EFFICIENT DATA-RACE DETECTION

Recap

- Happens-before based data-race detection
  - Shaz's guest lecture
  - Vector clock algorithm
- Lock-set based data-race detection
  - Eraser paper discussion

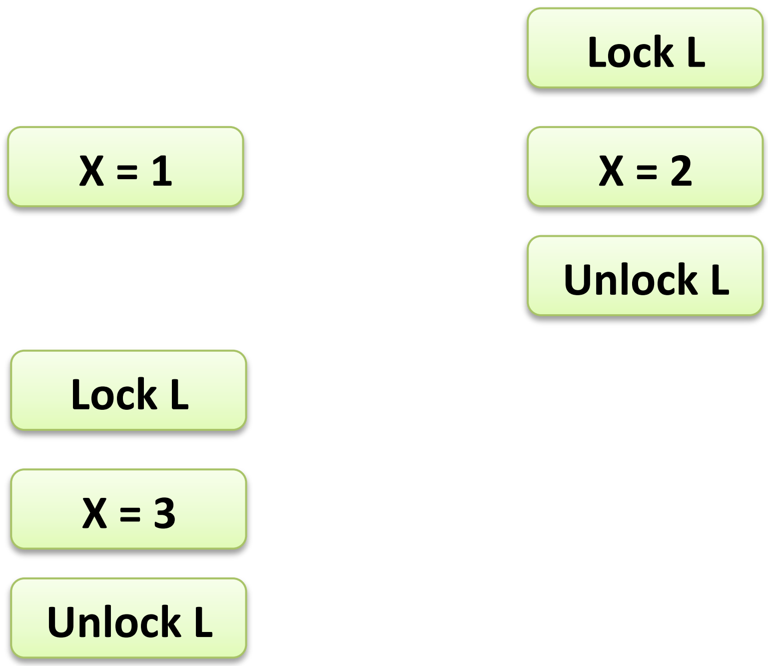
New Definition of a Data Race

- Two instruction conflict if
  - They access the same memory location
  - At least one of them is a write
- A program contains a data race if
  - One can schedule the program on a multiprocessor
  - Such that conflicting instructions execute simultaneously

New Definition vs Happens-Before

X = 1	
	Lock L
	X = 2
	Unlock L
Lock L	
X = 3	
Unlock L	

New Definition vs Happens-Before



New Definition vs Happens-Before

- True or False ?
- If a program has a data race as per the new definition
- Then it has a data race as per the happens-before definition

New Definition vs Happens-Before

- True or False ?

- If a program has a data race as per the new definition
- Then it has a happens-before data race

- If a program has a happens-before data race
- Then it has a data race as per the new definition

New Definition vs Lock Set

- True or False ?

- If a program has a data race as per the new definition
- Then it has a lock-set based data race

- If a program has a lock-set based data race
- Then it has a data race as per the new definition

Challenges For Dynamic Data Race Detection

- Dynamically monitor memory accesses
  - Need instrumentation infrastructure
  - Performance overhead

Challenges For Dynamic Data Race Detection

- Dynamically monitor memory accesses
  - Need instrumentation infrastructure
  - Performance overhead
- Understand synchronization mechanisms
  - Handle home-grown locks
  - Need to annotate the happens-before relationship

Challenges For Dynamic Data Race Detection

- Dynamically monitor memory accesses
  - Need instrumentation infrastructure
  - Performance overhead
- Understand synchronization mechanisms
  - Handle home-grown locks
  - Need to annotate the happens-before relationship
- Maintain detection meta-data per variable, per synchronization object
  - Manage memory
  - Performance overhead

DataCollider [OSDI '10]

- Lightweight
  - < 5% overhead
- Effective
  - Found data races in all applications we have run
- Easy to implement
  - The algorithm can be described in ~ 10 lines

Design Choice #1

- Sample memory accesses
  - Detect data races on a small, random sample of memory accesses

