SRC Technical Note
2002 - 004

14 May 2002

Finding stale-value errors in concurrent programs

Michael Burrows and K. Rustan M. Leino

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://research.compaq.com/SRC/

Copyright ©Compaq Computer Corporation 2002. All rights reserved

Abstract

Concurrent programs can suffer from many types of errors, not just the well-studied problems of deadlocks and simple race conditions on variables. This paper addresses a kind of race condition that arises from reading a variable whose value is possibly out-of-date. The paper introduces a simple technique for detecting such stale values, and reports on the encouraging experience with a compile-time checker that uses the technique.

Author Affiliations

Michael Burrows is currently working at Microsoft Research, 1065 La Avenida, Mountain View, CA 94043, USA.

K. Rustan M. Leino is currently working at Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.

0 Introduction

Writing correctly behaving software programs is hard. Among many kinds of errors, errors due to concurrency are notoriously costly. A reason that concurrency errors stand out is that it is difficult to devise test suites that will provoke the erroneous behaviors. Therefore, the prospect of using special program checkers for detecting concurrency errors has many appeals.

Of course, there are costs involved in using a program checker, too. For example, the checker may take a long time to run, either affecting the running time of the program itself, or taking a long time to perform analyses at compile time. Another possible cost associated with a program checker is that the warning messages it produces may be may be difficult to decipher or be too voluminous. If much of the checker's output is seen by the user as unquenchable noise, the checker is not likely to see much use. Yet another cost, which agitates many users, is that the checker may require users to spend time providing the checker with properties of the program design that otherwise are not directly evident. These extra properties are typically provided by adding executable calls into the checker's run-time support library or by supplying compile-time annotations.

Previous work has studied ways of detecting or preventing race conditions where variables have not been consistently protected by locks, and deadlocks. In this paper, we address a different kind of race condition. The error can occur even when each shared-variable access is correctly protected by a lock. The error arises when a lock is held at the time a value is read from a shared variable, but not held at the time the value is later used. We don't know of any previous practical checker that detects this kind of error.

We present a simple checking technique for detecting these *stale-value errors*. The technique can be used with either a run-time checker or a compile-time checker. We have built a compile-time checker based on the technique. We have found this checker to perform adequately, to require no annotations, and to produce few enough warnings that all of them can easily be inspected by hand. Moreover, the warnings produced by our tool have pointed to new errors in some otherwise well-tested programs.

1 Stale values

In this section, we give three examples. The first example shows a simple race condition that previous techniques address, but our technique does not. The other two examples show a different kind of race condition, the kind that our technique addresses.

A common paradigm when writing concurrent programs with shared variables is for a thread to access a shared variable only when the thread holds a designated lock. When a thread acquires a lock, it is said to enter a *critical section*; it remains in the critical section while it holds the lock; and it leaves the critical section when it releases the lock. A critical section is also called a *monitor* [9], so we call this programming paradigm the *monitored-value paradigm*. The monitored-value paradigm avoids simple race conditions, that is, the possibility of two or more threads reading and writing a shared variable at the same time, which can result in data corruption. The monitored-value paradigm also has the nice property that it can be checked, either at compile time or at runtime [12, 5, 7, 11]. As an example, consider the following program, where x is a shared variable and $t0$ and $t1$ are local variables:

$$
\begin{array}{l}
enter\_critical\_section();\\
\quad t0 := x;\\
exit\_critical\_section();\\
t1 := x;
\end{array}
$$

Because the first access of x occurs inside the critical section, other threads obeying the monitored-value paradigm will be excluded from accessing x. However, the second access of x occurs outside the critical section and therefore mutual exclusion for accessing x is not ensured. This means that another thread may be writing the variable x at the time x is being read into $t1$ by the thread displayed above, which would constitute a race condition.

Now consider a second, more complicated example:

$$
\begin{array}{l}
enter\_critical\_section();\\
\quad t0 := x;\\
exit\_critical\_section();\\
t1 := t0;\\
enter\_critical\_section();\\
\quad t2 := t0;\\
exit\_critical\_section();\\
t3 := t0;
\end{array}
$$

where x is a shared variable and $t0$, $t1$, $t2$, and $t3$ are local variables. Here, the access of shared variable x occurs inside a critical section, so previous checkers would consider the program free of simple race conditions. However, the snapshot of x's value that is read into $t0$ in that critical section is used three more times.

