Optimizing Dynamic Race Detection With Hash Consing

David Moon '16 (dm7@williams.edu) and Stephen Freund (freund@cs.williams.edu)
Department of Computer Science at Williams College

August 14, 2015

Multithreaded Programs and Race Conditions

- Multithreaded programs run multiple threads simultaneously, the steps of which are interleaved by a scheduler.
- Each thread can read and write to memory.
- Threads may interfere with each other if they access the same memory location at the "same time".
- This is called a *race condition* and causes non-deterministic behavior.

A Broken Bank Account and Two Interleavings

```
Deposit:            Good Sched:         Bad Sched:
  t1 = bal            bal == 0            bal == 0
  bal = t1+10         t1 = bal            t1 = bal
                      bal = t1+10         t2 = bal
Withdraw:             t2 = bal            bal = t1+10
  t2 = bal            bal = t2-10         bal = t2-10
  bal = t2-10         bal == 0            bal == -10
```

Locking to Control Scheduling

- Programmers can use *locks* to avoid bad schedules.
- Locks are objects which can only be held by a single thread at any time, forcing all other threads to wait their turn.

Bank Account with Locking

```
Deposit:                 Schedule:
  acquire lock             bal == 0
  t1 = bal                 acquire lock
  bal = t1+10              t1 = bal
  release lock             bal = t1+10
                           release lock
Withdraw:                  acquire lock
  acquire lock             t2 = bal
  t2 = bal                 bal = t2-10
  bal = t2-10              release lock
  release lock             bal == 0
```

