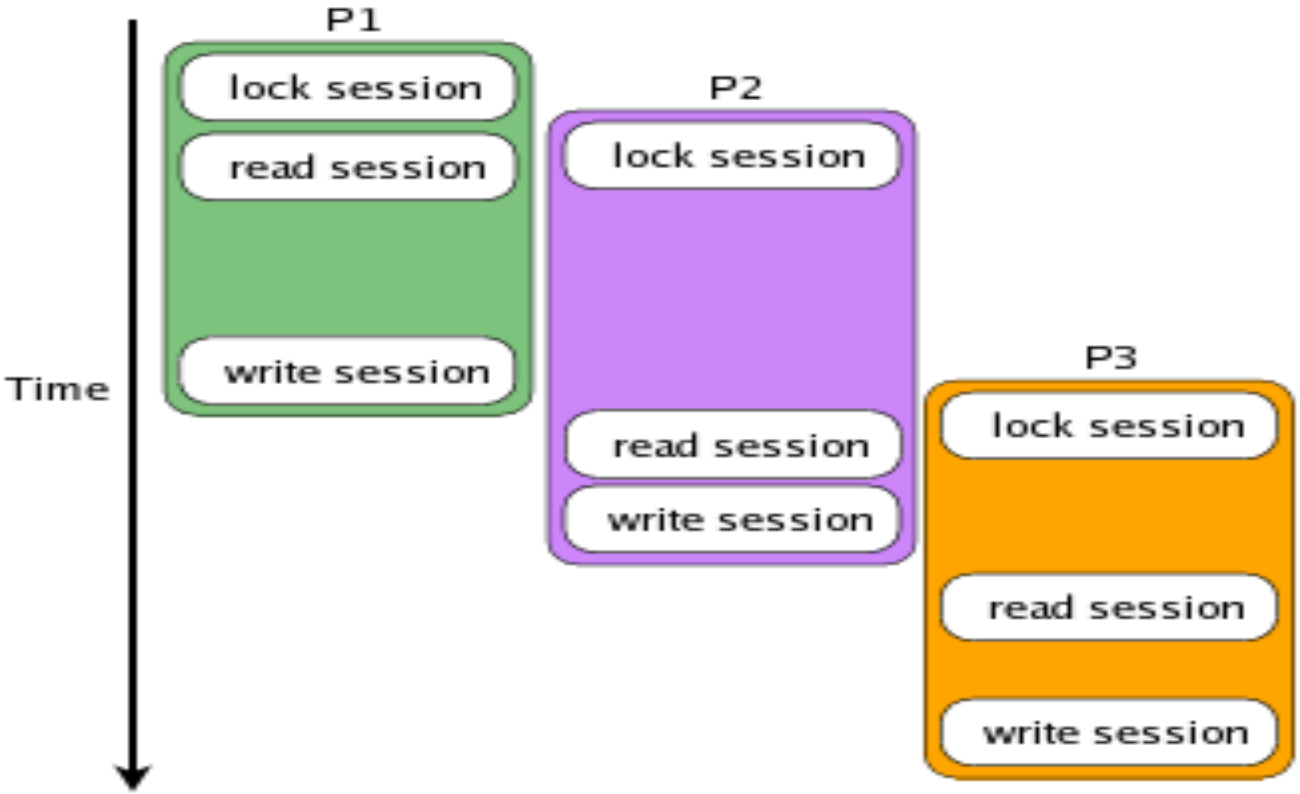


DATA RACE : NON-DETERMINISM



‘a’-----?????

DATA RACE : NON-DETERMINISM



Locks are used to avoid data race; Let's see a tool that detect's race.

