Identifying the potential for race conditions by analyzing thread behavior

Elliot Weiser (eweiser1) and Stella Cho (scho1)

1 Abstract

Multithreaded software is everywhere; the advantages of parallelism are applied to almost all modern day applications. However, the common parallel primitive known as threads can introduce bugs known as race conditions. This research is a part of an attempt to reduce the likelihood of these bugs, which is particularly important today because modern software is often written by several developers, at different times, without full knowledge of what the other code-writers are doing. Our objective is to examine existing multithreaded software to observe the behavior of threads within the software. In particular, we expect to identify and quantify potentially problematic threads that could be made more resistant to concurrency bugs with a superior implementation. This research is an important step aimed toward aiding those who are working on projects to reduce parallel-code bugs.

2 Motivation

Threads are an effective approach to parallel computation and increased responsiveness of interactive programs. However, they are notoriously prone to bugs because they share certain regions of memory, among other things. Race conditions can cause undefined behavior in a program, which may cause it to terminate, or continue execution with an incorrect or unintended state (which may be worse). Our project examines thread behavior with the overall goal of identifying and limiting these bugs.

3 Background

Threads have various uses: *Computation* threads satisfy the conventional need for parallelism. These are often used to divide up equal amounts of disjoint work across several processors (usually the number of cores in the machine). Such multithreaded software is typically written with data races in mind, and so mutexes and semaphores are used appropriately and the race condition is removed. Event-handlers are different. *Event-handlers* are routines that execute upon the

realization that some "event" has occurred (i.e. they are told to execute when some condition has been met). These routines can be written after the bulk of the application has already been written. Consequently, the writer of this routine may have no idea that he/she is introducing a race condition, and thus, the program has potentially bug-infested behavior.

Charcoal is a project created by Ben Ylvisaker that is geared toward solving problems related to multithreaded code. Specifically, Charcoal is a dialect of C that introduces *activities*, a variation on cooperative threads that limits the possibilities of concurrency bugs. Unlike traditional threads, only one activity can run at a time, although it is possible to "yield" control to a different activity. Ylvisaker claims that activities are better suited than threads for event-handling by reducing potential race conditions. [2]

Related Work: Blake *et al.* (2010) study how effectively modern desktop applications use threads on multi-core architectures. The paper explicitly looks at context switches and GPU utilization and determines 2-3 cores to be sufficient for most desktop applications. [1]

4 Our Idea

We will determine the kinds of threads that a given piece of software uses. In particular, we will measure the number of threads used, how long they run, and what they do. We expect the threads to fall into two functional categories: computation threads, and event-handler threads. As activities are expected to have the most impact on event-handler threads, which have short and unpredictable behavior, the thread-type composition will inform the usefulness of an activity-centric implementation.

We will follow closely the experimental set-up described in Blake *et al.* and look into *DTrace*, a dynamic tracing framework that reports when a thread is created, destroyed, started, or stopped. We will analyze the threads created by different pieces of widely used, multithreaded software.

5 Milestone Goals

5.1 Milestone 1 (10/30, 11/6)

Have working knowledge of DTrace and preliminary DTrace results. Write a program that scans DTrace output to parse for information that we want.

5.2 Milestone 2 (11/13, 11/20)

Write a program that performs analytics on results.

5.3 Milestone 3 (11/27, 12/4)

Run DTrace on other pieces of software. Compile statistics.

References

[1] Geoffrey Blake, Ronald Dreslinski, and Trevor Mudge. Evolution of thread-level parallelism in desktop applications. *ISCA'10*, 2010.

[2] Ben Ylvisaker. Charcoal: Easier concurrency for application developers, 2013.