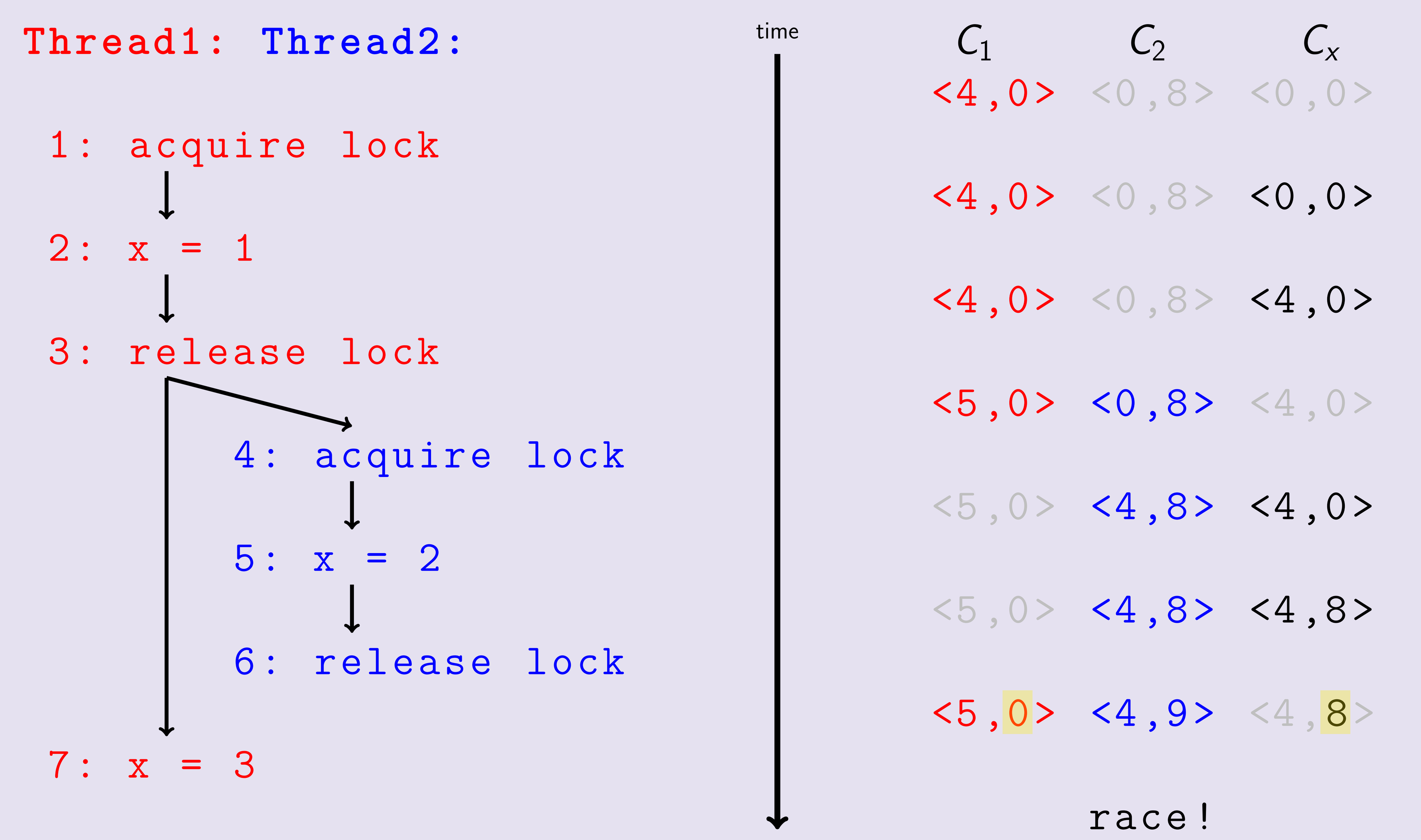
Race Conditions Are Still A Problem

- Writing correct multithreaded programs is hard!
- Programmers often forget to acquire the proper locks.
- This can lead to subtle bugs that produce observable error only on certain interleavings, making them hard to detect.

Automatic Dynamic Race Detection

- As the program executes, the detector builds a *happens-before graph* to capture the relative order of steps taken by different threads.
- This order is determined by the locking operations.
- For example, a `release lock` operation happens before the next `acquire lock` operation in the executed interleaving.
- The program may have many other interleavings with the same happens-before graph.
- A race condition between two memory accesses is present if there is no path between them in the graph, which means there exist other interleavings where the accesses occur opposite the observed order.

Happens-Before Graph & Shadow State



- In the happens-before graph on the left, line 2 happens before line 5.
- On the other hand, line 5 and line 7 form a race condition.
- For efficiency, a race detector represents the happens-before graph as the *shadow state* above on the right:
  - Clock vector C_x = <4,8> for memory location x after line 5 indicates that x was last accessed by Thread1 at time 4 and by Thread2 at time 8.
  - Clock vector C_2 = <4,9> for Thread2 after line 6 records at the second index Thread2's current time (9) and, at the first index, the time (4) of Thread1's last operation that happens before the current operation of Thread2
  - A race is detected at the write to x in Line 7, Thread1, by observing that the second clock in C_1 is smaller than the second clock in C_x.

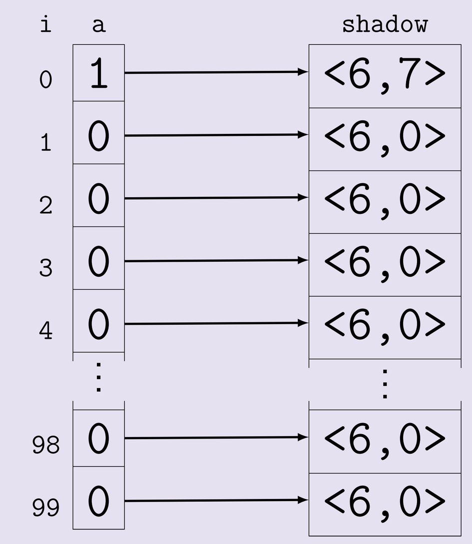
Problem: Lots o' State

- Unfortunately, the analysis above incurs a large space overhead because it allocates a shadow state object for *each memory location*.

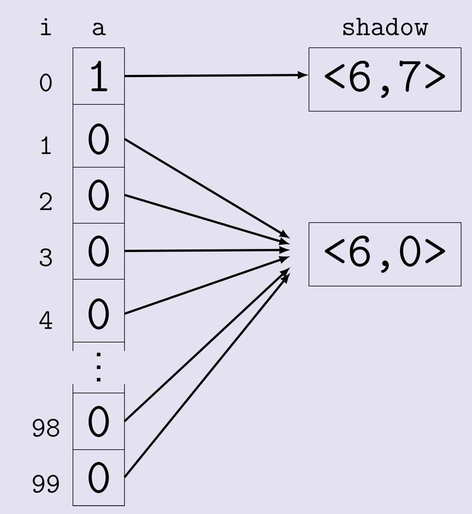
Our Work: Saving Space with Hash Consing