The first of these uses, which for pedagogic simplicity is shown here as an assignment to $t1$, is justifiable: a programming methodology must support using a snapshot of a shared variable outside a critical section. If such a snapshot were never allowed outside a critical section, then a program that printed an account balance stored in a shared variable would have to remain in the critical section while the printer put the ink on the paper. But a better program would enter the critical section, store a snapshot of the shared variable into a local variable, exit the critical section, and then print the value from the local variable.

The second use of the snapshot $t0$ of x (the assignment to $t2$) is more dubious. This use occurs inside a critical section, yet it is using the value of x from the previous time the thread was in a critical section. Between the two critical sections, other threads may acquire the lock and update the value of x, in which case $t0$ is a *stale* copy of x. Sometimes, this is indeed what the programmer intended. For example, it may be that the program compares $t0$ with x to see if x has changed, which may enable some optimization. However, one should be suspicious of such stale values, because they may well have arisen from accidental uses of the wrong variable. This error would be hard to catch using testing because it manifests itself only if another thread updates x between the two critical sections.

The third use of the snapshot $t0$ of x (the assignment to $t3$) is also dubious, probably even more so than the second use. Here, a code reviewer would ask, "Why does $t0$ hold the snapshot of x from the first critical section, why not a snapshot from the second critical section?".

For our third example, we consider the case where lock acquisitions and releases are hidden in procedure calls. Some procedures may for example first acquire a lock and then release it, while others, such as the $wait()$ operation on condition variables (see *e.g.* [9, 2]), first release a lock and then re-acquire it.

Here is the code the for $consume$ operation in a producer/consumer application (see, *e.g.*, [2]), which uses $wait()$ to temporarily exit the critical section to yield to producers:

$$
\begin{array}{l}
enter\_critical\_section();\\
\quad a := arr;\\
\quad \mathbf{while}\ (n = 0)\ \{\\
\quad\quad wait();\\
\quad \}\\
\quad v := a[nextc];\\
\quad nextc := (nextc + 1)\ \mathbf{mod}\ a.length;\\
\quad n := n - 1;\\
exit\_critical\_section();
\end{array}
$$

In this example, n, arr, and $nextc$ are shared variables, denoting the number of elements produced but not yet consumed, a reference (pointer) to the array that holds those elements, and the index into that array where the next element to be consumed is, respectively, and a and v are local variables. The code retrieves the next element to be consumed into variable v. The array is used as a cyclic buffer, so the increment of $nextc$ is performed modulo the size of the array.

The consumer code contains an error: the reference arr to the array is copied into local variable a before the $wait()$ operation, and it is this copy that is used to do the array operations after the $wait()$. Hence, if many producers get control during the $wait()$ call, so many that some producer needs to allocate a larger array, copy the elements from the previous array into the new one, and update arr to point to the new array, then the consumer code above will end up using the old array instead of the new one, which would result in v getting set to the wrong element. Note that this insidious error cannot be found by previous techniques, because the shared variables are all accessed under the protection of an appropriate lock.

In summary, programs can suffer from concurrency errors other than deadlocks and simple race conditions: we have shown how using stale values of shared variables can also be a source of error. Next, we turn to the issue of how to detect such errors in programs automatically.

2 Detecting stale values

In the most simple and common case, stale values are propagated in local variables. Therefore, our technique tracks the assignments to and uses of local variables. We proceed by instrumenting the original program with some additional local-variable declarations and program statements. These new declarations and statements are added to the original program. To distinguish them from the original program, we call the additions *ghost* declarations and statements. Our ghost statements do not change the functional behavior of the original program, except for the actions taken by a ghost assertion that fails, which occurs if a possible stale-value error is detected.

We start by introducing, for every local variable t in the original program, a boolean ghost variable $stale\_t$ with the following meaning:

Invariant J0: $stale\_t$ is $true$ if and only if the value of t is considered stale.

Note that we may have some leeway in just exactly what we consider to be stale.

Immediately preceding every read of local variable t in the original program, we add the following ghost assertion statement:

$\textbf{assert}\ \neg stale\_t;$

If this assertion fails, the value stored in t is considered stale, and so the checker produces a warning.

To establish Invariant J0 initially, at the point where t and $stale\_t$ are declared, $stale\_t$ is set to $false$. That is, we add the ghost statement:

$$stale\_t := false;$$

To maintain Invariant J0, we need to update $stale\_t$ in two situations: when t is assigned, it may become fresh (not stale), and when the thread enters a critical section, t may become stale.

