A Survey of Methods for Preventing Race Conditions

Nels E. Beckman

May 10, 2006

1 Introduction

Software analysis can be an extremely powerful tool. It has been used to find bugs and ensure program correctness in a large variety of situations. As time goes on, it seems that our analyses have become more powerful, allow us to be more expressive, and work for larger and larger classes of systems. In this literature survey, we consider several different styles of software analysis, and their effectiveness at alleviating one very specific software defect: race conditions in concurrent software.

1.1 What is a Race Condition?

For the purposes of this survey, we will use the definition of a race condition that is most frequently cited in the works covered. However, it is important to note that this is not the only definition of race condition. According to Henzinger et al. "a race occurs when two threads can access (read or write) a data variable simultaneously, and at least one of the two accesses is a write" [9]. Intuitively, we think of race conditions occurring when a program reads a variable and then has to take some action based on the contents of that variable. If it is possible for another thread or process to 'sneak in' and change the variable's value in between the read and the action in such a way that the action about to be taken is no longer appropriate, then our program has experienced a race condition.

1.2 Why Race Conditions?

To some extent, it would have been interesting to compare several different analysis styles for solving *any* problem. However, race conditions are of particular interest because they can lead to some rather devious bugs. From a practical standpoint, debugging errors caused by race conditions can be very frustrating. For one thing, whether or not a race condition leads to an actual error depends on the inter-leavings of the threads during a given run of the program. It is a non-deterministic bug, and as most developers will tell you, it's extremely hard to track down a bug that cannot be easily reproduced. Also, the kinds of errors caused by race conditions are often

very subtle. They often manifest themselves in the form of corrupt or incorrect variable data. Unfortunately, this often means that the error won't crash the system immediately, but will wait until some other code is executed that relies on the data being correct, making the process of locating the original race condition even more difficult. Finally, programmers in general have a hard time reasoning about the correctness of concurrent code. It can be very difficult to internalize the interaction amongst several different threads when examining a piece of straight-line code. Therefore, any tool that can double-check the decisions made by programmers of concurrent code can be highly valuable.

1.3 Dimensions of Comparison

For the purposes of this survey, the different techniques examined will be compared across three primary dimensions; ease of use, which includes annotations, expressiveness and scalability, soundness and precision.

By ease of use we mean generally how easy or difficult it would be to integrate a given tool or technique into a development process. When evaluating, we would like to know what is the burden associated with the annotations necessary to make a technique work. We are also interested in whether or not a given tool or technique restricts the programmer's ability to use the idioms and coding styles with which he or she is familiar (we are calling this property expressiveness). Finally, research tools are often known for not scaling well to large, realistic systems. Therefore, we will examine scalability as well.

When discussing soundness, we are discussing the level of assurance provided by a particular technique. If a tool is sound, then that tells us as programmers that if the tool signals that there are no race conditions then their absence is in fact guaranteed. We also might say that a tool is mostly sound if there are only certain, unimportant situations in which false negatives may occur. Precision is related. If a technique is perfectly precise, it is said to be 'complete.' A complete tool will signal on actual race conditions, and never give us false positives. If a tool is not complete, then the fewer the numbers of false positives given, the more precise we will claim the tool to be. It is important to note that these dimensions are not all entirely independent from one another. For example, a tool with a large annotation burden might also not be scalable for that exact same reason.

After an in-depth look at the papers and techniques surveyed, we will return to these categories to see how each style of analysis compares.

2 Race-Free Type Systems

Language-based mechanisms for eliminating race conditions from software have been explored at great length, particularly recently. Here we will focus specifi-

cally on the type systems that have been developed to help eliminate race conditions from code. You could, however, certainly imagine other "language-based" techniques besides type systems for achieving the same purpose, such as runtime detection and compulsory compiler analysis. We might call these language-based techniques purely because a language standard required them. However, these sorts of techniques can essentially be discussed separately from the language, whereas a language's type system is a fundamental part of the language itself. In some ways, adding race-detection, or more accurately, race prevention to the type system seems natural. We have gradually seen more and more bugs that plagued programmers of earlier generations be pushed off to the side thanks to type theory. For examples of this phenomenon we can look to option types, which eliminate null dereferences, and type-safe languages, which prevent many classes of bugs associated with pointer arithmetic.

There is a strong attraction to building these sorts of mechanisms directly into the language. Essentially, a type system for preventing race-conditions can offer us code that is race-free by its very construction. All of the type-based techniques surveyed offer a strong assurance; if the program type-checks then it is guaranteed to be free from race conditions. Further benefits of type systems over other techniques are the lack of any performance penalty that might be associated with instrumenting code and the convenience and familiarity to the programmer of a compiler-based solution. The real question is, do the languages based upon race-free type systems allow us the same expressiveness that we are used to in more popular languages, and are there additional annotation burdens associated with these new type systems?