- Oftentimes many shadow state objects have the same value.
- The following program execution leaves array elements 1 through 99 with the same shadow state value.

```
Thread1:     Thread2:
 acquire lock
 for (i=0; i<100; i++)
   a[i] = 0
 release lock
              acquire lock
              a[0] = 1
              release lock
```

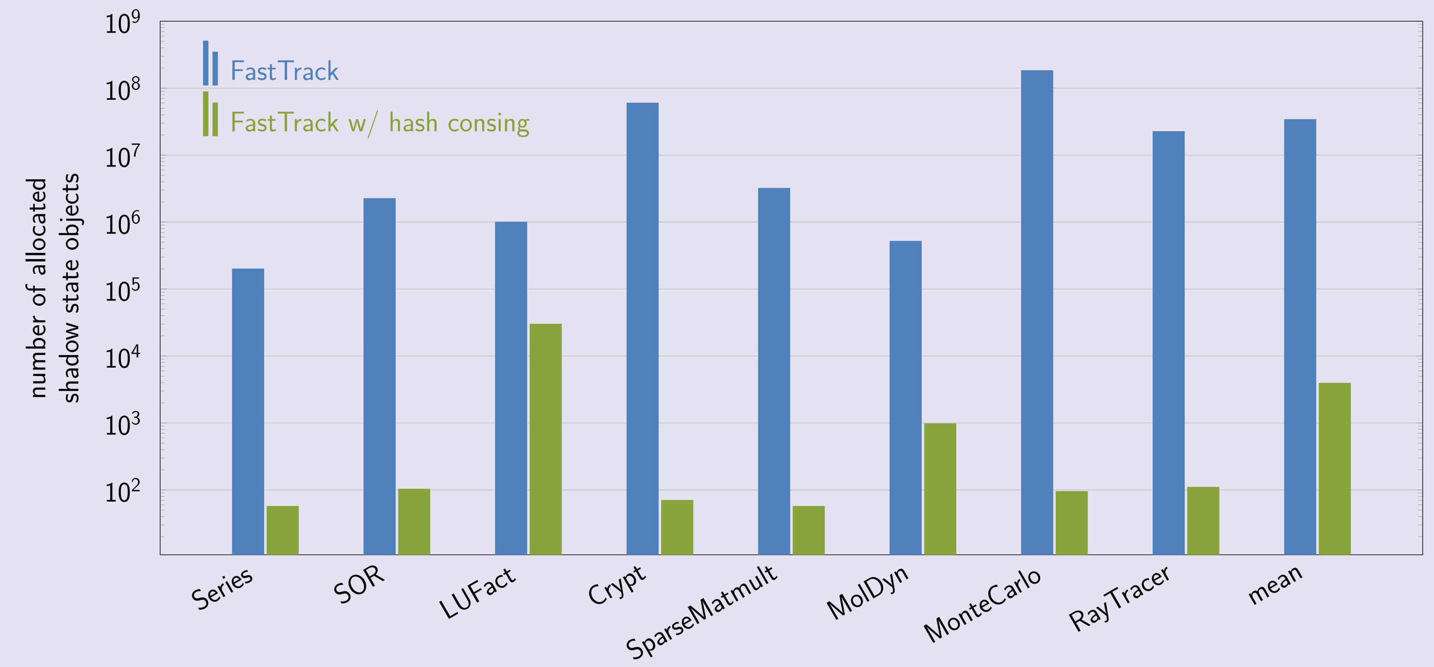


- Hash consing avoids creating multiple objects for the same value.
- Using a hash-consed shadow state greatly reduces our space overhead. The more shadow state values repeat, the more hash consing reduces our space overhead.
- We implemented hash consing in the FastTrack dynamic race detector.



Preliminary Results

- Tests on the Java Grande benchmarks show orders of magnitude reduction in the shadow state size.



Future Work

- Assess overall memory savings.
- Hash consing increases time overhead because it requires table lookups and copying of shadow state at each memory access. We plan to optimize our hash consing methods to address this.

This research was supported by NSF grant 1421051. Thanks to James Wilcox '13 for help with background material and the poster template.