Verified Instrumentation for Dynamic Race Detection

Abstract

Writing race-free concurrent code is notoriously difficult, and races can result in bugs that are difficult to isolate and reproduce. Dynamic race detection is often used to catch races that cannot (easily) be detected statically. One approach to dynamic race detection is to instrument the potentially racy code with operations that store and compare metadata, where the metadata implements some known race detection algorithm (e.g. vector clock race detection). In this paper, we lay out an instrumentation pass for race detection in a simple language, and present a mechanized formal proof of its correctness: all races in a program will be caught by the instrumentation, and all races detected by the instrumentation are possible in the original program.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords dynamic race detection, interactive theorem proving

1. Introduction

Multicore processors have steadily invaded a broad swathe of the computing ecosystem in everything from datacenters to smart watches. Writing multithreaded code for such platforms can bring good performance but also a host of programmability challenges. One of the key challenges is dealing with data races, as the presence of a data race introduces non-sequentially-consistent [**?**] and in some cases undefined [**?**] behavior into programs, making program semantics very difficult to understand. There are many systems that provide sound and complete data race detection via dynamic analysis [**? ? ?**] to help programmers detect and remove data races from their programs.

While these analyses have been proven correct, such proofs have two main shortcomings: they are proofs of the algorithms, instead of the implementations, and they are paper proofs instead of machine-checked proofs. Because of these shortcomings, it is possible that a race detector does not faithfully implement its algorithm, or that the algorithm itself is not fully correct. Indeed, our verified approach reveals a small issue in the paper proof from [**?**] that, while fixable, illustrates the potential dangers that stem from best-effort proofs and implementations.

Our approach seeks to rectify these concerns and place dynamic race detection on a provably-correct foundation by providing the first fully-verified implementation of vector clock race detection. We adopt a two-level approach, first verifying the high-level algorithm and then verifying that an instrumentation pass for a simple language faithfully implements the algorithm and preserves the semantics of the original, uninstrumented program. We believe that our approach is useful for verifying race detection algorithms and implementations, which we show by verifying the correctness of the FastTrack [**?**] algorithm. Crucially, our verification of FastTrack is achieved by proving it equivalent to the simpler vector clock race detection algorithm, which is more efficient than proving FastTrack's correctness directly. To the best of our knowledge, ours is the first scheme to adopt formal verification for either race detection algorithms *or* their implementation. Moreover, we believe our general approach may be useful for verifying a broad range of dynamic analyses, helping to ensure that debugging tools are themselves free from bugs.

[TODO: need more technical nuggets here, and/or a figure?]

This paper makes the following contributions:

- We give the first mechanized proofs of correctness of vector clock race detection and the FastTrack algorithm
- We discovered an issue in the paper proof of correctness for FastTrack that we repair in our mechanized proof
- We describe an instrumentation pass that has been proved to implement vector clock race detection for a simple language
- We present a method for verifying race detection instrumentation (the deep spec approach)

2. Proof Strategy

Our approach involves two levels: verifying the high-level algorithm, and proving that an instrumentation pass implements the algorithm.

3. Race Detection Algorithms

3.1 Vector Clock Race Detection

definition

verification

3.2 FastTrack

definition

verification

And we found a bug.

4. Instrumenting a Simple Language

4.1 The Language

We define a simple multithreaded language that is just complicated enough to have races and implement race detection instrumentation. The instructions of the language are defined as follows, where n is a natural number, a is a local variable, $e, e1, e2$ are expressions, p is a pointer, l is a lock, t is a thread id, and i_j are instructions:

$$\mathit{expr} ::= n \mid a \mid e_1 + e_2 \mid \texttt{max}(e_1, e_2)$$

$$\begin{aligned}\mathit{instr} ::= {} & a \texttt{ := } e \mid a \texttt{ := load } p \mid \texttt{store } e\ p \mid \texttt{lock } l \\ & \mid \texttt{unlock } l \mid \texttt{spawn } t\ i_1, ..., i_n \mid \texttt{wait } t \\ & \mid \texttt{assert}(e_1 \texttt{ <= } e_2)\end{aligned}$$

A *program* is simply a list of instructions. The dynamic state of a program contains a collection of threads, where a thread is a pair (t, li) of a tid and a list of instructions, and a family of local environments G_t, one for each thread, assigning values to local variables. We give semantics to the language via a transition system of the form $(P, G) \xrightarrow{o,c} (P', G')$, where P, P' are collections of threads, G, G' are environments, o is a race detection operation (if one is produced by the step), and c is a concurrent memory operation (likewise). We define a function eval that takes a local environment and an expression and yields the value of the expression. The semantics of the language are shown in Figure 1.

$$\text{assign}\ \frac{}{(e_1 \ldots (t, a \texttt{ := } e\ li) \ldots e_k, G) \to (e_1 \ldots (t, li) \ldots e_2, G[G_t[a \mapsto \text{eval}(G_t, e)]])}$$

Figure 1. Semantics of the simple language

4.2 Instrumentation

4.3 Necessary Synchronization

Completely unsynchronized instrumentation cannot be sound and complete. (example) At the same time, adding too much synchronization could significantly hurt performance.

5. Verifying Instrumentation

We verify the correctness of the instrumentation by showing that instrumentation records the same information and performs the same checks as the algorithm would perform on any program.

lemmas relating each instrumentation segment to vector clock operations

simulation relation and clear statements of two directions

6. Related Work

[TODO: discuss verified instrumentation like SoftBound]

There is a rich history of systems that provide dynamic data race detection, dating back to the original proposals of the vector clock algorithm [**? ? ?**]. Subsequent systems have implemented vector clocks, with various optimizations, using dynamic analysis to provide race detection for C/C++ [**? ?**] and Java [**? ? ? ? ?**] programs. These systems have relied on paper proofs to demonstrate correctness, but these proofs have not been mechanically verified until our work. Furthermore, these paper proofs operate at a high level of abstraction, using an operational semantics that elides important details such as the synchronization used within the race detector itself. Thus, the implementations of these algorithms are far removed from the algorithms themselves, leaving the door open for bugs.

Lockset-based race detection [**? ?**], an alternative to the traditional vector clock data race detection algorithm, reports false races on some common programming idioms like privatization but can also detect with a single execution some races that would require multiple executions to detect with vector clocks. Other work has generalized vector clock race detection to detect more races from a single execution [**? ? ?**] at the cost of decreased performance.

Several forms of sampling-based dynamic race detection have been proposed. Such schemes trade soundness [**? ? ? ? ?**] for reduced performance overheads.

There have been several proposals for static race detection [**? ?**] or static analysis [**? ?**] to prune race detection instrumentation and metadata at compile time, which can serve as a complement to our dynamic approach. Others have proposed type systems [**? ? ?**] and implicitly-parallel languages [**? ?**] that eliminate data races by construction, though these systems sacrifice expressiveness to obtain race-freedom guarantees.

Several systems exist for detecting data races in structured parallel programs such as fork-join programs [**? ? ? ?**], async-finish programs [**?**] or programs with asynchronous callbacks [**? ? ? ?**]. Structured parallelism admits more time- and space-efficient data race detection than the general multithreaded programs we support. None of these prior algorithms or implementations have been formally verified, however, so they represent a potentially fruitful area for future work.

7. Conclusions and Future Work

verified FastTrack instrumentation

more faithful/detailed implementation

relaxed memory, if it matters

other kinds of instrumentation, e.g. atomicity violation monitoring

References