How Developers Use Data Race Detection Tools

Caitlin Sadowski
Google, Inc.
supertri@google.com

Jaeheon Yi
Google, Inc.
jaeheon@google.com

Abstract

Developers need help with multithreaded programming. We investigate how two program analysis tools are used by developers at Google: THREADSAFETY, an annotation-based static data race analysis, and TSAN, a dynamic data race detector. The data was collected by interviewing seven veteran industry developers at Google, and provides unique insight into how four different teams use tooling in different ways to help with multithreaded programming. The result is a collection of perceived pros and cons of using THREADSAFETY and TSAN, as well as general issues with multithreading.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis

General Terms Human Factors, Design

Keywords data race detection; type systems; dynamic analysis

1. Introduction

Multithreaded code is difficult to write, understand, and debug [14]. Developers are faced with a variety of concurrency-specific errors such as deadlocks and data races. Synchronization policies may be poorly documented and incorrectly inferred by new developers. In order to better support programmers who work with concurrency, a variety of program analysis tools have been developed, both in research and industry. However, little is known about the tradeoffs industry developers make when using concurrency program analysis tools.

PLATEAU '14, October 21, 2014, Portland, OR, USA..

ACM 978-1-4503-2277-5/14/10.
http://dx.doi.org/10.1145/2688204.2688205

In this paper, we report on the results of interviews with developers about how they use two data race detection mechanisms: THREADSAFETY and TSAN. Both tools are used widely at Google, by interested teams. Teams typically find out about these tools via social mechanisms (e.g. an engineer on a team who used them with a previous team) or internal advertising. Engineers decide whether to try out or keep using the tools; there are not any requirements (e.g. by management) towards using either tool.

1.1 THREADSAFETY

THREADSAFETY is an annotation-based intra-procedural static concurrency analysis that identifies data races caused by inconsistent protection of a shared variable and violations of lock acquisition ordering [24, 25]. This analysis was originally implemented as a branch of gcc (under the name *annotalysis*), but has been reimplemented in the Clang compiler; all current support is for the Clang version. THREADSAFETY is publicly available, and is included by default with Clang. The analysis is run by compiling an annotated program with the `-Wthread_safety` flag enabled.

Two core annotations, which we describe below, are `guarded_by` and `acquired_after`:

- `guarded_by` specifies a particular lock should be held when accessing the annotated variable. Violations of this locking policy may lead to data races.
- `acquired_after` annotations document the acquisition order between locks that can be held simultaneously by a thread, by specifying the locks that need to be acquired before the annotated lock. Violations of this locking policy may lead to deadlocks.

A code snippet which demonstrates these two annotations is in Figure 1, along with sample compiler output. A full list of available annotations can be found with the Clang language extensions documentation [25].

These annotations have been in use at Google for a few years. In a relatively small subsection of code at Google that has thread safety annotations, one month of commit activity included 18 bug-fixing commits (not including commits

```
 1 #include "thread_annotations.h"
 2 #define GUARDED_BY(x) __attribute__((GUARDED_BY(x)))
 3 #define ACQUIRED_AFTER(x) __attribute__((ACQUIRED_AFTER(x)))
 4
 5 Mutex mu1;
 6 Mutex mu2 ACQUIRED_AFTER(mu1);
 7
 8 int x GUARDED_BY(mu1);
 9 int a GUARDED_BY(mu2);
10
11 void foo()
12 {
13   mu2.Lock();
14   mu1.Lock();
15   if (x > 2)
16     a = x + 1;
17   else
18     a = x - 1;
19   mu1.Unlock();
20   mu2.Unlock();
21 }
```

Sample compiler output:

```
ex.cc: In function 'void foo()':
ex.cc:12: warning: Lock 'mu1' is acquired after lock 'mu2' (acquired at line 14)
but is annotated otherwise
```

Figure 1. C++ code sample demonstrating use of `guarded_by` and `acquired_after` annotations.

which just fix the annotations) that cited THREADSAFETY explicitly as the way the bug was found [23].

These annotations are based on the type annotations presented for a type system that detects data races [8]. One key difference is that since not all source code may be annotated, there is no soundness guarantee. In particular, THREADSAFETY does not guarantee the absence of races. Specifically, it can only identify races that are on variables that have been annotated with a `guarded_by` annotation in which the lock expression is resolvable.