COMPLEXITY IN DATA RACE DETECTION

- Data Race detection is a NP complete problem
- For t threads of n instructions, the number of possible orders is about t^{n*t}.
- A through detection will involve examining all the possible order to make sure there exist only one order.
- Practical race detection tools are based on heuristics - so that they can detect maximum number of races, with in limited computation

ERASER: A DYNAMIC RACE DETECTOR FOR MULTI-THREADED PROGRAMS

Stefan Savage
University of Washington
Michael Burrows, Greg Nelson, Patrick Sobalvarro
Digital System Research Center
Thomas Anderson
University of Washington

SOSP' 1997
ACM Transactions on Computer Systems, Nov. 1997.
Cited by 876

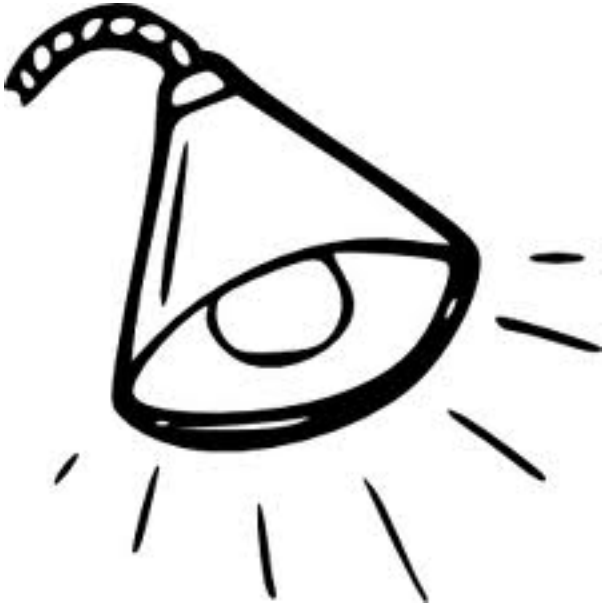
OBJECTIVE OF THE WORK

The work presents the theory, implementation and experience of a testing tool that detect dynamic data race in multi-threaded programs

AUTHOR'S CLAIMS

Does not identify all races in the program, but for programs using lock-based synchronization ensures better results than Lamport's (previous) work

OUTLINE

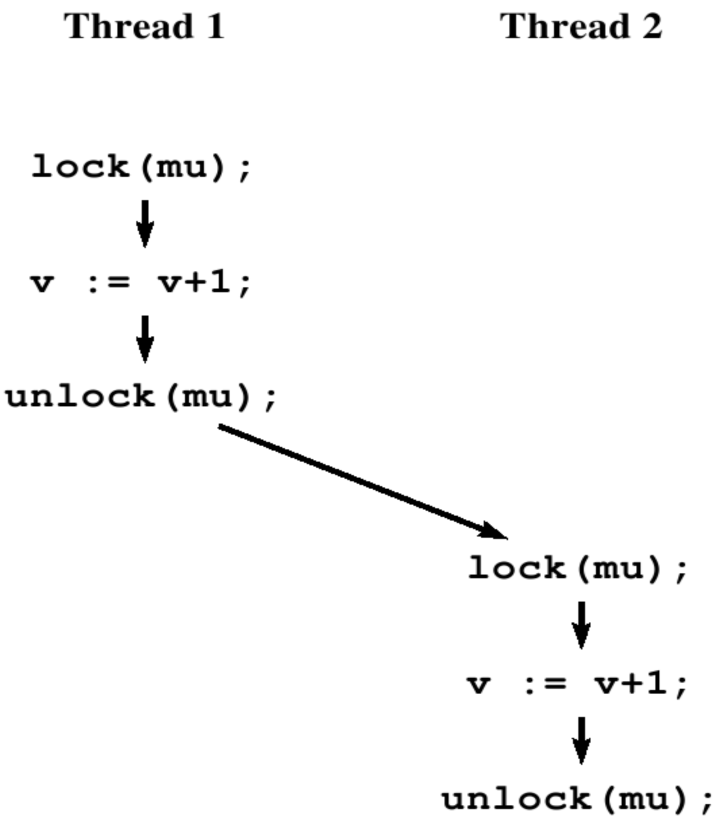


- Previous Work
- The Lockset algorithm
- Eraser implementation
- Experiences

LAMPORT'S HAPPENES-BEFORE

The happens-before order is a partial order on all events of all threads in a concurrent execution,

- Single Thread - events are ordered by their occurrence.
- Between threads - events are ordered by the synchronization objects they access.



