Deadlock checking by data race detection

Ka I Pun[1], Martin Steffen[1] and Volker Stolz[1,2]

[1] Department of Informatics, University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao

Motivation

In concurrent programs, locks are commonly used to avoid simultaneous access to shared resources. A deadlock occurs when multiple processes wait for locks in a cycle. The competition between these processes for the access to shared resources can be interpreted as the *race* of the last two processes to close the deadlock cycle.

Instead of developing/using a custom deadlock checker, we can thus use existing race checkers (e.g. `Chord` [2] and `Goblint` [5]) for *static* race detection after instrumenting a program with suitable shared variable accesses.

To facilitate the instrumentation, we present a type system which captures so-called *second lock points*, a static over-approximation of program points where deadlocks can actually manifest themselves. This type-information is used to transform the program into its corresponding instrumented version with conflicting accesses to race variables. We show the soundness of our approach, which is that for a program with a deadlock, a race analysis will report a race on the transformed program, and that the transformation preserves deadlocks.

Type System and Transformation

Our type system algorithmically tracks which locks are actually handled in the interactions by annotating each lock with the corresponding program point π of its creation. Furthermore, it tracks the relative change to the lock count, denoted as $\Delta \to \Delta'$, through each statement. The type system uses constraints [1, 3] to derive the *smallest* possible type (in terms of originating locations) for each variable of lock-type in the program.

The instrumentation transforms the program by comparing the static analysis information with the desired cycle. The cycle $\Delta_c$is expressed in terms of a "ring" of (abstract) processes, each one *holding* a particular lock and *requesting* another one. This corresponds to a deadlock/deadlocked configuration [4] with a heap σ and processes P where every involved process p has $waits(\sigma \vdash P, p, l)$ is blocked on taking l. (Such a deadlock can occur in various places in a program.)

We call a program location a static second lock point (SLP) for Δ_c, where the analysis information *from the type* indicates that both the lock "we" are trying to take is involved in the cycle, and we already hold the corresponding other lock in Δ. As due to the over-approximation of locations a lock-statement can refer to a set of possible lock locations, instrumentation must occur if *any* of the potential locks is involved in the cycle.

The judgement of the type system is given as

$$\Gamma \vdash e : T :: \Delta \to \Delta'; C$$

which means that the expression e has type T and has the relative change to lock counts from Δ to Δ' under the constraint C.

Below, we show some of the relevant typing rules of our system for a functional language with locks. Lock creation in rule T-NEWL introduces an abstract location which is henceforth tracked in the constraints.

Spawning a thread in rule T-SPAWN has no effect on the caller, and the premise of the rule checks well-typedness of the expression being spawned. Note that for that expression, since it will be executed in a new thread, all locks are assumed to be initially free (indicated by $\bullet$).

Rules T-LOCK and T-UNLOCK describe the operations of locking and unlocking, simply counting up, resp. down the lock counter, setting the post-condition to $\Delta \oplus \rho$, resp. $\Delta \ominus \rho$ where $\oplus$ and $\ominus$ record the change for the particular abstract lock. Note that the constraints can be solved *after* type checking and *before* calculating the Δs.

$$\frac{\rho \textit{ fresh}}{\Gamma \vdash \texttt{new}_\pi\ \mathrm{L} : \mathrm{L}^\rho :: \Delta \to \Delta; \rho \supseteq \{\pi\}}\ \text{T-NEWL} \qquad \frac{\Gamma \vdash e : \hat{T} :: \bullet \to \Delta_2; C}{\Gamma \vdash \texttt{spawn}\ e : \texttt{Thread} :: \Delta_1 \to \Delta_1; C}\ \text{T-SPAWN}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^\rho :: \Delta_1 \to \Delta_1; \emptyset \qquad \Delta_2 = \Delta_1 \oplus \rho}{\Gamma \vdash v.\ \texttt{lock} : \mathrm{L}^\rho :: \Delta_1 \to \Delta_2; \emptyset}\ \text{T-LOCK} \qquad \frac{\Gamma \vdash v : \mathrm{L}^\rho :: \Delta_1 \to \Delta_1; \emptyset \qquad \Delta_2 = \Delta_1 \ominus \rho}{\Gamma \vdash v.\ \texttt{unlock} : \mathrm{L}^\rho :: \Delta_1 \to \Delta_2; \emptyset}\ \text{T-UNLOCK}$$

Whether or not to prepend a race-variable to a single `lock`-statement depends on if it is a second lock point or not.

The number of possible deadlocks does not depend on the number of abstract lock locations occurring in the program: even a single location, i.e. all locks stem from the same `new`-statement, is sufficient to form deadlocks of arbitrary length.

The number of introduced race variables increases (by one) for each cycle that we would like to check for. It is possible to instrument a program for a set of potential cycles without the different race variables interfering with each other. Alternatively, for scalability it is possible to e.g. parallelize race checker-runs on programs instrumented for different (sub-sets of) cycles.

We illustrate our approach using the dining philosophers:

```
let l_1 = new^{π_1} L; ...; l_n = new^{π_n} L // create all locks
    phil = fun F(x,y) . ( x.lock; y.lock;
                          /* think */
                          y.unlock; x.unlock; F(x,y) )
in spawn(phil(l_1,l_2)); ... ; spawn(phil(l_n,l_1))
```

As *type* for the philosopher-function, we obtain that each of the two arguments of lock-type has type $\mathrm{L}^{\{\pi_1,\ldots,\pi_n\}}$, i.e., they may origin from *any* of the `new`-statements. As the function is *balanced*, there is no relative change, hence the pre-Δ corresponding to the *first* `lock`-statement will always be empty. Thus this statement is correctly never identified as a sSLP. The second `lock`-statement however will be instrumented by the transformation, and thus able to trigger a race if accessed simultaneously.

For the *fixed* dining philosophers, where e.g. the last philosopher accesses the locks in order l_1, l_n, although *the types change* for both locks (the first lock-set no longer contains π_n, the second no longer π_1), we still obtain the same instrumentation, and hence the same report about races, even though the program does not have any concrete deadlock.

Gate locks To increase precision (as a race can be already triggered by just *two* processes), we add gate-locks at appropriate places before the shared variable-accesses. This also requires reducing the amount of code shared between processes, as the transformation needs to be able to create *distinct* instrumentations for correlated SLPs. In the above philosophers example, all processes share the same function definition, so the `let` has to be pushed into the `spawn`s. This is here already sufficient to

disentangle the confusion about the origin of the locks in the types. In general though, the introduced gate locks are necessary to prevent the detection of partial cycles, but leave the overall behaviour with regard to actual deadlocks unchanged – they only influence how race variables are accessed.

Results

We give a formal description for our type system and prove the soundness of our approach, i.e., a program with a (potential) deadlock will be reported as having a race in its version instrumented/transformed for that particular cycle. Deadlocks can only occur at static second lock points for each of the involved processes. The transformation guarantees that each of these SLPs is protected by the same race variable for that cycle which implies a race between any two of the involved processes.

Acknowledgements Supported by the ARV grant of the Macao Science and Technology Development Fund.

References

[1] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[2] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.

[3] F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[4] K. I Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354, Mar. 2012.

[5] H. Seidl and V. Vojdani. Region analysis for race detection. In J. Palsberg and Z. Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2009.