1.2 TSAN

TSAN (ThreadSanitizer) is a dynamic concurrency analysis that identifies data races [29]. TSAN has two versions: TSAN V1 and TSAN V2. Both instrument a running program, and both find the same kinds of errors.

TSAN V1 is implemented in Valgrind, a heavyweight binary instrumentation framework [18]. It is based on two well-known techniques: locksets [27] and happens-before [3]. By default, TSAN V1 uses both techniques in combination to improve coverage and hence may report a false positive. TSAN V1 is publicly available and is an open source project [31].

A code sample and associated error report (reproduced here with permission from the TSAN V1 wiki [31]) is in Figure 2. The error report points out the racy accesses on line 48 and 53 in the sample code. Instructions on how to interpret the rest of the error report are available on the TSAN V1 wiki.

The recently developed TSAN V2 is a compiler-based instrumentation pass in LLVM [30]; it is also open source [32]. The TSAN V2 runtime is engineered for speed, and exhibits slowdowns of roughly 5–15x (TSAN V1 exhibited slowdowns of 20–300x). In contrast to TSAN V1's hybrid algorithm, TSAN V2 implements a pure happens-before algorithm, and hence has no false positives.

TSAN regularly finds critical bugs, and is in wide use across Google (previously TSAN V1 and currently TSAN V2). One interesting incident occurred in the open source Chrome browser. Up to 15% of known crashes were attributed to just one bug [5], which proved difficult to understand – the Chrome engineers spent over 6 months tracking this bug without success. On the other hand, the TSAN V1 team found the reason for this bug in a 30 minute run, without even knowing about these crashes. The crashes were caused by data races on a couple of reference counters. Once this reason was found, a relatively trivial fix was quickly made and patched in, and subsequently the bug was closed.

TSAN V2 was designed with lessons from TSAN V1 in mind. Hence, TSAN V1 has been discontinued in favor of TSAN V2. However, at the time of this study, teams were still using TSAN V1. For the remainder of the paper, TSAN will be used to refer to TSAN V1. Note that both tools are run in the same way.

Excerpt from `demo_tests.cc` *test file:*

```
 42 Mutex mu1;  // This Mutex guards var.
 43 Mutex mu2;  // This Mutex is not related to var.
 44 int   var;  // GUARDED_BY(mu1)
 45
 46 void Thread1() {  // Runs in thread named 'test-thread-1'.
 47   MutexLock lock(&mu1);  // Correct Mutex.
 48   var = 1;
 49 }
 50
 51 void Thread2() {  // Runs in thread named 'test-thread-2'.
 52   MutexLock lock(&mu2);  // Wrong Mutex.
 53   var = 2;
 54 }
```

Sample tool output:

```
WARNING: Possible data race during write of size 4 at 0x6457E0: {{{
   T2 (test-thread-2) (L{L3}):
    #0  RaceReportDemoTest::Thread2 demo_tests.cc:53
    #1  MyThread::ThreadBody thread_wrappers_pthread.h:341
  Concurrent write(s) happened at (OR AFTER) these points:
   T1 (test-thread-1) (L{L2}):
    #0  RaceReportDemoTest::Thread1 demo_tests.cc:48
    #1  MyThread::ThreadBody thread_wrappers_pthread.h:341
  Address 0x6457E0 is 0 bytes inside data symbol "_ZN18RaceReportDemoTest3varE"
  Locks involved in this report (reporting last lock sites): {L2, L3}
   L2 (0x645720)
    #0  pthread_mutex_lock ts_valgrind_intercepts.c:935
    #1  Mutex::Lock thread_wrappers_pthread.h:147
    #2  MutexLock::MutexLock thread_wrappers.h:286
    #3  RaceReportDemoTest::Thread1 demo_tests.cc:47
    #4  MyThread::ThreadBody thread_wrappers_pthread.h:341
   L3 (0x645780)
    #0  pthread_mutex_lock ts_valgrind_intercepts.c:935
    #1  Mutex::Lock thread_wrappers_pthread.h:147
    #2  MutexLock::MutexLock thread_wrappers.h:286
    #3  RaceReportDemoTest::Thread2 demo_tests.cc:52
    #4  MyThread::ThreadBody thread_wrappers_pthread.h:341
}}}
```

Figure 2. C++ code sample demonstrating use of TSAN.