So first, $stale\_t$ needs to be updated at assignments to t. A simple, somewhat coarse approach is to add the ghost assignment:

$$stale\_t := false;$$

at every assignment

$$t := E;$$

This simple approach would treat the new value of t as fresh regardless of the right-hand side expression E. A more precise approach would take E into account. For example, for the assignment statement

$$t := t + 1;$$

a more precise approach would not change the value of $stale\_t$, since the new value of t may be derived from a stale value.

Second, $stale\_t$ needs to be updated on entry to a critical section: if t's current value comes from a previous critical section, then $stale\_t$ should be set to $true$. To accomplish this, we need to introduce another ghost variable, $from\_critical\_t$. Ideally, $from\_critical\_t$ is $true$ just when t has received its current value inside a critical section from a shared variable. We will settle for something simpler:

Invariant J1: $from\_critical\_t$ is $true$ if and only if the last write to t occurred inside a critical section.

Initially, $from\_critical\_t$ is $false$. To maintain Invariant J1, we add an update of $from\_critical\_t$ whenever there's an assignment to t. If the assignment occurs in a critical section, we add:

$$from\_critical\_t := true; \tag{G0}$$

otherwise, we add:

$$\mathit{from\_critical\_t} := \mathit{false}; \tag{G1}$$

With $\mathit{from\_critical\_t}$ and Invariant J1 at our disposal, we put the finishing touches on maintaining Invariant J0: on entry to a critical section, we add the following ghost statement:

$$\textbf{if } \mathit{from\_critical\_t} \textbf{ then } \mathit{stale\_t} := \mathit{true} \textbf{ end} \tag{G2}$$

This maintains Invariant J0, because if on entry to a critical section the last assignment to t occurred in a (previous) critical section, then the current value of t is now to be considered stale.

This concludes the general description of our technique. The technique can be implemented in a dynamic checker. That is, by instrumenting the original program with the ghost statements above, one can detect possible stale-value errors through failing ghost assertions. Our technique can also be implemented in a compile-time checker, where a warning is produced if the checker detects a possibility or certainty that a ghost assertion will fail.

3 Implementation

We have implemented our technique in a compile-time checker. We built the checker as an extension to the Compaq Extended Static Checker for Java (ESC/Java) [8, 10]. ESC/Java translates a program into a *verification condition*, a logical formula that, ideally, is valid if and only if the program is free of the kinds of errors under consideration. The verification condition is then passed to an automatic theorem prover, which searches for counterexamples. Each counterexample is turned into a message that warns the programmer of a possible error in the program. ESC/Java performs *modular checking*, checking one method body at a time, much like a compiler performs separate compilation. The heart of ESC/Java's analysis comes down to checking the validity of assertions, which makes it a good engine for our stale-value detection technique. In this section, we describe some specific considerations of our implementation and how it pertains to Java.

A major design consideration that we omitted from the previous section is the fact that programs have more than one lock. In Java, locks may be dynamically allocated and assigned, so a syntactic scan of the $\mathit{enter\_critical\_section}()$ statement that acquires a lock is not in general sufficient to determine which lock is being acquired. Aiming for a simple initial design, our checker mostly ignores which lock is being acquired. That is, when in the description above we said "on entry to a critical section", our checker adds the same ghost statements regardless of which lock is acquired. This simple design may result in more stale-value warnings than a more elaborate design. However, in trying out this simple initial design, we found the number of reported warnings to be so low that it did not seem worth the time to design something more elaborate, except at calls to `synchronized` methods, as we describe next.

Another design consideration in our checker is that Java has a modifier called `synchronized` that can be applied to methods. Although this modifier looks like it is part of the signature of the method, it is simply a convenient way to turn the body of a particular method implementation into a critical section. Nevertheless, by looking at this method modifier from call sites, we have incorporated some special behavior in our checker.

A call to a `synchronized` method will enter and exit a critical section. If the called method is declared in the *same abstraction* as the caller—meaning in a superclass or subclass of the class enclosing the call—and the call site is not itself in a critical section, then our checker will treat the call as entering and exiting a critical section for the purpose of checking for stale-value errors. This has two implications.