The three works discussed here ([8], [2], and [14], which are three of most recent works on the subject) build on an area of active research. Moreover, they all more or less build on the same basic idea, with each system adding its own improvements to the expressiveness of the underlying language. The idea at the heart of these type systems is that shared data must be protected with a lock in order to prevent race conditions. If part of the type of that shared data is the name of the lock that must be used to protect it, we can then check subsequent appearances of that data to make sure it is enclosed in code that acquires that specific lock. Of course, good software engineering practice dictates that the files and classes that make up a software system must be separately compile-able. So in order to avoid doing the sort of whole-program analysis that it would require to determine which methods could be held at a given program point, these three systems also use some variant of an *effects clause*. An effects clause is a part of a function interface that acts as a summary of that function's effects. In this case, those effects are of the sort, "my caller must possess lock X."

In order to get a better idea of how these type systems look in practice, we show here in figure 1 a representative example from EPRFJ [14]. This code comes from an example stack class. The `TNode` class here is being parameterized by two dif-

```
class TNode<thisOwner,TOwner> {
  T<TOwner> value guarded_by thisOwner;
  TNode<thisOwner,TOwner> next guarded_by thisOwner;

  T<TOwner> pop() requires (thisOwner) {
    return this.value;
  }

  // other methods omited
}
```

Figure 1: Code from EPRFJ

ferent locks, one lock that provides the protection for the object itself and another lock that is meant to protect the contents of that particular node. This form of lock parameterization is a vast improvement in expressiveness over previous type systems in that it allows different objects of the same class to be protected by different locks. Furthermore, it even allows different fields of the same object to be protected by different locks. When a field is `guarded_by` a lock, the type system knows that it needs to verify all accesses of that field are protected by that lock parameter. The `pop` method is an example of a method that relies on its caller to acquire the appropriate lock. This is signified by the `requires` clause, which allows the type checker to assume that the specified lock has already been acquired when type checking the body of the method. Elsewhere, at each call site for this method, the type checker will know to verify that the correct lock has been acquired. (This is the previously mentioned effects clause.) On the downside, it seems somewhat unnecessary to specify both the lock that protects a field and, when that field is being accessed inside a method, if that method requires a caller to have the lock in advance. It would seem that once we knew which lock was protecting the field we could just search the code to make sure that it matches this protocol. This is true, but it would require whole-program analysis, and would prevent separate compilation. Therefore, most type-based systems require this approach.

Upon entering a method with a given effects clause stating which variables must be held when that method is called, all three techniques [8] [2] [14] follow the same basic algorithm:

1. Add all locks listed in the effects clause to the current set of held locks and begin to step through method statements.
2. When a locking statement is encountered (e.g. `synchronize(this)` { . . . }), add that particular lock to the set of held locks.
3. When encountering a variable dereference, look up that variable's type (which

contains the lock that must be used to protect it) and verify that that lock exists in the set of held locks. (If not, signal an error.)

While the specific works examined here do not mention exactly how this algorithm operates in the face of branching statements, since the type systems are sound, we assume that a conservative merging technique will be used. This is necessary, since it is possible for different locks to be acquired in different control flow paths.

Up until this point, we have mentioned shared fields or variables without actually explaining how these type system approach them. A naive type system might assume that every variable could potentially be shared. This would require unnecessary locking and would certainly be unacceptable to programmers given the frequency of *thread-local* variables. In reality the type system must be informed that a variable is thread-local so that it does not look for explicit lock usage. In [2], objects can be *owned* by another object, itself, or `thisThread`, a special per-thread owner. Both other systems have nearly identical constructs, which do not affect the run-time behavior of the application. (Curiously, however, Cyclone [8] still requires the programmer to call `synch` on a `nonlock` variable, even though it has no runtime effect.)

Despite the similarities in basic technique, each paper adds on certain features that help make their system more programmer friendly in one way or another. Each has a different system of type inference along with defaults that let the programmer get away without writing very many lock annotations. Cyclone, has polymorphic functions, a feature not possessed by Java at the time of [2] and [14], and therefore must add some more machinery in order to make their lock types work with these functions [8]. Boyapati et al. integrate their type system with a similar system for preventing deadlocks [2], while Sasturkar et al. [14] integrate their technique with a type system for preserving method atomicity; the property that a given method cannot be interleaved with itself. They furthermore add a *readonly* type in addition to the shared variable type. Since race conditions can only occur when reads and writes are happening at the same time, this type allows their system to have shared read-only variables without lock protection.