2. Methodology

We conducted interviews with seven veteran industry developers at Google about the process of multicore programming in C++. All participants had 8-30 years of experience programming with concurrency, were based in the US, and were male. Each interview lasted about an hour. The interviews were semi-structured, and were loosely based around the following questions:

- How many years of experience do you have programming with concurrency? How often do you write, read, or debug concurrent code? How many years of C++ experience?
- Tell me about a couple bugs or problems that you found with the help of THREADSAFETY/TSAN. About how many bugs do you think it helped find?
- What do you see as major problems with the current THREADSAFETY/TSAN implementation?
- For THREADSAFETY, do you think the annotation burden is reasonable? What about the potential for false positives? Tell me a little about the productivity tradeoff when using these annotations. What would you like to see as the future of THREADSAFETY?
- How do you test/debug concurrent programs? Can you think of any ways to improve this process? Other thoughts on frustrating things with concurrent programming?

We then extracted themes from the interview data, by coding the interview responses (Section 4). We took the interview results and iteratively coded them to identify themes. We both open-coded the results from one interview, and reconciled the two codings so as to calibrate schemes and improve internal validity [19, 28]. The remainder of the coding was performed by one person and checked by the other.

Three of the participants had a predominately cross-team role where they advised other teams on concurrency. For the remaining four participants, we additionally asked about

team practices for THREADSAFETY and TSAN; we report how four different teams used tooling in different ways to help them with multithreaded programming in Section 3.

2.1 Investigators

Neither author of this paper is on the same team as any of the interviewees. The second author is currently doing work related to TSAN, but was not at the time of this study (and is not involved with THREADSAFETY). The first author worked on the the transition from gcc to Clang for THREADSAFETY at the time of this study, but is currently not working on either tool.

2.2 Participant Recruitment

We only selected interviewees that had experience with THREADSAFETY and were very experienced with multithreaded programming. No interviewees worked on either THREADSAFETY or TSAN. Four of the participants were representatives of teams that use THREADSAFETY. We asked the maintainers of THREADSAFETY for leads to teams that had emailed with questions about the tool. The remaining three participants are only semi-affiliated with a specific team and often act as concurrency expert consultants. We identified these participants by asking Google developers about concurrency experts.

2.3 Limitations

This study has a number of limitations that may hamper generalizability. It was performed at one company, with members of only a few teams. We only were able to look at two tools, focused on one language.

3. A Look at Four Teams

We asked members of four teams to describe how their team was using THREADSAFETY and TSAN.

3.1 Team A

This team has nightly runs of TSAN, but uses THREADSAFETY primarily for documentation purposes. They estimate that approximately 1% of their code is annotated; the most useful annotations for them are `guarded_by` annotations. By their estimate, they detect at least 1 race per 10 weeks with TSAN, and find this a clear win for continuing to use the tool.

> *"When I write code, I add annotations where I think it is useful."*
>
> — Interview Participant

> *"We write enough concurrent code [so that] we are good at writing code without races."*
>
> — Interview Participant

3.2 Team B

This team recently added threading to their project. They cite the static analysis as a key tool in adding threading with confidence. They went through all their core libraries and added annotations. In fact, they even wrote a tool which analyzes source code for likely mutexes, and makes sure that there is at least one annotation using every mutex. They also run TSAN regularly to check for any missed data races.

> *"The whole team would be very positive about it [*THREADSAFETY*]. It is excellent for the easy case. "*
>
> — Interview Participant

3.3 Team C

This team experimented with adding the static annotations to their project, but did not find any major bugs this way. Currently their project contains some annotations but they do not run the analyzer often. Their system is in a state where the core synchronization is not changing very often. Because of this, they did not find the payoff to be large enough to use THREADSAFETY regularly. Also, they encountered some bugs in the analysis implementation which stymied efforts to annotate. They had never heard of TSAN.

> *"Given the relatively small amount of information it gave, I am not sure it is worth it to run it regularly."*
>
> — Interview Participant

> *"[The annotations are a] great thing to have...I would use them for writing new code."*
>
> — Interview Participant

3.4 Team D

This team has annotated portions of their codebase, but only checks the annotations sometimes. They prioritize analysis of tricky code, and only annotate and run the analysis when they know there is a multithreading-related issue. They ran up against expressibility limitations of the static annotations when they tried to annotate their own custom locks, which limited the applications of THREADSAFETY. They had not heard of TSAN.