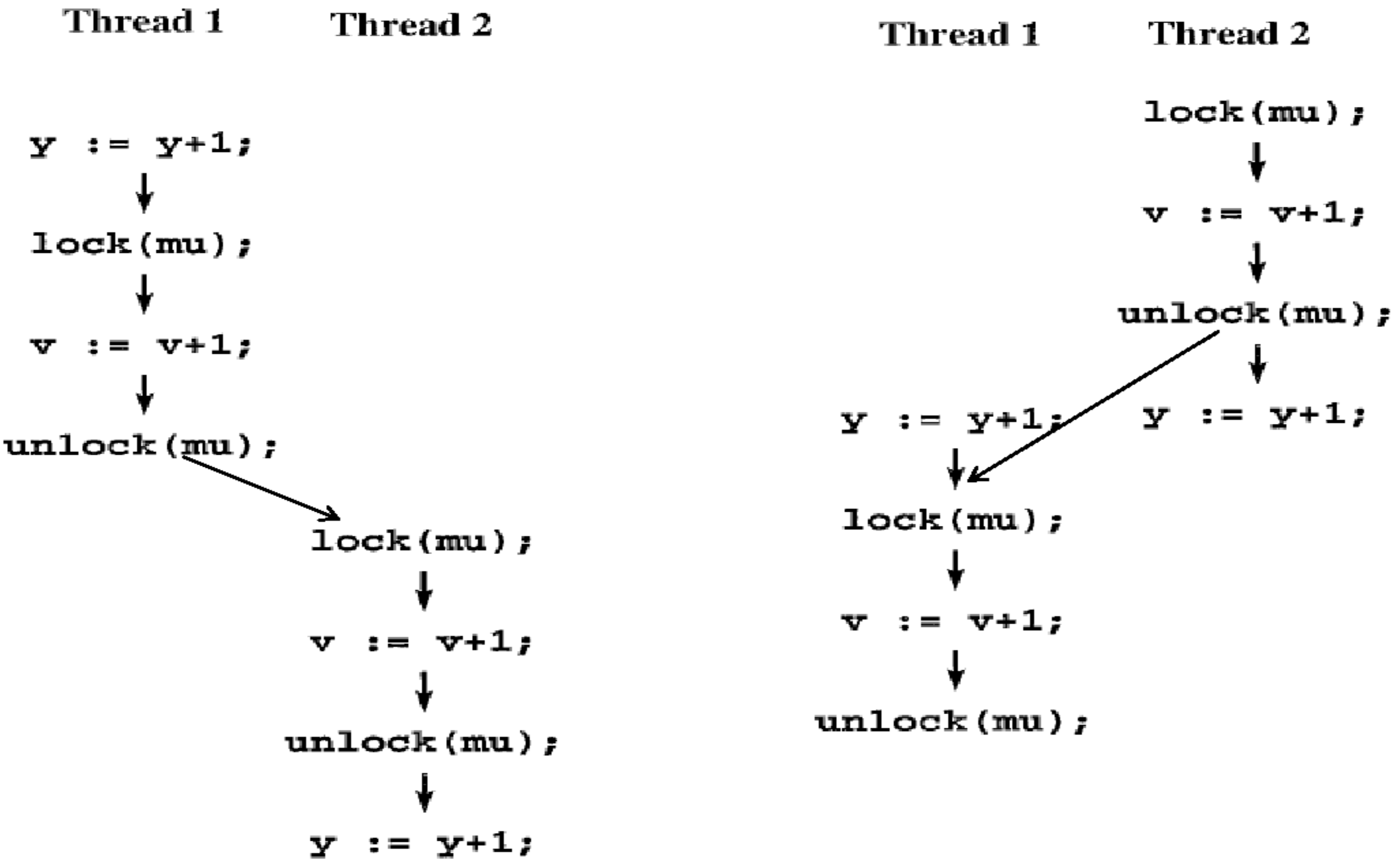
Thread 1

Thread 2

```
lock(mu);
v := v+1;
unlock(mu);
```

```
lock(mu);
v := v+1;
unlock(mu);
```

HAPPENS-BEFORE FAILS



Thread 1

```
y := y+1;
lock(mu);
v := v+1;
unlock(mu);
```

Thread 2

```
lock(mu);
v := v+1;
unlock(mu);
y := y+1;
```

Thread 1

```
y := y+1;
lock(mu);
v := v+1;
unlock(mu);
```

Thread 2

```
lock(mu);
v := v+1;
unlock(mu);
y := y+1;
```

Other Issues

- Difficult to implement efficiently - need per-thread information about ordering to all shared memory locations.
- Highly dependent on scheduler - needs large number of test cases.

