Profiling Synchronization Patterns in Multithreaded Programs

Kasey Shen, Wellesley College || Adviser: Ben Wood, Wellesley College
Funding generously provided by: Sherman Fairchild Research Grant

1 Introduction

Multithreading is a model of concurrent software where multiple threads of execution share the same data. The parallel execution of threads allows for faster processing. However, there is always the possibility of multiple threads simultaneously accessing the same data, causing collisions.

Mutual exclusion locks and other synchronization mechanisms can help prevent some of the collisions. Yet, they rely on the programmer knowing where to use them, which is notoriously difficult. Various error detection tools find these problems but sacrifice accuracy or speed.

To optimize these tools [2], we can make informed changes based on profiled program behavior and choose the best tracking approach for each object.

2 Threads, Locks, and Race Conditions

Fig. 1 shows an example of bad interleaved code that results from multiple threads trying to access and update `votes` at the same time but doing steps at different times when the two statements need to occur together.

This simultaneous access results in an error for the total number of votes.

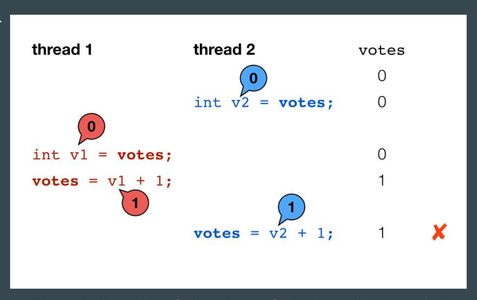

Figure 1: Race Condition Example in Vote Counters [1]

To prevent bad interleaving, programmers can use mutual exclusion locks to synchronize on the specific piece of datum.

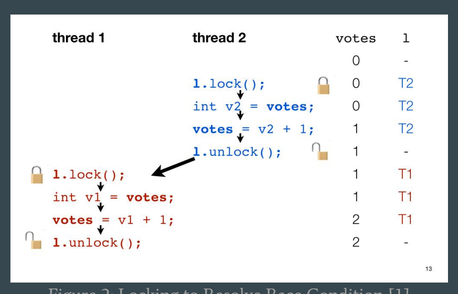

Figure 2: Locking to Resolve Race Condition [1]

Fig. 2 shows how by adding locks to atomic pieces of code, programmers can ensure certain statements are executed together and resolve the race condition between threads.

3

How can we make use of locking behavior in programs and inform our decision making when optimizing tools?