> *"Our experience with the tool [*THREADSAFETY*] is that this is one of our few tools for gaining confidence that we do not have multithreading bugs. Does not guarantee there are no such problems."*
>
> — Interview Participant

3.5 Discussion

When we started this project, we thought that there would be one "right" way to use concurrency analyses at Google. The interviews gave us an interesting picture of the different ways that teams may chose to use static and dynamic concurrency analyses. For THREADSAFETY, teams ranged from occasionally using annotations for documentation purposes to making a concerted effort not to ever commit unannotated code. For TSAN, teams ranged from never having used the tool to automating nightly runs.

4. Themes

We identified a variety of themes from the interview data focused on TSAN specifically, THREADSAFETY specifically, or else both (General).

4.1 General

Finding and debugging concurrency errors is still hard. Although developers were overall positive about both THREADSAFETY and TSAN, both tools have limitations.

> *"We still get race conditions getting into production and crashing things."*
>
> — Interview Participant

That said, it was clear from the interviews that there are best practices for multithreaded programming, including using tooling such as THREADSAFETY and TSAN.

Reproducibility is very important. If a bug is reproducible, developers are willing and able to fix concurrency errors.

> *"If you can reproduce a bug quickly, we can fix it, even if it is arcane and non-deterministic...We don't have tools for the once every 24 hours in a 100 machine cluster. "*
>
> — Interview Participant

Team culture matters. Use of the tools needs to be supported by a team culture; individual programmers were unlikely to use them consistently without this support.

There is a tradeoff between races and deadlocks. Deadlocks are a bigger issue for some teams, and races are for others. The difficult races are those that cause subtle inconsistencies, or are not easily reproducible.

> *"Crash is easy, inconsistency is hard."*
>
> — Interview Participant

Performance is hard to get right. One tradeoff not well discussed in the documentation for the analysis tools is that of performance. Holding locks across expensive operations is a big problem. Also, it can be difficult to safely break locks apart.

> *"Most of the time they get the locking right, but it is not fast."*
>
> — Interview Participant

Manual inspection is still a powerful tool. We noticed that THREADSAFETY and TSAN were implicitly being compared with a third tool: manual inspection. For infrequent data races, careful manual inspection of suspect code, sometimes with the assistance of a concurrency expert, is the only way that people were able to make traction. In fact, sometimes manual inspection is the best of all.

> *"I found more problems by hand going through the code."*
>
> — Interview Participant

Developers build effective mental models from clear documentation. The importance of good documentation came up repeatedly in the interviews. Clear, strict, straightforward documentation was listed as really important, particularly for static analyses. Developers needed to build their own mental models of how the tool worked in order to use it effectively.

Low false positive rates are critical for adoption. Another theme that repeatedly emerged was the importance of a low false positive rate. Developers find their own impression of the false positive rate of a tool, and take action based on this impression. A lower false positive rate encourages developers to be proactive – they may even fix extra bugs.

> *"We take this seriously because we have seen very little false positives. We even fix it if it is not our code."*
>
> — Interview Participant

Your true positive is my false positive. Interestingly, what constitutes a false positive is a matter of perspective. For example, while most participants considered the false positive rate for THREADSAFETY to be very low, a couple of participants rated THREADSAFETY as having a much higher false positive rate. These participants considered an error message about missing annotations to be a false positive, even if the error message is correct. In other words, they considered "bugs" in the *annotations* to be false positives, since they were not indicative of bugs in the *code*. In contrast, researchers typically consider missing annotations to be true positives; the annotations should correctly describe the code.

Two (sometimes unexpected) causes of thread safety issues came out during the interviews:

Code complexity is underestimated. Thread safety issues occur in code which was perceived to be simpler than it actually is; for example, code that has not been well reviewed or in which the shared state was unanticipated.

> *"Most thread safety issues are in code that does not have any notion of thread safety."*
>
> — Interview Participant

> *"Lots of bugs happen when we try to be cute."*
>
> — Interview Participant

Dependencies and ownership can hinder fixes. One particular challenge is when concurrency errors occur in dependencies or with interactions between dependencies. These situations can be challenging to fix, since the problematic code is not owned by the team that is running into problems.