ERASER'S APPROACH

Eraser uses binary rewriting techniques to monitor every shared memory reference and verify that consistent locking behavior is observed.

Heuristics – Consistent Locking Behavior

ERASER'S APPROACH

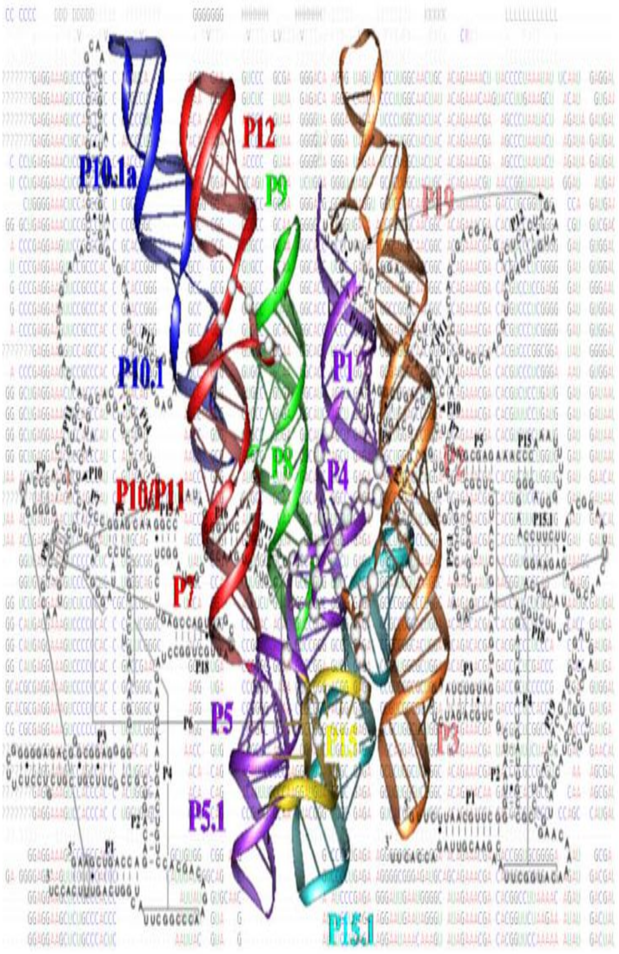
Eraser uses binary rewriting techniques to monitor every shared memory reference and verify that consistent locking behavior is observed.

'binary rewriting' – Observe all the Load and Store instructions.

'consistent locking behavior' - All instance of a shared variable v is locked by same set of locks L(v)

LOCKSET ALGORITHM

- v = shared variable
- C(v) = candidate locks for v
- locks_held(t) = set of locks held by thread t
- A lock l is in C(v) if all threads hold l while accessing v
- A new variable at initialization is supposed to have all possible locks in C(v)



TRAVERSING LOCKSET ALGORITHM

Program	***locks_held***	***C(v)***
int v;	{}	{mu1, mu2}
v := 1024;		
lock(mu1);		
	{mu1}	
v := v + 1;		
		{mu1}
unlock(mu1);		
	{}	
lock(mu2);		
	{mu2}	
v := v + 1;		
		{}
unlock(mu2);		
	{}	

INCREASING THE COMFORT ZONE

Initialization - Initialization can be done with out holding a lock.