Design Choice #1

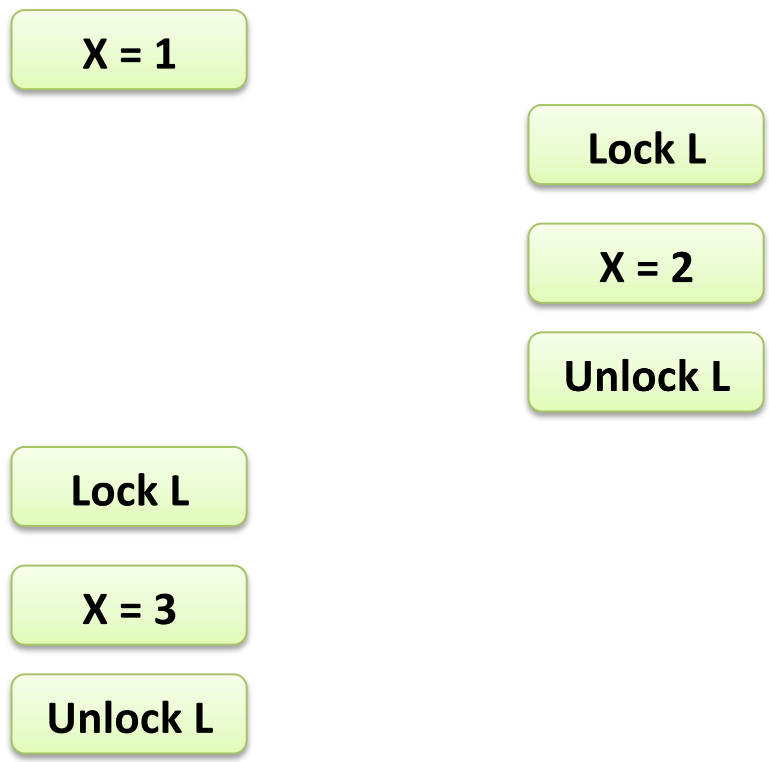
- Sample memory accesses
  - Detect data races on a small, random sample of memory accesses
- It is acceptable to miss races
  - Dynamic techniques cannot find all data races anyway
- A lightweight tool will have more coverage
  - Likely to be run on a lot more tests
- But data races are 'rare' events
  - Need to be careful when sampling

Design Choice #2

- Perturb thread schedules
  - By inserting delays
  - Force the data race to happen, rather than infer from past memory/synchronization accesses