When considering a type system for preventing data races, one has to consider certain trade-offs. For one, these type systems only enforce one programmer discipline, that of protecting shared data with locks. All of these type systems are sound in that they will not miss any potential race conditions, however, they limit the programs that can be written, and some race-free programs are not allowed. For instance, programmer knowledge about the thread ordering, which may arise from explicit forks and joins, can obviate the need for locking. These systems would still require locks to be used. Also, variables often have an initialization phase (before other threads have been forked off) where variables do not need to be locked but where these systems would force them to. Finally, there is the simple burden of annotation. In addition to traditional typing annotation, all three systems add a host

of new locking annotations that impose an additional burden on the programmer. While this cost is certainly worth it for concurrent code with high reliability requirements, it is not at all clear that corporations developing this type of software would be willing to adopt a non-standard language for this purpose. The best possible outcome would be that ideas developed in this field would make it in to a future version of an industrial strength language.

3 Dynamic and Hybrid Race Detectors

While in general it is more desirable to find software defects before run-time using static analysis tools, there are certain benefits that dynamic race detection tools can offer. The general rule of thumb, at least at this point in time, is that where static analysis tools are forced to be conservative and produce more false positives, dynamic analysis tools can use very intimate knowledge about the runtime behavior of the application in order to increase precision.

There are two major tools that are used when designing a run-time race detection program. *Lockset*-based tools essentially work based on the principle of *locking discipline.* In other words, the assumption is that race conditions occur because of shared variables that are mistakenly not protected by an appropriate lock. (As we will see later, locks are not the only way to provide safe synchronization and therefore *lockset*-based techniques tend to produce superfluous false-positives.) On the bright side, however, lockset-based tools are more sound than other approaches. They have the ability to, based on the knowledge gathered during one thread interleaving, detect potential race conditions that could only occur in another thread interleaving. In general, lockset analysis does a good job of detecting most of the potential race conditions, but has poor precision.

Happens-before race detectors work on a different principle, one that essentially watches for thread accesses to a given piece of data that do not have any implied ordering between them. This system for detecting race conditions is highly dependent on the actual thread execution ordering that occurs when the instrumented code is run. Therefore, the warnings that are given by dynamic race detectors operating on the happens-before principle only represent a subset of the actual race conditions in your program. They are, however, race conditions that actually exist. (Meaning the technique is complete.) Modern dynamic race detection tools work by using both of these techniques at the same time [11] [18] (with the possible addition of information from static tools in order to improve their efficiency). First, and in order to develop a better understanding of the pros and cons of dynamic race detection, we will give a brief overview of these two techniques before discussing how they might be used in combination.

All existing *lockset*-based analyses descend from Eraser [15], the first dynamic tool to use this particular algorithm. In a lockset analysis, the program runs under

the assumption that all shared variables must be accessed within the protection of a lock and, furthermore, that that specific lock should be used consistently when accessing that location in memory. The software that is being monitored for race conditions is instrumented so that each shared variable has an associated *candidate set*; the locks that have up until that point been used to protect that location in memory. (At initialization, this candidate set contains every possible lock in the system.) When a thread accesses that shared location in memory, the algorithm takes the intersection of all locks held by that thread and the memory location's candidate set. Eventually this candidate set will be pared down to exactly the locks that are supposed to be protecting the shared memory location. So if at any point the intersection between a thread's held locks and the candidate set of the shared variable that it is accessing is empty, the algorithm signals a possible race condition, since at that point none of the previously used locks are protecting that shared variable.

There are a number of problems with the lockset algorithm, but the most obvious one is that locks aren't always necessary, even when accessing a shared variable. Locks aren't the only method for synchronization, so the user of this type of analysis must live with that assumption. Furthermore, there can be times in a multi-threaded program when the programmer has greater understanding, and knows that in a certain program point the variable cannot be accessed by more than one thread. An example of this type of synchronization would be a system in which one thread repeatedly forked and joined another thread. As long as locks were used to protect the shared variable when two threads exist, no race conditions will exist and the programmer is free to not protect the variable after the explicit join operation [18]. The lockset algorithm, however, would still signal an error. For this reason, the lockset algorithm tends to produce numerous false-positives, which make it somewhat painful to use. In general, real race conditions are tricky to understand. Therefore, if a tool is giving real and false race conditions back to the programmer, he is likely to get quite frustrated looking for bugs that may not exist.

