Finding Concurrency Errors with GrammaTech Static Analysis

Introduction

Although decades of advances in miniaturization have yielded enormous performance gains for single processors, it now appears that this era is coming to a close. The industry has placed a big bet on future single-chip performance gains coming from increasing core counts, but this will only be a winning wager if software can be programmed to take advantage of parallel processors. Unfortunately, concurrent programming is difficult. Even experts familiar only with single-threaded programming often fail to appreciate that concurrent programs are susceptible to entirely new classes of defect, such as data races, deadlocks, and starvation. Avoiding these pitfalls requires deep reasoning about concurrency, which is difficult for humans, and is not made easier by mainstream programming languages that were not designed for concurrency. Consequently, these hazards frequently trip up even highly experienced programmers. In one case, a race condition (now fixed) in iOS 4.0 through 4.1 meant that any person with physical access to an iPhone 3G or later could bypass its passcode lock under certain conditions[1].

Concurrent programs and their problems have been with us for much longer than multi-processor machines. However, concurrency defects of all kinds are much more likely to manifest on multi-processor (including multi-core) computers. On single-processor systems, threads typically run uninterrupted for reasonably large time quanta, and there is no truly simultaneous execution, which dramatically constrains the set of likely behaviors. In practice, concurrent programs that run perfectly well on single-processor systems often manifest previously latent defects when run on multi-processor systems.

This paper describes common concurrency pitfalls and explains how static analysis with GrammaTech CodeSonar® can help find such defects without executing the program. CodeSonar ships with a range of checks for problems that can arise in concurrent programs. For example, it includes an innovative Data Race analysis that is paired with user interface functionality for understanding the interactions between different program threads. In addition to the included checkers, an extension API is provided, enabling users to add their own checks for software defects.

Throughout this document, CodeSonar warning class names are rendered in italic, sans-serif font: Null Pointer Dereference. If a warning class is disabled by default, the class name is marked with an asterisk: Recursive Macro*.

[1] http://www.cvedetails.com/cve/CVE-2010-4012/

Multithreading Complicates Development

When multiple operations can execute concurrently, everything becomes more complicated. One of the most significant complications is that instructions in multiple threads can be *interleaved*. The number of possible interleavings can be huge, and increases enormously as the number of instructions grows, a phenomenon known as the *combinatorial explosion*. If thread A executes M instructions and thread B executes N instructions, there are $\binom{N+M}{N}$ possible interleavings of the two threads. Even the smallest threads have many possible interleavings – see Figure 1 for some examples. The number of possible interleavings for a pair of threads with twelve instructions each exceeds two million. In practice, the actual number of interleavings is reduced somewhat by constraints on execution order imposed by synchronization operations and the system scheduler. Even with such constraints, real concurrent programs have astronomical numbers of legal interleavings.