The first implication regards entry to the call. Our checker treats local variables mentioned in actual parameters of the call as though they were read by the callee. To catch errors where the values of these local variables would be considered stale inside the critical section of the callee, we add ghost statement (G2) just before evaluating the actual parameters.

The second implication regards return from the call. Any result value of the called method is returned from within a critical section. Therefore, we may need to update $\mathit{from\_critical\_t}$ if some local variable t receives a value derived from the result value of the call. In particular, for any assignment of an expression E to a local variable t, if E contains a call to a `synchronized` method of the same abstraction, then we add (G0) instead of (G1) at the time t is updated.

A final design consideration is the use of condition variables and the $wait()$ operation that can be applied to these. The $wait()$ operation temporarily exits a critical section, waiting for a corresponding $signal()$ operation. On return from the call to $wait()$, our checker models the re-entry to the critical section by adding ghost statement (G2).

We mention one more fine detail that pertains to our checker. When above we mention an assignment of $true$ to $\mathit{stale\_t}$, our checker actually assigns an arbitrary value to $\mathit{stale\_t}$. The reason for this is as follows: When ESC/Java encounters an assertion $\textbf{assert}\ E$, it generates a warning if E is not always true, and continues checking if E is not always false. For the further checking downstream of this point, the checker will assume $\neg E$. By assigning to $\mathit{stale\_t}$ an arbitrary value, one obtains the functionality of checking the condition $\neg stale\_t$ correctly and retains the benefit of being able to do downstream checking.

We have described the various ghost statements above for local variables. It would make sense also to treat formal parameters in this way, since a formal parameter in Java method body can be used as a local variable. In fact, one can also imagine giving the same treatment to the temporary variables that ESC/Java introduces to hold values of intermediate subexpressions. However, our implementation treats only local variables.

Finally, we show what a warning message produced by our checker looks like. If the third example from Section 1 were written in Java in a file `ProduceConsume.java`, then our checker would produce a warning like:

```
ProduceConsume.java:82: Warning: Possible use of
stale value of monitored entity (StaleValue)
    v =  a[next];
         ^
```

where 82 is the source line number where the error was found, and the caret indicates the position within the line.

4 Experience

We have applied our stale-value checker to several concurrent Java programs. We have found that the checker produces a modest number of warnings, including some real errors in the programs.

The Table shows the results from running our checker. The columns are: the name of the program being tested; the number of thousands of lines of code; the number of false alarms generated by the tool; the number of warnings that indicated benign races; and the numbers of warnings that indicated bugs in the code.

Here are brief descriptions of our test programs: *pachyclient* is an applet to assist in reading mail; *mercator* is a web crawler; *webl* is the implementation of the language WebL; *ambit* is the implementation of a language used to describe agents; *cluster* finds groups of related web pages; *jigsaw* is the World-wide web Consortium's reference implementation web server; *sys man A* and *sys man B* are proprietary programs for managing configurations of computers and peripherals.

We are encouraged by these results, because even with no annotations we obtained fewer than 1 warning per 10,000 lines of code, and yet found 4 previously unknown bugs.

		warnings		
program	kloc	false alarms	benign races	bugs
pachyclient	11	6	0	1
mercator	27	9	0	0
webl	23	5	0	0
ambit	3	2	0	0
cluster	3	0	0	0
jigsaw	125	20	1	0
sys man A	325	1	0	3
sys man B	100	0	0	0
Total	617	43	1	4

Table: The results of running our checker over various programs. The table shows the program sizes and the number of warnings produced. The warnings are broken up into three categories. We are unsure of the categorization for *ambit* and *jigsaw*, because we did not consult the authors of the code.

Our prototype checker is rather slow. It can check about 2000 source lines per minute. The speed could be greatly improved, because our prototype contains code intended for more heavy-weight checking. We believe that a special-purpose checker could be nearly as fast as the type checker.

5 Related work

Much work and several different techniques have targeted the detection and prevention of simple race conditions. For example, Warlock [12] uses static data flow techniques, ESC uses program verification techniques [5], and the Race Condition Checker for Java [7] and the programming language Guava [0] use type checking techniques. Eraser [11] and Nondeterminator-2 [3] analyze dynamic executions, finding races beyond those evident in the given execution. Some new work by Choi *et al.* [4] allegedly improves precision and performance by some combination of static and dynamic techniques.