> *"We had a deadlock between two unrelated components neither of which are ours."*
>
> — Interview Participant

4.2 Dynamic

TSAN find common races and is easy to understand. Our interviewees found TSAN to be better at common case races and finding bugs in existing code than THREADSAFETY. This tool was useful when testing code, although it requires good test coverage. The developers also appreciated the precise error messages. The output from TSAN is easy to triage, and developers only investigate potential races after closely examining reports.

"If stack trace looks pretty serious and the warning is in own code, then investigate further."

— Interview Participant

4.3 Static

THREADSAFETY works well for the easy case, but has trouble with harder idioms. THREADSAFETY helps find some bugs, and is excellent for the easy case. There is a surprising amount of code that fits into this easy case; for example, functions which acquire a lock, access a shared variable, then release that lock. It is often not ambiguous where THREADSAFETY can be applied, although one team we talked to did have to back out of an attempt to annotate some low-level libraries after discovering portions of the library which did not fit well with the THREADSAFETY model.

"When it works it is just kind of trivial. Cases where it didn't just didn't fit the model."

— Interview Participant

THREADSAFETY helps with confidence and understanding. In general, developers found the annotation burden to be low, and were willing to add them (at least partially). Using the annotations gave developers confidence in their concurrent code. The annotations most importantly help programmers *understand* that the code is safe on a local level. Furthermore, THREADSAFETY has high coverage (when it applies). Most people did not encounter a lot of false positives when running THREADSAFETY, unless you consider having to change the annotations to be a false positive. Once in place, the annotations could help developers think less about the synchronization discipline. One developer we talked to used the annotations to help develop code by running the analysis and adding suggested fixes until the program compiled. When the synchronization discipline is relatively straightforward (e.g. the case where you have a class with a mutex and a shared variable as fields and you just want to make sure the variable is consistently protected by the mutex), this sort of analysis-driven development works well.

THREADSAFETY provides enforceable documentation and discipline. The annotations provide documentation (particularly important for interfaces). Furthermore, the documentation provided by the annotations was labeled as strictly better than comments because it could in theory be analyzed, even by teams which did not usually run the analysis. Writing the code with the annotations also forced developers to be aware of the concurrency structure.

"The thing about the static checking is that it causes people to have discipline. They have to say what they are expecting to have happen."

— Interview Participant

Annotating legacy code is a lot of work. THREADSAFETY requires extra work for legacy code; all libraries need to be annotated. This annotation step involves a time investment including extensive reading of the code in order to annotate it correctly. Annotating was recognized as being a lot of manual work, despite the fact that the annotation burden was recognized as being relatively low. A robust annotation inference system was on every team's wishlist.

Some patterns are not expressible in THREADSAFETY. There are also expressibility limitations to the annotation language that prevent it from being universally applicable. Sometimes it is difficult to write out where the mutex is. For example, dynamic lock acquisition, distance between mutex and the data it is protecting, or functions that are called with differing locks.

Some patterns are changed to accommodate THREADSAFETY. Using THREADSAFETY is sometimes clunky. The annotations may change the way the code must be written (in a way orthogonal to concerns such as readability) so as to better accommodate them. There were also differing opinions about the effort required to use the annotations.

"Most of the time we change the design to make it easier to do the lock annotations. Not clear new design better or worse."

— Interview Participant

Also, some developers thought the annotations made the source code cluttered. When both dynamic and static annotations/assertions are used, the clutter may be magnified.

Limitations of THREADSAFETY affect its usability. Bugs inside of THREADSAFETY and unclear annotations (such as `locks_excluded`, which denotes that the function must *not* hold the specified lock) severely limit the analysis usability. They can also lead to bad practices inside the code such as adding locking in strange places to deal with incorrect analysis.

"I found code where someone took a lock out on an object in the destructor. This is not a good pattern! This was an attempt to fix a timing bug, that will eventually bite someone."

— Interview Participant

Misusing annotations can also be problematic. Developers may forget to add annotations. Also, incorrect annotations on low-level libraries can have far-reaching effects [5].

5. Related Work

To the best of our knowledge, there has been little to no prior work on how developers use concurrency-focused program analysis tools. In this section, we briefly describe the greater research context to THREADSAFETY and TSAN. We also highlight a few papers that have looked at how developers use analysis tools or debug concurrent programs.