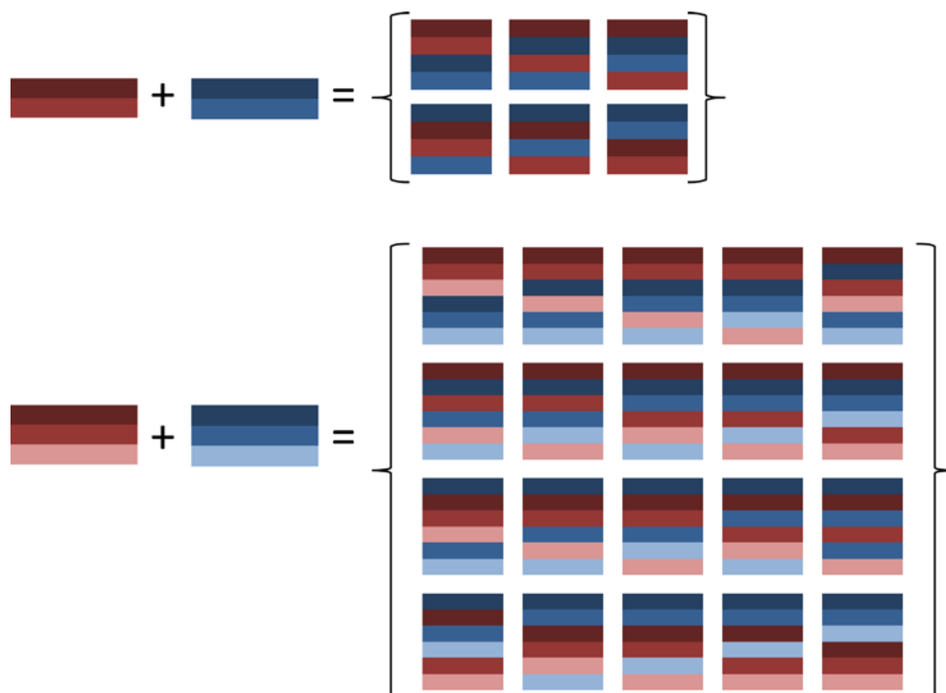

Figure 1. There are six possible interleavings of two threads with two instructions each. With three instructions each, there are twenty possible interleavings.

Real-world software involves many more instructions than this, and tends to involve branching and other complications. Testing every interleaving is completely infeasible. To make matters worse, interleaving decisions are made by the system scheduler, with very little control given to the programmer or end-user. Not only is it infeasible to test *every* interleaving, it is very difficult to enforce *any* specific interleaving on a given execution. System scheduling is enormously complex, to the point of being effectively nondeterministic. This is especially true of deployed software that will run in environments outside the control of developers.

The nondeterminism in multithreaded programs makes traditional software testing significantly less effective. In principle a test can be written to reliably expose any defect in a

single-threaded program, though generating a complete set of tests using standard methods can be incredibly expensive. For multithreaded programs this is *not* the case in general, because nondeterministic interleaving means that a single test can have many different behaviors, and it's difficult or even impossible to force a particular behavior to occur on a given run of the program.

Nondeterminism also reduces the generalizability of Proven In Use arguments. For single-threaded programs, a long track record of safe operation is a strong indication that the software will continue operating correctly, even if its environment is changed. For multithreaded programs, even tiny changes in the operating environment can lead to wildly different behavior, because previously unexercised interleavings occur. The nearly unbounded number of possible interleavings means that even aggressive stress testing is not necessarily an effective way to expose concurrency defects.

CodeSonar can discover software defects without exhaustively exploring interleavings. It thus plays an important role in multithreaded software verification: one which testing alone cannot fill. It uses sophisticated symbolic execution techniques to reason about many possible execution paths and interleavings at once. These techniques find concurrency errors without executing the program at all.

The consequences of interleaving stretch far beyond testing. The interleaved threads can actually affect each other's behavior. Ideally these effects are intentional and correct, but in practice they sometimes they involve *race conditions* – a class of problem that does not even exist in a single-threaded environment. The community has devoted extensive effort toward developing techniques to eliminate these ill effects. Among the most frequently used are locks, semaphores, and message passing. Unfortunately, these techniques introduce problems of their own. They increase the size and complexity of the code base, making it harder for human readers to understand. They can be used incorrectly in ways that lead to processes or even entire programs failing to make progress. They can cause needless slowdown when used unnecessarily, and fail to provide protection when accidentally omitted. CodeSonar's static analysis provides important assistance in identifying and addressing these kinds of issues.

A single-threaded worldview remains pervasive in software development, even in projects that have made the transition to multithreading. In some cases, developers do not think about multithreading at all. Other developers may be aware of multithreading and its attendant hazards, but treat artifacts like semaphores, thread-safe libraries and the `volatile` keyword as magical talismans for warding off concurrency bugs. Even experts usually do not have a sufficiently holistic understanding of the system to reliably spot concurrency defects.

Many development practices implicitly consider threads individually and so do not account for all the issues that can arise when several threads execute simultaneously. Unit testing, for example, will not uncover all bugs caused by the interaction of multiple threads. Similarly, running multithreaded programs in a debugger is not an effective way to diagnose concurrency problems, because the debugger itself disrupts the operating environment, and only one interleaving can be explored per execution. Such practices remain appropriate and

valuable for addressing "classical" software defects, but must be augmented with techniques that take into account the true multithreaded nature of the system.