The *happens-before* analysis is another way to dynamically detect races, and it has the benefit that it does not give false-positives. False-negatives, however, can be a much greater problem. The happens-before algorithm is based on the principle that certain operations, like the forking and joining of a thread of the locking and unlocking of a lock can establish a partial order of threads in a software system. Each thread keeps what is known as a *vector clock,* a time-stamp that increases as synchronization events occur. At the same time, shared memory variables are also keeping track of the time-stamp of the last thread to access it. Additionally, threads are also keeping track of the timestamps of other threads, at least to the best of their abilities. When two threads synchronize in some way (a join for instance) they learn of the other's current time value. After keeping track of all this, when a thread t goes to read a variable, the following must hold: for every other thread that has

previously accessed the variable, t's knowledge of that thread's time must be less than or equal to the last time that thread accessed the given variable. If this does not hold, a race has been detected.

MultiRace [11] and RaceTrack [18] are two recent examples of tools that join these two techniques together in order to get the best of both worlds, and are the current state-of-the-art in dynamic race detection. Both of the tools make specific engineering choices in order to make their analyses more practical to use. For instance, one of the problems with previous dynamic race detectors was an artificially impose granularity on the size of shared memory to be checked. As one can well imagine, monitoring shared memory at a granularity of one word will vastly increase the memory used by the application that is being monitored and the amount of time that application takes to run. Existing tools monitored data at the object level. Of course, since multiple locks can be used to protect different fields of an object, this tended to lead to false positives. Therefore, RaceTrack and MultiRace both dynamically reduce the detection granularity once a race condition has been detected on an object. Another important choice that these two tools take is to have a phased usage of both lockset and happens-before race detection. This is achieved by stepping the shared variables through state machines. A shared variable will start off using the lockset analysis, which is in general cheaper to perform and catches more potential races. If, however, a race condition is detected for that object, the state machine will step to performing a happens-before analysis for subsequent memory accesses in order to verify the validity of that race condition.

Even the best dynamic race detectors have their own share of problems. For instance we just discussed how these detectors step through different forms of race detection, from a more sound form to a complete form after race conditions are detected. Of course, since there is no guarantee that even a real race condition will reoccur, this only allows the tools to better rank which race conditions are most likely to be bugs. The results themselves are neither sound nor complete. Also, the general problem of dynamic detectors of any kind is that you have to actually run your program for them to do any good. In other words, you don't have the assured feeling that comes from running the compiler or a static analysis ahead-of-time. Dynamic race detectors still can only find bugs in the execution paths that are actually taken at runtime. False positives are still a problem, but the combination of *lockset* and *happens-before* analysis has helped to alleviate this issue. In addition, researchers are now finding ways to feed the results of static analyses into these dynamic tools to help focus the checks that are performed at runtime [10] [1]. The code instrumentation tends to increase memory usage by about 1.2 times and slows them down about 2 times. This is probably enough overhead to prevent these analyses to be included in actual deployed software. In summary, while there dynamic detectors are by no means perfect, thanks to hybrid analysis, adaptive granularity and better ranking systems, modern tools are able

to find race conditions while presenting drastically fewer false positives than in previous dynamic tools.

4 Using Model-Checking to Detect Race Conditions

Model-checking is an extremely powerful analysis technique. Therefore it seems only natural to attempt to use it in the verification of concurrent systems. The idea behind model-checking is conceptually very simple; to explore every possible execution path for all possible variable values in order to determine if certain undesirable behaviors might occur. Of course, phrased in this manner, the problem is completely intractable. For this reason abstractions of both the control flow and the data values of a particular program are created, and it is this *model* of the software system that is explored.

In a very real sense, model-checking is a simulation of the application in question. Therefore, the naive way to extend model-checking to concurrent applications would be to encode all possible thread inter-leavings into the model itself (i.e. given two threads, t_1 and t_2 from a given state we could transition to another state where t_1 was picked first by the scheduler *or* a different state where t_2 was picked first by the scheduler). This, of course, would lead to a combinatorial explosion as all possible control paths were considered for each of all possible thread inter-leavings. Therefore, the primary challenge of using model-checking as a tool to find or assure the absence of race conditions is finding a model of the system that can be explored in a reasonable amount of time.

In several ways, the line of research that culminates in [9] is more natural and straightforward way of using model-checking for this purpose. However, for completeness, we first discuss the concept of *state-less search* and the related concepts of *persistent sets* and *sleep sets* as a way to reduce the overall search space.

