Profiling Synchronization Patterns in Multithreaded Programs

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

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.

Threads, Locks, and Race Conditions

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

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.

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.

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)

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

Each location has a protecting lock that must be held before accessing. This simultaneous access results in an error for the total number of votes.

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

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

References & Acknowledgements


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