Development tools are also sometimes fundamentally based on a single-threaded worldview. For example, compiler optimizations are often based on the assumption that if the current thread does not modify a value, the value remains unmodified: if more than one thread is running at once, this is not necessarily true. The definitions of C and C++ did not include any reference to multithreading until the most recent (2011) revisions of their respective standards. Compiler and runtime support for these standards is still incomplete, which means that multithreaded programs are exposed to implementation-specific behavior to a much greater extent than single-threaded programs. As Boehm [2] puts it, "essentially any application must rely on implementation-defined behavior for its correctness". Eliminating all potential concurrency defects like data races and deadlocks is a good way to avoid bad implementation-specific behaviors.

In the remainder of this paper we describe software defect classes that are specific to multithreaded programs, and how CodeSonar can be used to find these defects and reduce the probability of their occurrence.

Data Race

A *data race* arises when multiple threads of execution access a shared piece of data, and at least one of them changes the value of that data, without an explicit synchronization operation to separate the accesses. Depending on the interleaving of the two threads, the system can be left in an inconsistent state. Data races are especially insidious because they can lurk undetected indefinitely and only show up in rare circumstances with mysterious symptoms that are difficult to diagnose and reproduce. In particular, they are a common source of errors in (well-tested) deployed software. At best, the presence of data races means increased development times; at worst the consequences can be devastating. A data race in a computerized Energy Management System dramatically worsened the 2003 Northeast blackout by causing delayed and misleading information to be communicated to the operators [4]. In an article titled *Tracking the blackout bug* [3], Kevin Poulsen notes that "[t]he bug had a window of opportunity measured in milliseconds." The chances of a problem like this manifesting during testing are infinitesimal.

A simple data race example is shown in Figure 2. A manufacturing assembly line has entry and exit sensors, and maintains a running count of the items currently on the line. The entry sensor controller increments the count every time an item enters the line, and the exit sensor controller decrements it every time an item reaches the end of the line. If an item enters the line at the same time that another item exits, the count should be incremented and then decremented (or vice-versa) for a net change of zero. However, computers implement increment and decrement as a sequence of simpler operations that first load the value from memory, then modify it locally, and finally store it back to memory. If the updating transactions are processed in a multithreaded system without sufficient safeguards, a data race can arise because the controllers read and write a shared piece of data: the count. The interleaving in Figure 2 results in an incorrect count of 69. There are also interleavings that result in an incorrect count of 71, as well as a number that correctly result in a count of 70.

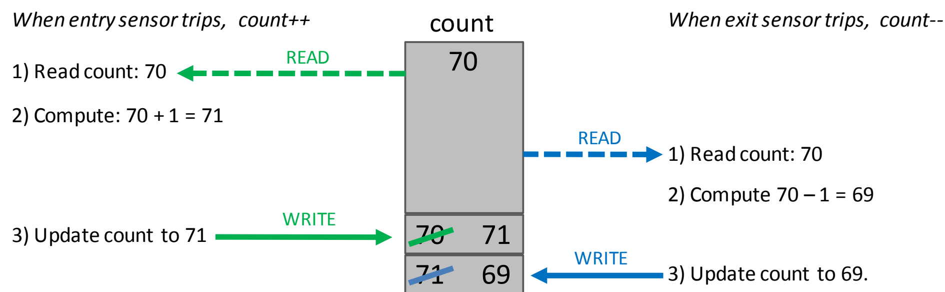

Figure 2. Data race leads to incorrect count of items on an assembly line.

Data races like this are difficult to eliminate for several reasons:

Rare occurrence means little chance of even noticing that there is a problem. If the problem manifests infrequently, it may never show up during testing. As noted above, the number of possible interleavings of two processes is enormous and interleaving decisions are not under user control. Testing every interleaving is simply impossible, and even if testers identify a small number of interleavings that merit inspection they will generally not have the means to enforce test executions with those interleavings.