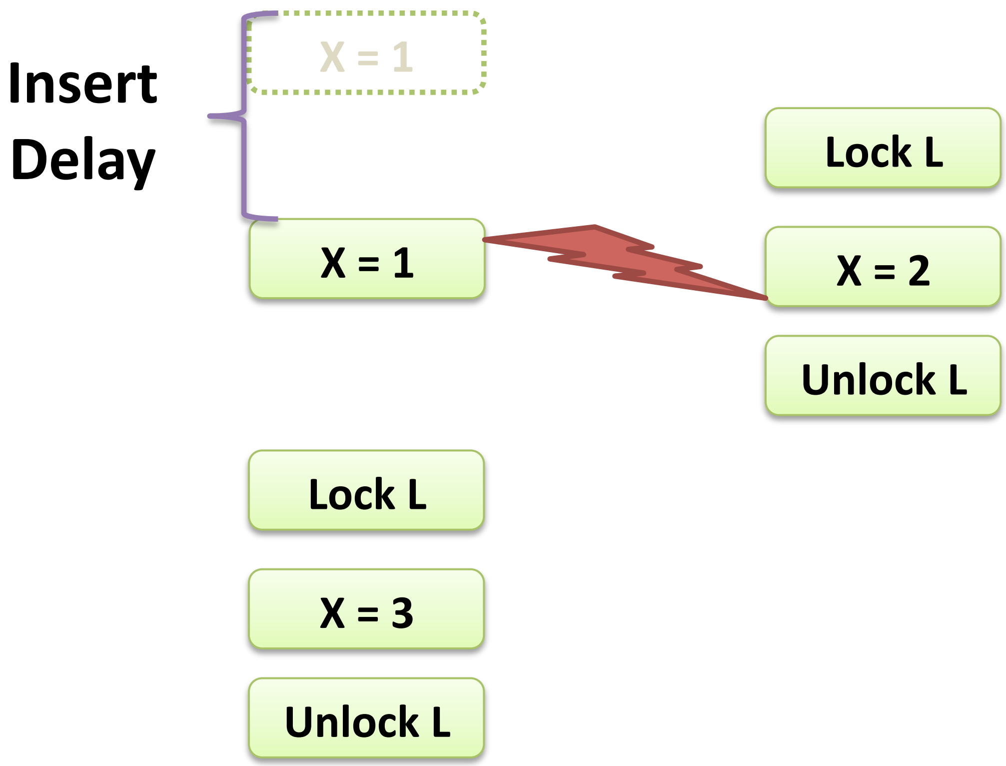
Design Choice #2

- Perturb thread schedules



Design Choice #2

- Perturb thread schedules



The Algorithm

```
At_Every_Memory_Access ( x ) {
   if (not sampled) return;

   delay for some time;

   if another thread performed
      a conflicting operation
      during the delay
     report data race;
}
```

Use Data Breakpoints for Detecting Conflicting Operations

```
At_Every_Memory_Access ( x ) {
   if (not sampled) return;
   rw = accessIsWrite ? RW : W;
   SetDataBreakpoint ( x, rw );
   Delay();
   ClearDataBreakpoint ( x );
   if (breakpoint fired)
      ReportDataRace();
}
```

Sampling Memory Accesses

- Need to execute the sampling function at every access
  - Performance overhead
- Need a good sampling algorithm
  - Data races are 'rare' events

Dynamic Sampling

1. Toss a biased coin at each memory access
2. Sample if coin turns up heads (with small probability)

Dynamic Sampling

1. Toss a biased coin at each memory access
2. Sample if coin turns up heads (with small probability)

- Hot (frequent) instructions are sampled overwhelmingly more that cold instructions
- Bugs (data races) are likely to be present on cold instructions

Code Breakpoint Sampling

1. Pick a random instruction X
2. Set a code breakpoint at X
   - Overwrite the first byte of X with a ‘0xcc’
3. Sample X when the breakpoint fires and goto Step 1

Sampling Using Code Breakpoints

- Samples instructions independent of their execution frequency
  - Hot and cold instructions are sampled uniformly

Sampling Using Code Breakpoints

- Samples instructions independent of their execution frequency
  - Hot and code instructions are sampled uniformly

```
repeat {
  t = fair_coin_toss();
  while( t != unfair_coin_toss() );
  print( t );
}
```

- Sampling distribution is determined by you (fair_coin_toss)
- Sampling rate is determined by the program (unfair_coin_toss)

Sampling Using Code Breakpoints

- Samples instructions independent of their execution frequency
  - Hot and code instructions are sampled uniformly

```
repeat {
  t = fair_coin_toss();
  while( t != unfair_coin_toss() );
  print( t );
}
```

Set a breakpoint at location X

Run the program till it executes X

Sample X

- Sampling distribution is determined by you (fair_coin_toss)
- Sampling rate is determined by the program (unfair_coin_toss)

The DataCollider Algorithm

```
Init:
   SetRandomCodeBreakpoints( n );

AtCodeBreakpoint( x ) {

   SetDataBreakpoint ( x );

   Delay();

   if (breakpoint fired)
      ReportDataRace();

   ClearDataBreakPoint ( x );

   SetRandomCodeBreakpoints( 1 );
}
```

DataCollider Summary

- Tunable runtime overhead
  - Tune code breakpoint rate so that we get k samples per second
- Finds lots of data races in practice
- The fastest implementation
  - Someone hacked it up during the talk
- Pruning intended data races is a problem
- Sampling algorithm improvements?