4 Implementation Example: Monitoring Opportunities During Execution

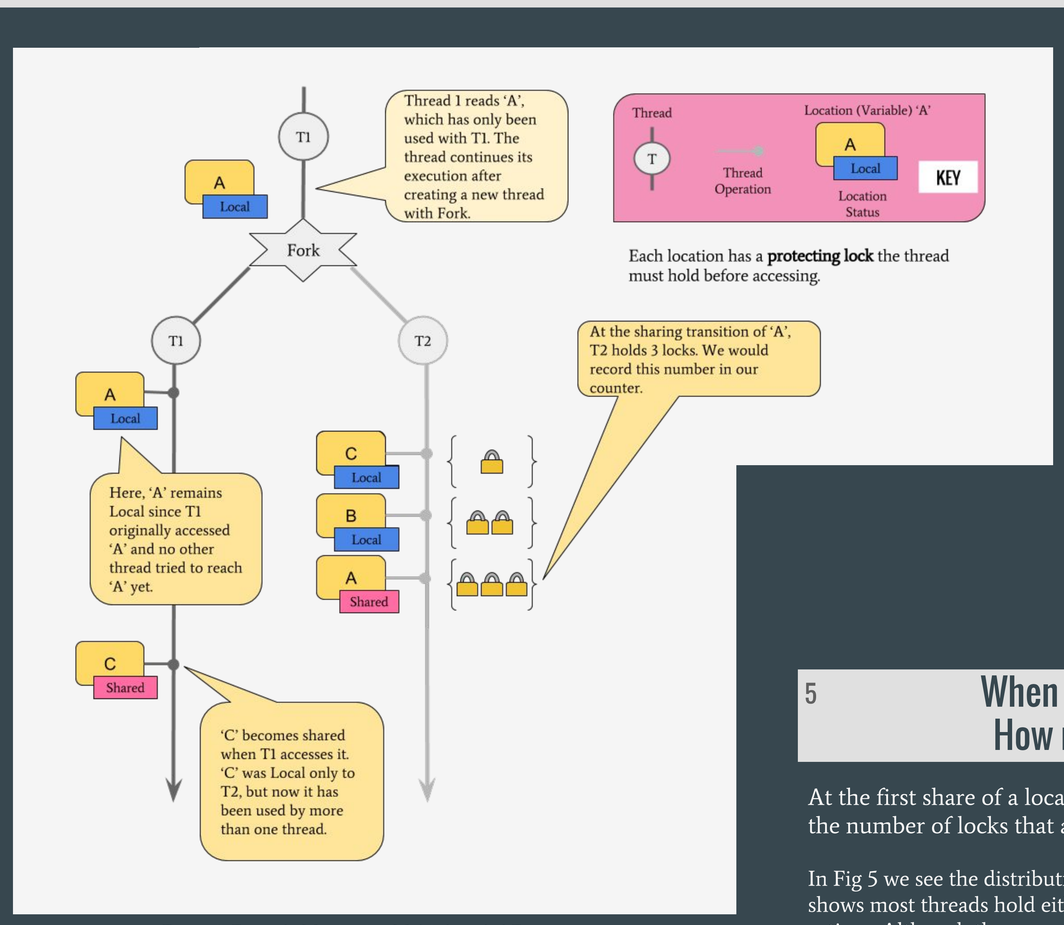

Figure 3: Example Monitoring Multi-threaded Execution

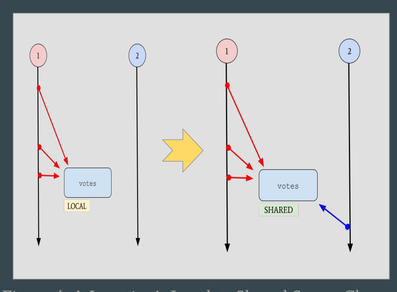

Figure 4: A Location's Local to Shared Status Change

Locations have either a status of local or shared. All locations start out as local and become shared when another thread accesses it for the first time.

5 When Data are First Shared, How many locks are held?

At the first share of a location, we noted the size of the thread's lockset, the number of locks that are being held by the thread making that access.

In Fig 5 we see the distribution of locks held across the benchmark programs shows most threads hold either 1 or 2 locks when making these "first share" actions. Although there are still some programs with 0 locks held, there are enough programs that have at least 1 lock being held.

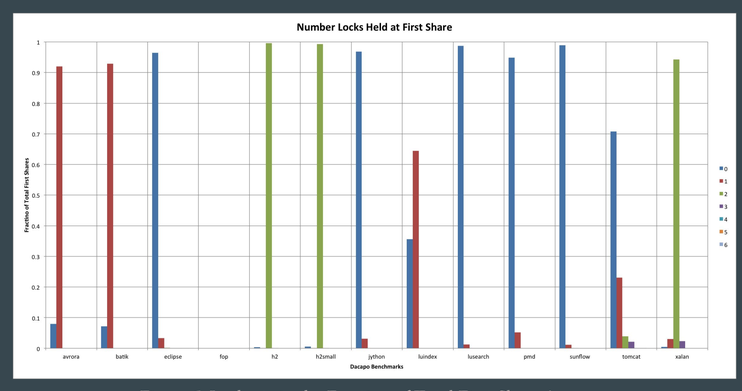

Figure 5: Lockset size by Fraction of Total First Share Accesses

6 How predictable are these locks?

To measure the predictability of a guarding lock within the initial lockset [1] recorded at first share, we continued to track the sets of locks held at subsequent accesses to each location.