Data race diagnosis is difficult. Firstly, the symptoms can be perplexing. In the Figure 2 example, the running count will (probably) usually be correct, but sometimes too high and other times too low. Secondly, programmers unaccustomed to considering the particular pitfalls of multithreaded programming may spend a lot of time puzzling over the code before the possibility of a data race occurs to them. The effects of data races often seem impossible when the symptomatic code is considered in isolation; this sometimes leads developers to discard data-race-related bug reports as unreproducible. CodeSonar's static analysis is especially helpful in this regard. It identifies data races by examining patterns of access to shared memory locations – that is, it focuses on the causes, not the symptoms. When a data race is identified, CodeSonar issues a Data Race warning that includes supporting information to aid the user in evaluation and debugging. The need for a developer to work backwards from a particular symptom is eliminated, which reduces the overall debugging burden.

We note here that CodeSonar also provides a File System Race Condition check. This is a different form of data race vulnerability in which a program calls a function that *checks* a named file and then later calls a function that *uses* the same named file. The source code assumes the file is the same at both times, when in fact another process may have changed the file between the 'check' and 'use'. For example, an attacker could replace the original file with a link to a file containing confidential data.

Eliminating data races can introduce new problems. Data races are typically avoided by using locks or other synchronization techniques to protect shared resources. However these can introduce performance bottlenecks that might prevent the program from taking advantage of the full potential of multiple cores, so programmers must exercise care in using them. In the worst case, they can lead to a different set of problems, namely *deadlock* and *starvation*.

Deadlock

In a *deadlock*, two or more threads prevent each other from making progress by each holding a lock needed by another. Figure 3 shows how a deadlock can arise with two locks used to protect two shared variables. In this example there are multiple assembly lines that share a `count` of the total number of items currently under assembly and a second `bad_items` value recording how many finished items have failed quality control. One thread acquires the lock on `count`, another acquires the lock on `bad_items`. Neither thread can now obtain the second lock it needs, so neither can carry out its operations, and so neither will get to the point where it will release its lock. Neither update can be completed, and both threads are completely stuck.

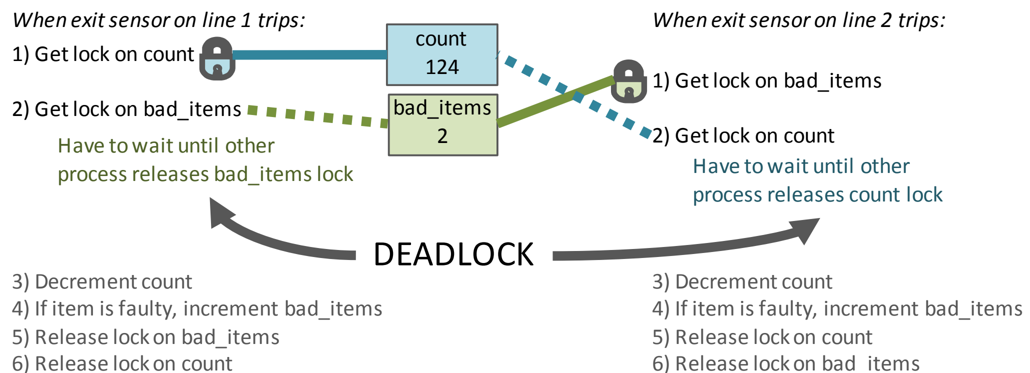

Figure 3. Deadlock between two threads: neither can progress.

CodeSonar can help identify software at risk of deadlock by issuing Conflicting Lock Order warnings if the same locks can be acquired in different orders by different threads: the example in Figure 3 has this property. Eliminating all such cases is sufficient to ensure that the system cannot become deadlocked.

The Nested Locks check is even more aggressive: a warning is triggered whenever a thread tries to obtain two or more locks. If each thread can only hold one lock at a time then deadlocks cannot arise. However, completely eliminating all lock nesting is an ideal that many real projects cannot attain: some users will disable this check and enable only Conflicting Lock Order.

Even though either of these restrictions on locks is sufficient to eliminate deadlock, process starvation can still occur.