State-less search is a form of model-checking where there is no "memory" of the states that have already been visited. In the model-checking systems discussed in class, if the model were to transition to a state in the CFA that it had already visited and where all predicates had the same value as on a previous visit, it would know that this was a previously explored path, and would not do it again. This is not the case when using state-less search. This is necessary because keeping track of the full state would then mean maintaining a full stack for each thread. [12] This system tracks only *visible* transitions in the state machine; these are non thread-local operations. Tracking only visible transitions (in some sense the only transitions that matter when looking for concurrency bugs) greatly decreases the possible state-space. Persistent sets and sleep sets help us to cut down on the number of states that must be explored.

Stoller [16] spends a fair amount of time building up the machinery necessary for his analysis, in particular he creates a detailed model of Java's concurrency

system. The real meat of the research, however, comes from the fact that *persistent sets* can be used as a justification for not exploring all possible inter-leavings of state transitions (In his model one state machine exists for all threads, therefore which thread is chosen is represented by the transition from a state that is chosen). Persistent sets are sets of transitions leaving a state that are independent from every sequence of transitions leaving that state which are not in the persistent set. The independence of a transition T from a sequence of transitions S implies that the resulting system will be the same no matter whether T or S is taken first. Both inter-leavings are identical and therefore do not need to be explored separately. Sleep sets are just the transitions that make up the sequences independent from those in the persistent set. The transitions in the sleep sets are never explored.

After all of this, the method by which Stoller detects race conditions seems almost anti-climactic; he uses the lockset algorithm as previously discussed in section 3. Since his system is essentially simulating the all possible executions of a program, there's no reason why a dynamic detection technique cannot be used. Furthermore, since all possible executions are being explored (unlike in dynamic detection where the executions explored depend on the tests that you run) his system can make a sound claim that a given program does or does not satisfy the shared-variable locking discipline. False positives will occur if a strict locking discipline is not observed. On the bright side, his model includes an initialization phase where variables are as of yet unshared and therefore are not required to be protected by locks [16].

A more precise way to detect race conditions would be to search the state space for a state where amongst multiple transitions existed reads and writes by different threads to the same variable. This is exactly the approach taken by Henzinger et al [9]. In order to handle the multiple thread issue, their system uses what's known as a *context*. Building upon earlier work, their system has one abstract reachability graph (ARG) for the "main" thread which keeps track of local and global variables. The context represents every other thread in the system. It is built up while constructing the main abstract reachability graph so as to be only as precise as absolutely necessary. Each state in the context has a counter that keeps track (up to a certain constant) of the number of threads in each abstract state. Furthermore, local variables are not tracked on this graph. The entire system is based upon what is known as the "assume/guarantee" principle: It starts by verifying that the context and ARG represent a safe system. This is based on the assumption that the context represents a sound approximation. Then it is verified that the context in fact represents a sound approximation of the system itself ("guarantee"). Here is a more thorough explanation of the process:

1. Initially the context is empty and there are no predicates.
2. Using the current context and predicates, construct an ARG from the combi-

nation of control flow automaton (CFA) and the current context. This follows the exact same algorithm that we used for our homework problems, except that now, at any point we can chose to take a transition on the main CFA or on the abstract CFA (another term for the context).

3. Stop when an error state is found. An error state is one where the same variable can follow either a transition where a variable is read or another transition where that same variable is written to.

4. Just as in class, the system determines if the path taken was actually feasible. In it was, an error is signaled, if not new predicates are inferred (based on the principle of CEGAR) and the process begins again.

5. If an error was signaled in the previous step, we must now guarantee that the ACFA was in fact a sound approximation. If it was, we have a genuine example of a race condition. If not, then we refine the context, yielding a more accurate approximation, and then rerun the process.

One thing that is not clear from the above description is what it means for an ACFA to be a sound approximation. The ACFA must represent an over-approximation, or simulation, of the actual ARG. The technique for determining whether or not this is the case was developed in prior work. Intuitively, however, we say that if something is possible in the CFA, it must somehow be possible to represent that same thing in the ACFA, even if in a less precise manner. By keeping this ACFA as abstract as is sound and by using it to represent every other thread in the system, we dramatically cut down the possible states that must be explored. Just in the was CEGAR allows predicates to not be tracked until they are needed.

There are some very clear benefits that come from using this particular model-checking technique for detecting race conditions. The technique is sound, but more importantly it seems to be much more precise than other techniques we have explored here. This is because instead of detecting violations of the locking discipline that can be used to prevent race conditions, here we are actually detecting the race conditions themselves. State variable based synchronization and split-phase based synchronization are two non-locking synchronization techniques that would normally cause false positives in other techniques that do not here [9]. Unlike previous model-checkers in the same research tree, this particular system does not put a cap on the number of concurrently executing threads. It works for an arbitrary number. Finally, because the system is counter-example based, when an error is detected, the actual thread interleaving that produced the race can be given back to the programmer. Race conditions are tricky to locate, and this will likely make fixing an existing bug much simpler.