The meta-level compilation technique in *metal* [6] has been used to implement checks for various locking-related errors. In some sense, our technique is similar in flavor to some of these checks, in that we search for things that look suspicious. Perhaps metal would be a good framework in which to implement our technique. The SLAM Toolkit [1], which uses model-checking and theorem-proving techniques, also finds various locking-related errors in software.

Our technique is complementary to these tools, languages, and techniques: we find errors they don't, and vice versa.

We don't know of any previous practical checker that detects stale-value errors, but there is a previous technique that can find them: program verification. For illustration, consider the following program:

$$
\begin{array}{l}
\mathit{enter\_critical\_section}(); \\
\quad t := a.\mathit{length}; \\
\mathit{exit\_critical\_section}(); \\
\mathit{enter\_critical\_section}(); \\
\quad s := 0;\ i := 0; \\
\quad \textbf{while}\ (i < t)\ \{ \\
\qquad s := s + a[i]; \\
\qquad i := i + 1; \\
\quad \} \\
\mathit{exit\_critical\_section}();
\end{array}
$$

where a is a shared variable and t, s, and i are local variables. This program contains a stale-value error, since the use of t occurs in a different critical section than its definition from a shared variable. Our technique will find this error.

The loop in the second critical section relies on the invariant $t \leq a.\mathit{length}$, which the first critical section establishes. But a program verifier that supports monitors would not allow the property to be inferred in the second critical section, because the value of $a.\mathit{length}$ seen by the current thread can change arbitrarily while outside the monitor. Such a program verifier would not be able to prove the access $a[i]$ correct, thus finding the error.

The Extended Static Checkers for Modula-3 [5] and for Java [8] use program verification technology and can find errors involving monitors and shared variables. However, these two checkers do not take into consideration the effects of other threads between critical sections, because doing so would place an additional burden on the tool user to declare *monitor invariants* [9] in order to suppress otherwise-spurious warnings. In the design of these checkers, this additional annotation burden was perceived to outweigh the benefit of finding the additional errors, and thus the checkers consider only those executions where the values of shared variables remain unchanged between critical sections [5, 10].

6 Summary and future work

In summary, we have discussed a kind of race condition, a *stale-value error*, that can arise in concurrent programs. We have introduced a simple method for detecting many stale-value errors. The technique can be used in either a dynamic checker or a compile-time checker. In either case, the technique does not require user annotations. We have implemented the technique in a compile-time checker for Java. We have applied the checker to several hundred thousands of lines of code and have found it to produce a small number of warnings that are nevertheless of high quality.

A possible improvement on our technique is to replace the boolean $stale\_t$ by a counter age_t with the following meaning:

Invariant J2: If $from\_critical\_t$ is $true$, then age_t is the number of critical sections that have been entered since the variable t was last assigned.

To maintain this invariant, age_t is zeroed when t is assigned, and incremented on each entry to a critical section. When the variable t is used, we can then add the assertion

$$\mathbf{assert}\ \neg from\_critical\_t \lor age\_t = 0;$$

This allows the warnings to be ordered by age. We suspect that warnings with higher age will be more relevant.

Acknowledgements We'd like to thank Andrew Birrell for being dissatisfied with the fact that race condition checkers do not check for stale-value errors, and to Andrew, Marc Najork, Hannes Marais, and Cormac Flanagan for helping evaluate the warnings that our checker produced. We'd also like to thank Lyle Ramshaw for carefully reading the paper.

References

[0] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java with data races. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, volume 35, number 10 in *SIGPLAN Notices*, pages 382–400. ACM, October 2000.

[1] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.

[2] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, January 1989.

[3] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 298–309, June 1998.

[4] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 2002 ACM SIGPLAN conference on Programming Design and Implementation (PLDI)*, 2002. To appear.

[5] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[6] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation*. USENIX, October 2000. Proceedings published at `http://www.usenix.org/events/osdi2000/technical.html`.

[7] Cormac Flanagan and Steven N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN conference on Programming Design and Implementation (PLDI)*, pages 219–232, 2000.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN conference on Programming Design and Implementation (PLDI)*, 2002. To appear.

[9] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[10] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[11] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997. Also

appears in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 27–37, Operating System Review 31(5), 1997.

[12] Nicholas Sterling. Warlock — a static data race analysis tool. In *Proceedings of the Winter 1993 USENIX Conference*, pages 97–106. USENIX Association, January 1993.