Process Starvation

A thread *starves* if it is waiting for a lock that is held by another thread for a very long time. The most common instances of this problem involve the lock-holding thread waiting for an event like a large disk read or the arrival of data from the network. Suppose our example manufacturing automation system includes a regular audit thread that examines all entry and exit records to ensure that the running count matches total items entering less total items exiting. The audit thread needs to hold locks on the count and on all sensors, so all updates

must wait for the audit to finish. If the audit runs for a long time, updates can be significantly delayed. If it runs for too long, the next audit may manage to acquire all the locks and start running before the outstanding thread can make any progress. In the worst case, some or all of the updates may *never* have the opportunity to run.

Static analysis can help find starvation problems by examining the set of all procedures called by a thread that holds some lock. CodeSonar users can add custom checks of this form. For example, if there is a function `f()` that can block or is known to have a long running time, engineers can add a custom check that triggers a warning whenever `f()` is called by a thread that holds one or more locks.

Correct Use of Synchronization Techniques

It can be tricky to write code that uses synchronization techniques effectively, and coding standards often impose restrictions on which techniques can be used and under which conditions. For example, the *JPL Institutional Coding Standard for the C Programming Language* [1] does not permit task delay functions to be used for task synchronization. CodeSonar includes a suite of checks for the JPL coding standard, including a Task Delay Function check that issues warnings at any use of a function that has been identified as having this purpose. A configuration parameter allows users to extend the list of known task delay functions as required. More generally, users can use the `BADFUNC_*` family of configuration parameters to extend the CodeSonar analysis by specifying forbidden synchronization functions whose use should trigger a warning. (The `BADFUNC_*` parameters are also useful for identifying functions – especially those in third-party code – that are or may be thread-unsafe. CodeSonar's built in Use of ttyname* check has this motivation.)

CodeSonar is also ideally suited to identifying potentially risky *patterns* of synchronization function usage.

- Unknown Lock: a lock or unlock operation refers to a lock that cannot be identified.
- Missing Lock, Missing Unlock, Lock/Unlock Mismatch: an unlock (lock) operation in the body of some function does not have a corresponding lock (unlock) operation in the same function. This does not necessarily mean that the matching operation is not carried out, but keeping the lock and unlock operations in the same function ensures that the program is more human-readable, and thus easier to maintain.
- Double Lock, Double Unlock: the same resource is locked (unlocked) multiple times, which can have adverse effects on the resource or the locking infrastructure. Even if these effects are not experienced for a particular implementation, the doubled operation may indicate the existence of a previously-unconsidered execution path.
- Try-lock that will never succeed: indicates a redundant and possibly misleading try-lock operation.

Users can add their own checks for risky usage patterns with the CodeSonar Extension API. Such checks could be based on local coding rules, or on the particular synchronization techniques used in a given project.

Conclusion

Multithreading adds entirely new classes of potential bugs to those that must be considered by developers. At the same time, the nondeterminism and sheer number of possibilities introduced by thread interleaving make it significantly more difficult to find bugs in multithreaded systems by testing and other traditional methods.

The static analysis provided by GrammaTech CodeSonar supports development teams in addressing both of these issues. It provides checking and reporting for a range of concurrency-related problems without the limitations experienced by execution-based techniques or the oversimplification imposed by a single-threaded point of view.

To learn more about CodeSonar, and for a free trial, contact GrammaTech.

About GrammaTech

GrammaTech was founded by computer science professors Tom Reps of the University of Wisconsin and Tim Teitelbaum of Cornell University. The staff includes thirteen Ph.D.-level experts in static analysis. Leading organizations such as Lockheed Martin, Northrop Grumman, BAE Systems, GE Aviation, LG Electronics, Samsung, Qualcomm, Panasonic, Kawasaki, the FDA, and NASA use GrammaTech CodeSonar and CodeSurfer to improve software quality and reduce costs.

Reference List

1. *JPL Institutional Coding Standard for the C Programming Language*, 2009, Jet Propulsion Laboratory, California Institute of Technology JPL DOCID D-60411.
2. Boehm, H.-J., *Threads Cannot be Implemented as a Library*. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. Chicago, IL: ACM. pp. 261-268.
3. Kevin Poulsen, *Tracking the blackout bug*. in *SecurityFocus*. April 7, 2004.
4. U.S.-Canada Power System Outage Task Force, *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations*. 2004.