Still, there are some unanswered questions that are left open for future research and prevent this from being the ultimate race detection tool. One is that their system

as described only works for basic data types. When attempting to adapt their system to work for pointers, the authors ran up against aliasing problems. The system could not tell if two pointers referenced the same object, and therefore could not tell if a race was occurring or not. Additionally, their tool was actually used on nesC, a variant of C whose locking mechanism is the `atomic` keyword. Since `atomic` actually turns of the interrupts for the duration of its block, it does not act in the same manner as traditional locks. It is not clear from the text that this system could be trivially modified to find races in a system using more traditional locks.

5 Flow-Based Race Analysis

While research into other, less "traditional" forms of race detection and analysis has been pretty steady in the past few years, the same cannot be said for flow-based analyses. Recent papers on the subject were fewer and farther between, and it was necessary to go back to 2003 [4], 2001 [5] and 2000 [3] in order to find significant discussions of this research ground. It is quite interesting to note that some of the other techniques that we have and will examine claim that they could be even more effective if used in combination with existing flow-based static analysis techniques. In particular, when discussing model-checking [16] and dynamic race analysis [10] the developers of these techniques recommend using a traditional static analysis in order to determine all the possible statements where races *might* occur. Then based on that information they propose reduce the overhead of their respective techniques by only examining those possible statements.

Achieving soundness in the detection of race-conditions is old hat for static race detectors. What makes these tools painful to use are the maddening number of false positives that can be reported. The work surveyed in this area seems to take three distinct approaches to the issue of making static analysis of race conditions more precise: 1) What essentially amounts to engineering effort and the willing sacrifice of *some* soundness, 2) improving and awaiting further improvements in the underlying static analysis technologies that make detection of race conditions (and many other potential bugs) possible, and 3) using programmer annotations to increase precision. Here, in turn, we will take a look at the results of some of these efforts.

Of all the systems reviewed in this survey, RacerX seems to be the best prepared to be run in an actual software development environment. No other tool was capable of examining such large, (millions of lines of source code) realistic (Linux, FreeBSD, and a large commercial system) systems with comparable speed or numbers of false positives (on the order of ten, versus hundreds and thousands for other systems). The authors start with a basic, interprocedural, lockset analysis and then push on the concept using heuristics, statistical analysis and ranking to produce a result that, while no longer sound, tends to actually find bugs.

The lockset analysis itself is relatively straightforward. RacerX starts by building a control-flow graph of the entire application. Using a flow-sensitive procedure (one that remembers the past results of each execution path) the analysis adds locks to the lockset when encountering locking statements and removes them when encountering unlocking statements. As the tool encounters accesses of thread-shared variables, it verifies that their locks are in the current lockset, and if not a warning is produced.

But how does RacerX know which variables are shared and which locks protect those that are? These issues are solved in some interesting ways. Belief analysis is used to determine if code is even multi-threaded and to help determine if a given variable needs to be protected. The use of any sort of concurrency statement implies to the system that that section of code is probably multi-threaded. (The term probably is used because nothing is known for sure and the likelihoods of different scenarios are worked into the tool's final ranking of possible races.) Similarly, if accesses of a variable tend to occur inside a lock it's a good indication that that variable does in fact need to be protected; "bonus points" if that access is the first or last statement in a critical section. To determine *which* lock should protect a given variable, RacerX counts the number of times that a variable is protected by a given lock verses the number of times that it is not. If this number of times is statistically significant, then an association is created.

Some mundane (from a research point of view) engineering details also help RacerX to be a more useful tool. For example, the set of statements that constitute a lock or an unlock can be specified by the user. This helps the tool to be more useful in applications that have a large number of locking idioms. Also, in order to improve performance, RacerX caches the locksets that have already arrived at a given function or statement. If a duplicate lockset revisits a statement or function call site through another path but with the same locks held, the previous results can be reused. (Although a function will be skipped if its cache exceeds a certain size, one particular source of unsoundness.)

The number and variety of techniques used by the authors to rank the potential severity a particular warning is too large to discuss them all here. Suffice to say that the authors are serious about developing a more precise race detection system and appear to have the analysis experience to include helpful heuristics. Houdini/rcc [5] is another tool that has taken a similar, engineering-driven approach of using a simple analysis and then aggressively attacking false positives. It is similarly unsound.