5.1 Static Analyses for Data Race Detection

There is a long history of research on static race detection. Here we briefly mention several well-known techniques.

Previous concurrency-focused type and effect systems find races by encoding the synchronization discipline of a program into the type system [1, 4]. Although type systems scale better than other static analyses and can prove the absence of races, they require extensive developer annotations. The annotations used in THREADSAFETY [23–25] are based on a such a type system against races [1]; THREADSAFETY makes the compromise that there may be false negatives, but the annotation burden is more flexible.

Automatic approaches attempt to find races without the aid of developer annotations. RACERX runs a static lockset algorithm using a flow-sensitive interprocedural analysis [6]; a *lockset* is the set of locks held at a given memory access, and a lockset algorithm checks if a consistent lockset is held for a given memory access throughout the program. RELAY also computes a static lockset, but uses *relative locksets* in function summaries to improve scalability [33]. Both RACERX and RELAY scale to analyze an operating system kernel, but are unsound (may have false negatives). LOCKSMITH is a flow-sensitive, context-sensitive analysis for C programs that implements a static lockset algorithm, and is based on a sound type system [22]. CHORD uses a flow-insensitive, context-sensitive analysis [17] for scalability and conditional must-not aliasing [16] to ensure soundness (no false negatives) for Java programs.

5.2 Dynamic Analyses for Data Race Detection

Dynamic race detection techniques can be largely divided into happens-before approaches and lockset approaches. A relatively small class of data race detectors aim to actually catch data races in the act, e.g., through hardware watchpoints [7]. Modern happens-before [13] race detectors use vector clock [15] representations to gain efficiency, and are precise (no false positives). The FASTTRACK algorithm improves efficiency further by optimizing the analysis on most operations from $O(n)$ to $O(1)$, where n is the number of threads [9]. Lockset race detectors enforce a lock-based synchronization discipline, where each access to shared memory is expected to be consistently protected by a lock. Although efficient, these detectors are typically imprecise (may have false positives). The ERASER algorithm uses a state machine to reduce the number of false positives [27]. Hybrid techniques combine happens before and lockset approaches to improve precision and efficiency [21, 34].

The TSAN algorithm is a happens-before algorithm, and is implemented as a compiler instrumentation pass in LLVM [30] for higher performance.

5.3 How Developers Use Analysis Tools

Formal studies that investigate how developers use program analysis tools are practically non-existent. A recent study looks at how Linux kernel developers respond to static analysis bug reports [12]. However, the kernel developers studied did not actually use the analysis tool; they were presented with bug reports in a questionnaire format. An older study investigated whether static analysis can economically help improve the quality of software products at a commercial organization [35]. This report describes, at a high level, how static analysis tools were used in the organization, in terms of the development pipeline.

In addition, very little work has been done at the intersection of parallel programming and user evaluation: even less with professional developers [26]. For his thesis, Scott Fleming observed programmers debugging a multithreaded server application that had been seeded with a concurrency bug [10, 11]. Interestingly, several participants were able to find and fix the flaw, but were not able to correctly describe the design defect that caused the problem. However, these programmers did not use data race detection tools to identify the errors.

6. Discussion and Conclusion

Every team we interviewed that was using THREADSAFETY or TSAN explicitly said they thought it was a net positive, despite any problems they had with the analysis. In general, TSAN seemed to find more insidious bugs than THREADSAFETY. On the other hand, THREADSAFETY was most useful for gaining understanding of and confidence in concurrent code. However, since developers must fix warnings from THREADSAFETY before checking in the code, it is difficult to assess how severe these problems would have been had they been released to production. The biggest issues developers had with TSAN were the slow speed and lack of coverage. The biggest issues developers had with THREADSAFETY were the difficulty of annotating legacy code, expressibility limitations of the annotation language or analysis, and clutter or tedium caused by annotations. Interestingly, some teams use partial sets of THREADSAFETY annotations to mitigate these issues.