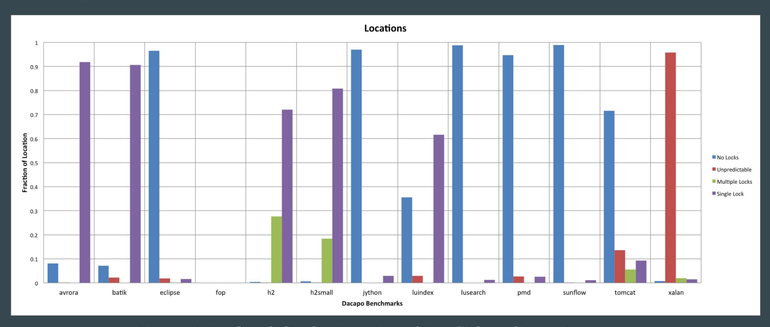

Figure 6: Predictability by Fraction of Total Shared Locations

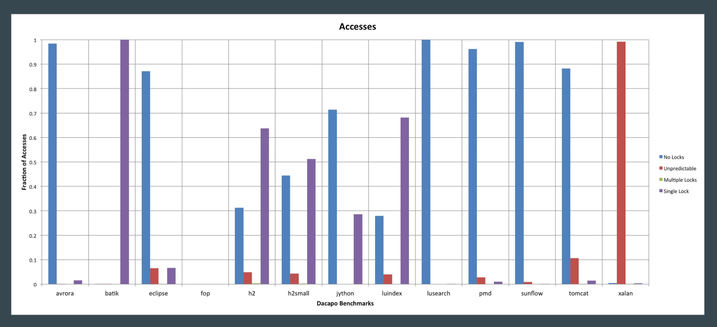

Figure 7: Predictability by Fraction of Total Shared Accesses

Our experiments on the Dacapo suite of Java benchmarks show these locksets follow 4 patterns. Each pattern has different implications for predictability. (Fig 6 & 8)

Single Lock - a single lock is held at first share and can be predicted
Multiple Locks - multiple locks are held at first share and can be predicted
No Locks - no locks are held at first share (data is likely read only)
Unpredictable - no locks are held consistently for all accesses

Our first set of results measures whether we can predict a protecting lock the first time a location is shared. The value of that prediction depends on how many times it will be accessed in the future.

To assess the potential impact, we measured the number of accesses to each of these 4 kinds of locations in the program. (Fig 7 & 9)

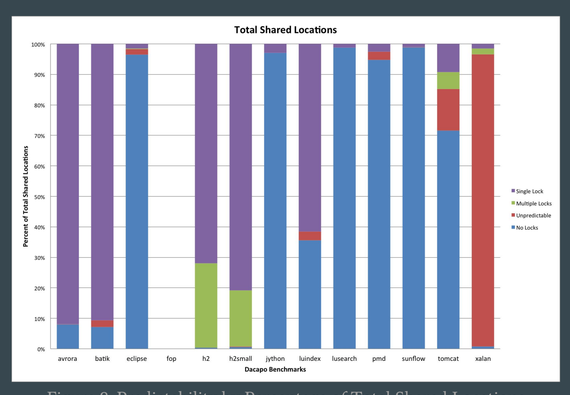

Figure 8: Predictability by Percentage of Total Shared Locations

Fig 8 and 9 show another way of viewing the distribution of types proportionate within a program.

In the total shared locations, there are 3 main patterns apparent (Fig 8): Single Lock, No Locks, and Unpredictable.

In total shared accesses, the 3 main patterns are also the same as shared locations but leaning more towards No Locks.

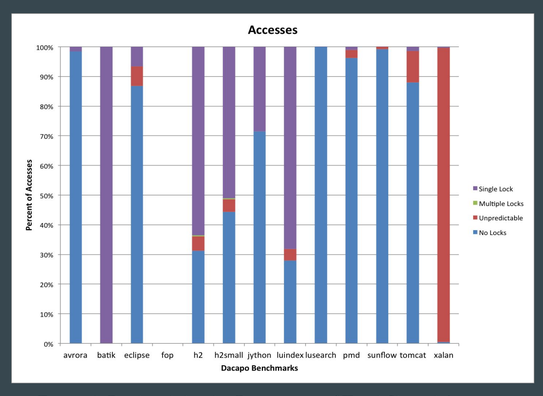

Figure 9: Predictability by Percentage of Total Shared Accesses

7 Conclusions & Future Applications

From profiling the sets of locks held by threads when they first share data, we observed that typically few locks are held at that point. This suggests we would be able to make accurate and potentially useful predictions about which lock consistently guards that data.

However, when we monitored the rest of the accesses to the locksets after that first share, we see that there are very mixed results in terms of useful predictability. Our profiling of predictability showed 4 distinct patterns.

In future work, we hope to exploit these patterns in bug detection tools that optimize for expected locking behavior.

8 References && Acknowledgements

[1]S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ToCS, 15(4), 1997.
[2]C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In PLDI, 2009.

I want to thank the Sherman Fairchild Foundation for their funding support. Additionally, thank you to my adviser Ben Wood for all the time, effort, and belief that I can do this and to Wellesley College for providing the opportunity.