Static data race detection analyses like the one developed by Choi et al. [3] rely in great part (by the authors' own admission) on the precision of alias and escape analysis available. The authors discuss how their system performs a path-sensitive dataflow analysis over a program's inter-thread call graph (a traditional call graph that additionally encodes thread spawns as directed edges). Over the course of this analysis, sets of abstract objects (abstractions of actual objects that

would exist at runtime) are built up. These abstract objects are used with alias analysis to determine if a race-detection predicate is true or not. The race detection predicate encodes four clauses that must be true for a race condition to occur:

1. Two ICG nodes must access the same object in memory.
2. Two nodes must occur within the execution of different threads.
3. The two events represented by these nodes must be protected by different synchronization objects.
4. No specific thread ordering (join, wait, notify) is enforced between those two threads.

Alias analysis is used as a way to test for all of these conditions, even the last condition, which is determined by a combination of path ordering encoded in the ICG and knowledge about whether or not two variables represent the same thread.

Because of the wide applicability of alias analysis and escape analysis, a large amount of research in these areas is currently ongoing (e.g. [13]). However, these seem to be fundamentally hard problems to solve, ones that push up against the boundaries of what is and is not decidable. Therefore, the time when alias analysis is good enough to allow sufficiently precise detection of race conditions may be a long way off in the future.

Fluid [7] has chosen to take a somewhat different approach in its solution to the problem of detecting race conditions (and a variety of other defects). This tool uses a large number of sophisticated and detailed annotations, that in total make up the policy of concurrency in a given software system. The authors argue that traditionally the concurrency policy is expressed anyway, in the form of comments and other documentation which are not machine checkable. Their system essentially uses annotations to make checking modular, as well as for explaining to analyses exactly which usages are correct and which may constitute an unsafe operation. These annotations allow the developers to describe many different policies. Annotations can be used to define groups of state which may potentially cross object boundaries. Once state has been grouped together, annotations can be used to specify which locks are being used to protect that state, and whether or not the state needs to be protected at all. It also allows for a very precise description of which methods can be inter-leaved and which ones cannot, something not really available in other systems. This includes the ability to specify whether object clients or the objects themselves are in charge of assuring proper state protection. These annotations can then be used by relatively straigtforward analyses in order to verify that they are consistent with the code itself. In some ways this approach appears similar to the ones used by type-based race detection mechanisms. Although thanks to additional

analyses such as thread-coloring, they have somewhat more expression. This analysis can determine that certain data is only accessed by single-threaded code and therefore locking would be unneccessary.

6 A High-Level Comparison of the Techniques Surveyed

After an in-depth look at each of the four analysis styles covered, in this section we will quickly review each of them across the dimensions of comparison discussed in section 1.3.

6.1 Annotation Burden

Race-Free Type Systems Here annotations are one of the major limiting factors. Each type system seems to require a fair number of annotation in order to tell the type system about the relationships amongst locks and variables. Type inference and well chosen defaults (i.e. What does it mean when no annotation is given?) have alleviated this problem to a certain extent, but not entirely.

Dynamic Race Detectors In all of the dynamic race detectors analyzed, no annotations were necessary, which makes sense because the instrumentation itself can figure out from the execution everything that it needs to know.

Model-Checking Again, annotations were not necessary for any of the model-checking techniques surveyed.

Flow-Sensitive Analyses Some form of annotation seems to be required for all flow-based analyses. The amount required, however, appears to vary with the soundness and precision of a given technique. RacerX [4] requires a constant number of annotations for programs of any size. It can still be precise because it is not a sound tool. When a tool is both sound and precise, (Fluid [7], for instance) it appears that more annotations are necessary in order to 'prune' away false positives.

6.2 Expressiveness

Race-Free Type Systems The expressiveness of type-based approaches is rather limited because of the enforcement of a strict locking discipline. However, this area has seen great improvement in recent research (object migration and parameterized functions can be expressed) and it seems that this area will improve somewhat.

Dynamic Race Detectors Thanks to the combination of two existing techniques, expressiveness is quite good. The false positives that might otherwise be given

when straying from a locking discipline and just using *lockset* are suppressed thanks to the addition of *happens-before* analysis.

Model-Checking Model-checking can allow for very great expressiveness since we are essentially looking for the race conditions themselves, not the conditions that might cause race conditions.

Flow-Sensitive Analyses Flow-sensitive analyses are expressive thanks to programmer knowledge which can either be encoded through heuristics ([4]) or annotations ([5]).

6.3 Scalability

Race-Free Type Systems In principle race-free type systems are scalable, however the large number of annotations required by them makes it unlikely that they would be used for large applications. Similarly, it would be hard to force developers to annotate the libraries or legacy code that might be used in a large system.