There are many factors that may affect team adoption of concurrency analysis tools. We found the team culture had an impact on how developers thought of the analysis tools. Also, different tools may be effective for legacy code than for writing new code. For example, most developers we talked to said they would want to use THREADSAFETY on a new project, even if they were not willing to annotate all their existing code to employ it now. Ways to migrate legacy code smoothly onto new systems (e.g. annotation inference) could be very valuable. For a prospective tool, having a low false positive rate and good documentation is very important. The tool performance directly impacts how often a tool is run, and so a slow tool may find less bugs since it is run less often. Developers valued annotations that were useful for documentation, but did not like clutter. Ideally, annotation-based analyses will involve few annotations that describe a

large amount of idioms concisely and powerfully. Helping developers *understand* (and gain confidence in) concurrent programs is just as important as finding new bugs in them; there is an opportunity for better tooling and visualization methods here.

Even simple production-quality tools may be really useful; there are a lot of easy things to check that could be verified by robust tools, such as error-prone [2]. There were also some tricky bugs that came up that we do not currently have good tooling for. For example, the race that occurs once in every 24 hours in a 100 machine cluster, or the deadlock that happened because of an unexpected interaction between dependencies. Enriching standard debugging information, such as by adding information about locking to stack dumps, could make it easier to track down errors. Developers were also very interested in extending THREADSAFETY in various ways, such as to include inter-procedural analysis, building an inference system, or enriching it with the ability to write higher-level invariants about how combinations of locks interact.

Concurrency analyses could explain bugs better (e.g. with failure traces). Developers need to understand the errors and believe they are errors, otherwise they may distrust and abandon the tool. A few people suggested combined static and dynamic analysis could improve upon both techniques; for example, it may be possible to leverage annotations to decide when lock acquires happen. Lastly, it may be possible to extend the effectiveness of manual inspection for finding concurrency bugs by performing a structured inspection, like in heuristic evaluation [20].

> *"We wish we had something better, but we don't want less. The problem is too hard without it."*
>
> — Interview Participant

Acknowledgments

We wish to thank Robert Bowdidge, Kostya Serebryany, Timur Iskhodzhanov, Dmitry Vyukov, DeLesley Hutchins, and our helpful reviewers for valuable information and feedback.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible compiler. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, 2012.

[3] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2006.

[4] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[5] Chromium Team. Issue 15577. Available from `http://code.google.com/p/chromium/issues/details?id=15577`, 2012.

[6] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[8] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[9] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[10] S. D. Fleming. *Successful Strategies for Debugging Concurrent Software: An Empirical Investigation*. PhD thesis, Michigan State University, 2009.

[11] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In *International Conference on Software Engineering (ICSE)*, 2008.

[12] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conference*, 2009.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[15] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.

[16] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Symposium on Principles of Programming Languages (POPL)*, 2007.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[18] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[19] B. Nicodemus and L. Swabey. *Advances in Interpreting Research: Inquiry in Action*. John Benjamins Publishing Company, 2011.

[20] J. Nielsen. Heuristic evaluation. In J. Nielsen and R. L. Mack, editors, *Usability Inspection Methods*, pages 25–62. Wiley, 1994.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[22] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[23] C. Sadowski. Usage of thread safety attributes. Available from http://lists.cs.uiuc.edu/pipermail/cfe-dev/2011-July/016144.html, 2011.

[24] C. Sadowski. Proposal for thread safety attributes for clang. Available from http://lists.cs.uiuc.edu/pipermail/cfe-dev/2011-June/015899.html, 2011.

[25] C. Sadowski and D. Hutchins. Thread-safety annotation checking. Available from http://clang.llvm.org/docs/LanguageExtensions.html#threadsafety, 2011.

[26] C. Sadowski and A. Shewmaker. The last mile: Parallel programming and usability. In *FSE/SDP Workshop on the Future of Software Engineering Research (FoSER)*, 2010.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[28] C. B. Seaman. Qualitative methods in empirical studies of software engineering. In *IEEE Transactions on Software Engineering*, 1999.

[29] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[30] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler. In *International Workshop on Runtime Verification (RV)*, 2011.

[31] ThreadSanitizer Team. ThreadSanitizer. Available from http:// code.google.com/p/data-race-test, 2012.

[32] ThreadSanitizer Team. ThreadSanitizer v2. Available from http://code.google.com/p/thread-sanitizer, 2012.

[33] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *International Symposium on Foundations of Software Engineering (FSE)*, 2007.

[34] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[35] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. In *IEEE Transactions on Software Engineering*, 2006.