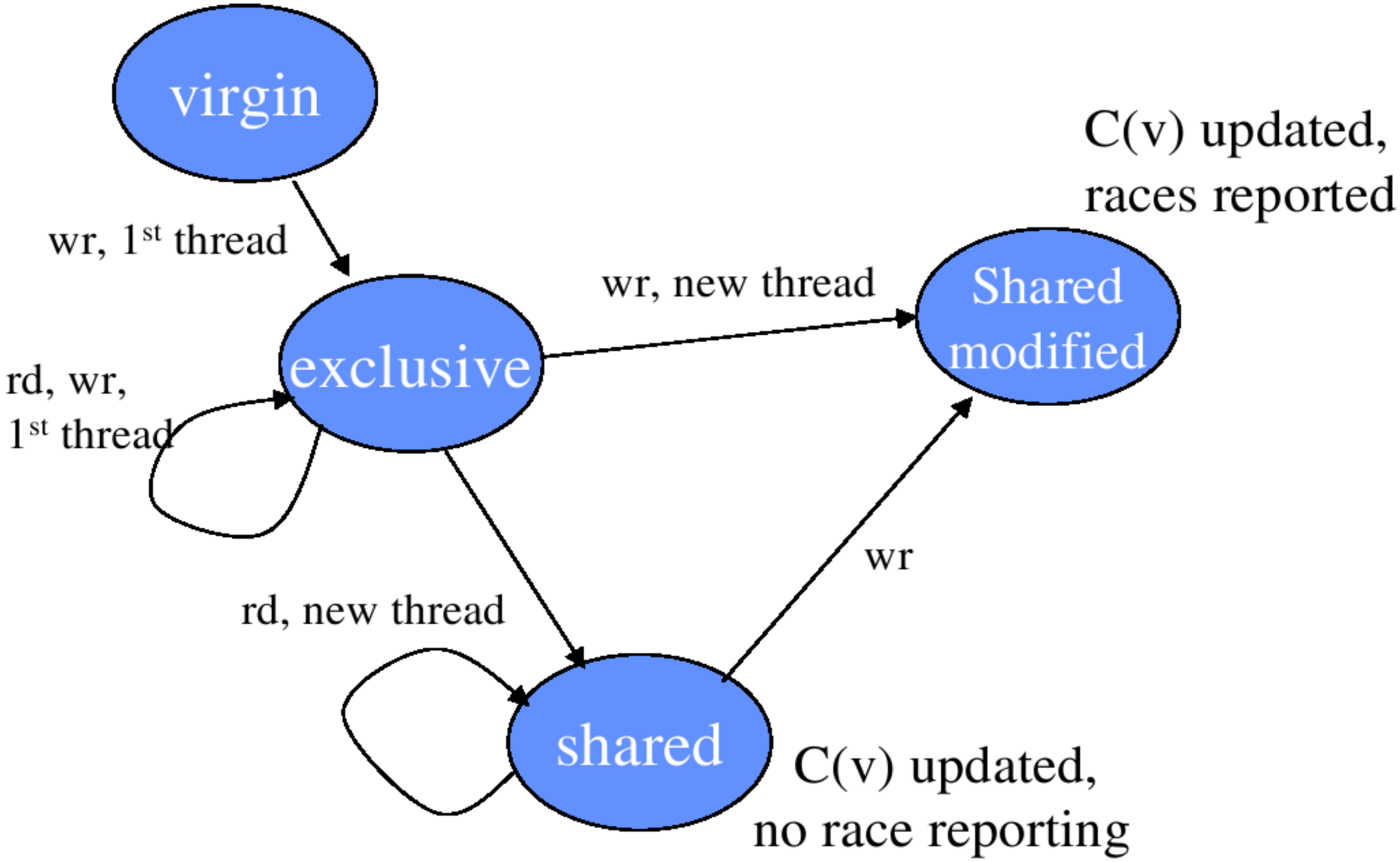
Read-shared data - Multiple reads safely accessed without locks.

Read-write locks - Multiple readers, but allow only a single writer.

INITIALIZATION AND READ-SHARING

- A variable is considered initialized when it is accessed by second thread
- Simultaneous reads of shared variable are not races
- Report races only after an initialized variable become write shared

PER-LOCATION STATE



virgin

wr, 1st thread

exclusive

rd, wr,
1st thread

wr, new thread

Shared
modified

C(v) updated,
races reported

rd, new thread

wr

shared

C(v) updated,
no race reporting

EXTENDING LOCKSET ALGORITHM

- Read / Write locking modes
- Locks held in read mode are removed from *$C(v)$* when a write occur
- *On each read of v by thread t,*
    - *set $C(v) := C(v) \cap locks\_held(t)$;*
    - *if $C(v) = \{\}$, then issue a warning.*
- *On each write of v by thread t,*
    - *set $C(v) := C(v) \cap write\_locks\_held(t)$;*
    - *if $C(v) = \{\}$, then issue a warning.*