Dynamic Race Detectors In some sense, the dynamic detectors work just as well on large applications as they do on small applications. However, there is a memory and performance overhead associated with dynamic detectors, and on a production system this overhead may be too much of a burden. If the overhead (something current research is trying to improve) were small enough, then organizations would be able to gain valuable bug data from the field.

Model-Checking Scalability is a very interesting issue here. The short answer seems to be no, model-checking isn't practical for analyzing large systems. However, large may not necessarily be defined by lines of source code when it comes to detecting race conditions. VeriSoft [6], a precursor to [9] based on very similar technology, was effectively used on 500,000 lines of C++ code. However, in a program with tens of processes executing at the same time, it would quickly be overwhelmed [17]. It all depends on the tool's ability to eliminate potential thread inter-leavings.

Flow-Sensitive Analyses Flow sensitive analyses are generally scalable because they are generally designed for industrial use. RacerX has shown that it can be run on real production quality code, and has been on Linux and FreeBSD among other applications. Similar statements can be made about Fluid, which has also been run on realistic code bases.

6.4 Soundness

Race-Free Type Systems One of the biggest benefits of race-free type systems is that they are sound.

Dynamic Race Detectors Dynamic race detectors are not sound by their very nature. They cannot explore code over which they are not executed. However, thanks to *lockset* analysis, which can find potential race conditions even for thread inter-leavings that did not occur in testing, dynamic race detectors have a fairly low incidence of false negatives.

Model-Checking Model-checking is also sound, but it may not always terminate.

Flow-Sensitive Analyses Some flow-sensitive analyses are sound, and others are not. However, even for the tools that are not sound, they seem to be sound in ways that do not dramatically affect their rate of false negatives. RacerX even has to ability to detect false negatives and warn the user about them (to a certain extent) [4].

6.5 Precision

Race-Free Type Systems Thanks to their strict adherence to locking discipline, race-free type systems tend to produce numerous false-positives.

Dynamic Race Detectors While in the past, dynamic race detectors have also been known to produce numerous false positives, the addition of *happens-before* analysis has helped to increase their accuracy somewhat. Furthermore, dynamic tools may return a list of race conditions known absolutely to be race conditions, which is very helpful.

Model-Checking Model-checking also has a very high precision, but due to the undecidability of determining reachability, it may never terminate.

Flow-Sensitive Analyses The state-of-the-art in flow-sensitive static analysis can have very high precision, thanks either to sacrificing soundness or using annotations to express programmer intent. Since the current state-of-the-art in conservative alias detection tends to produce numerous false positives, the same can be said for race detection tools based around this technology.

6.6 Discussion

For the purposes of this discussion, we will define the 'best' race detection technique as the one that is currently the most read to be applied to a large-scale industry project. Based on this definition, it seems that a flow-sensitive analysis, of the type discussed here might be the best bet. They seem to have the combination of

precision and ease of use that would be required to actually be useful. However, it seems like in the future, when appropriate research advancements have been made, model-checking may turn out to be an even better tool. Model-checking allows for a variety of programming idioms, is complete and sound (within the bounds of computability) and requires no annotations. Based on the modeling abstractions that we currently have, however, it is not quite ready to scale to industrial level systems. Perhaps we are almost there.

7 Conclusion

In this survey, four different analysis styles were compared, all with the goal of detecting or preventing race conditions. Race conditions are a devious form of bug, and therefore the effectiveness of these techniques is of great interest. The techniques surveyed varied widely in the characteristics of their operation, but in the end, it seems as if a flow-based analysis would be the best tool for finding race conditions in an industrial setting, at least at this point in time.

References

[1] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 233–242, New York, NY, USA, 2005. ACM Press.

[2] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[3] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs, 2001.

[4] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[5] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT*

workshop on Program analysis for software tools and engineering, pages 90–96, New York, NY, USA, 2001. ACM Press.

[6] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[7] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 453–463, New York, NY, USA, 2002. ACM Press.

[8] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[9] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[10] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[11] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, New York, NY, USA, 2003. ACM Press.

[12] Sriram K. Rajamani. Crash-course in model-checking part 3: Model-checking software. Microsoft Research Lecture Series, 1999.

[13] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[14] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, New York, NY, USA, 2005. ACM Press.

[15] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[16] Scott D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 224–244, London, UK, 2000. Springer-Verlag.

[17] Lucent Technolgies. Verisoft: Frequently asked questions. http://cm.bell-labs.com/who/god/verisoft/faqs.htm.

[18] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM Press.