Traversing Lockset Algorithm

	Program	locks_held	C(v)	State(v)
	int v;	{}	{mu1, mu2}	Virgin
T1	v := 1024;			
				Exclusive
T2	lock(mu1);			
		{mu1}		
	v := v + 1;			Shared
				Shared-Modified
	unlock(mu1);		{mu1}	
		{}		
T1	lock(mu2);			
		{mu2}		
	v := v + 1;			
			{}	
	unlock(mu2);			
		{}		

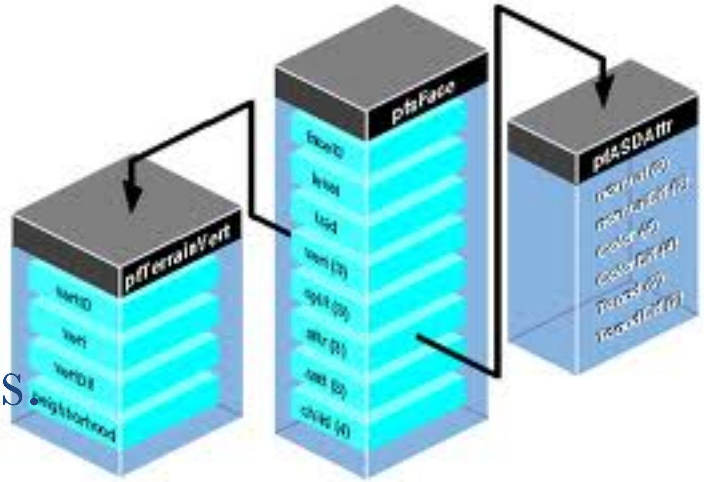
Race detected correctly

IMPLEMENTATION

Eraser's binary modification involves

1. Calls to storage allocator initializes C(v)
2. Each load and store updates C(v)
3. Each acquire or release call updates locks_held(t)

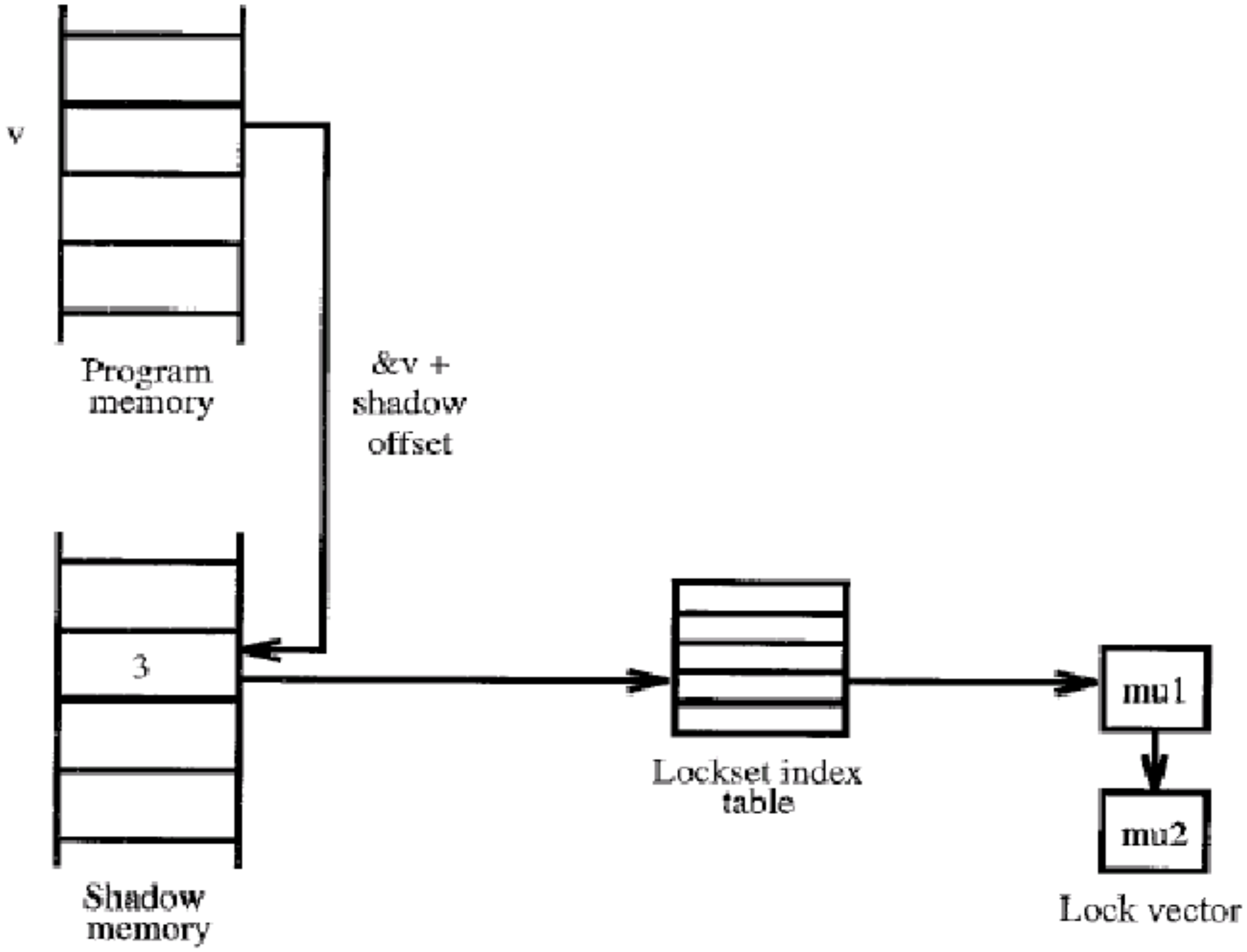
IMPLEMENTATION



Data Structure,

1. Maintains hash table of sets of locks.
2. Represents each set of locks with an index.
3. Every shared memory location has shadow memory containing lockset index and state.
4. Shadow memory is located by adding offset to shared memory location address.

Implementation



FALSE POSITIVES

- Memory reuse

  Caused by memory reset, with out resetting the shadow memory.

- Private locks

  When Locks that are not part of standard pthread interface are used.

- Benign races

  True Data race that did not affect the execution of the program.

"I have a positive attitude…
but it might be a false positive."

ANNOTATIONS TO AVOID FALSE POSITIVES

1. For memory reuse
    1. EraserReuse(address, size)
2. For private locks
    1. EraserReadLock(lock)
    2. EraserReadUnlock(lock)
    3. EraserWriteLock(lock)
    4. EraserWriteUnlock(lock)
3. For benign races
    1. EraserIgnoreOn()
    2. EraserIgnoreOff()

Experience

1. AltaVista
    1. *Mhttpd http server* - 5,000 lines of C source code, 100 distinct locks, 9 annotations.
    2. *Ni2* indexing engine - 20,000 lines of C source code, 900 distinct locks, 10 annotations.
2. Vesta Cache Server - 30,000 lines of C++ source code, 10 threads, 26 distinct locks, 10 annotations.
3. Petal - distributed disk server - 30,000 lines of C++ source code, 64 threads
4. Undergrad coursework – 100 multi threaded programs

Experience

- Deliberately introduced race conditions were detected.
- Other data races were also detected.
- False alarms were raised, but use of annotations resolved sizable number of them.

Experience

Program	*Serious races*	*Minor races*	*Benign races*
AltaVista		✓	✓
Vesta	✓	✓	
Petal		✓	
Undergrad assignments	✓		

PERFORMANCE

- NP-Complete – Computationally hard problem.
- Implementing Eraser slows down the application by a factor of 10 to 30
- Overhead of making a procedure call at every load and store instruction
- Performance was never a major goal

RECAP :OBJECTIVE OF THE WORK

The work presents the theory, implementation and experience of a testing tool that detect dynamic data race in multi-threaded programs

Conclusion

- Uses the locking principles as a heuristics to identify data races
- Successful implemention identifies potential races in enterprise software
- Implementing Eraser slows down the application by a factor of 10 to 30
- Diverse case study was given to support the tool

LET'S DISCUSS