SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:1259–1288
Published online 18 January 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1040

Isolating bugs in multithreaded programs using execution suppression

Dennis Jeffrey[*,†], Yan Wang, Chen Tian and Rajiv Gupta

The University of California at Riverside, CSE Department, Riverside, CA 92521, U.S.A.

SUMMARY

Memory-related program failures in multithreaded programs can be caused by a variety of bugs. Concurrency bugs can occur due to unexpected or incorrect thread interleavings during execution. Other kinds of memory bugs, such as buffer overflows and uninitialized reads, may also occur in multithreaded as well as single-threaded programs. Most prior techniques for isolating these bugs are specialized, addressing only one type of concurrency bug or certain types of other memory bugs. The memory corruption caused by these bugs can also undergo significant propagation during program execution. When a program failure finally occurs due to memory corruption, the true root cause of the failure may be effectively concealed as significant portions of memory may have become corrupted. We propose a general framework that can isolate the root cause of any failure in a multithreaded program that involves memory corruption and reveals at least a subset of this memory corruption. This includes three important types of concurrency bugs—data races, atomicity violations, and order violations—as well as other kinds of memory bugs. To account for propagation of memory corruption, our approach uses a dynamic technique called 'execution suppression' that iteratively reveals memory corruption in a failing execution to isolate the true root cause of the failure. Copyright © 2011 John Wiley & Sons, Ltd.

Received 16 June 2009; Revised 23 October 2010; Accepted 25 October 2010

KEY WORDS: software debugging; multithreading bugs; data races; atomicity violations; order violations; memory bugs; execution suppression; memory corruption propagation

1. INTRODUCTION

Software debugging is a difficult but necessary phase of software development. Debugging multithreaded programs in particular can be especially challenging. This is due to the complexity involved in reasoning about the behavior of multiple threads that may execute in parallel. Automated techniques to assist in debugging multithreaded programs can relieve developer burden and increase the efficiency of the debugging process.

Software failures in multithreaded programs can be caused by a variety of bugs. Failures due to concurrency bugs, such as data races, atomicity violations, and order violations, are caused by unexpected interleavings of parallel threads during execution. Other failures may or may not be caused by the multithreading. Failures due to buffer overflows involve accessing memory locations that are outside of proper buffer boundaries. Failures due to uninitialized reads involve reading from memory locations that have not yet been properly written to. Still other failures can be caused by other types of memory bugs.

[*]Correspondence to: Dennis Jeffrey, The University of California at Riverside, CSE Department, Riverside, CA 92521, U.S.A.
[†]E-mail: jeffreyd@cs.ucr.edu

A fundamental challenge to debugging multithreaded (and single-threaded) programs is that the point of an execution failure may be far away from the point at which the root cause for the failure was traversed. The root cause may be a concurrency bug or some other type of memory bug, which may corrupt a memory location. This corruption may propagate arbitrarily far during execution, corrupting possibly many other memory locations, until a failure finally occurs. This memory corruption propagation can effectively conceal the true root cause for a failure.

Prior approaches for isolating bugs in multithreaded and single-threaded programs are often specialized, designed to search for particular kinds of bugs. For instance, *race detection* techniques [1–5] can be leveraged to try to identify data races that may have caused a failure. Other techniques specially handle other kinds of bugs, including *CP-Miner* [6] (copy-paste bugs); *EXPLODE* [7] (data integrity bugs in storage systems); and *Valgrind* [8], *Purify* [9], and *CCured* [10] (particular kinds of memory-related bugs). However, when a failure is observed, a developer often does not immediately know what type of bug caused the failure. Any single specialized bug detector may therefore not be sufficient to debug a multithreaded program. Without knowing what type of bug is to blame, a developer may have to rely on many different kinds of specialized bug detectors. The approach proposed in the current work is a general framework for isolating the root causes of multithreading and single-threaded bugs, which handles three important kinds of concurrency bugs as well as other kinds of memory bugs that cause memory corruption. Specifically, the concurrency bugs handled by our approach are data races, atomicity violations, and order violations. Our approach is designed to be sufficient for isolating many different kinds of bugs in multithreaded programs. Our approach requires that a failure-triggering execution (with associated thread interleaving) is known beforehand, and that the failure can manifest itself in such a way that at least one corrupt memory location causing the failure is revealed. For example, an execution that crashes due to directly accessing a corrupted memory location can be handled by our approach.

Our approach overcomes the issue of memory corruption propagation by using a technique that we call *execution suppression*. This technique iteratively reveals memory corruption within an execution until the root cause for a failure is found. The idea of *suppression* involves ‘nullifying’ or ‘avoiding’ the effects of particular statement instances during the execution of a program. The key idea of execution suppression is as follows. First, it is observed that if the root cause of a failure corrupts one or more memory locations during execution, then the observed failure should reveal a subset of this memory corruption that exists during execution. If the statement instances directly involved in the program failure are suppressed (nullified) during re-execution, and if the statement instances directly or indirectly dependent upon these suppressed statement instances are themselves suppressed during execution, then the original failure will be avoided. Essentially, the *known* subset of memory corruption existing in the execution is suppressed, allowing execution to proceed further, past the point of the original failure. If another failure then occurs later in the execution, this will reveal *more* of the memory corruption that may still exist in the execution, which can be suppressed in turn. Execution suppression iterates in this way until (1) a data race, atomicity violation, or order violation is found to be directly involved in a failure or (2) the suppression execution results in no additional failures, in which case the last-identified point of suppression is reported as the likely root cause of the original failure. The underlying assumption of execution suppression is that if memory corruption exists in an execution, then the execution will result in a failure that reveals at least a subset of this memory corruption.

When determining whether a concurrency bug is directly involved in a failure caused by memory corruption, our approach analyzes particular sequences of memory accesses during execution to check whether the conditions for certain kinds of concurrency bugs are present. As was observed by Tallam *et al.* [11], detection of data races requires considering three particular kinds of memory access dependencies to the same memory location from two different threads: read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies. As was observed by Lu *et al.* [12], detection of atomicity violations requires considering four particular sequences of accesses to the same memory location, each sequence involving two accesses from one thread, interleaved by an access from a different thread: $W_1R_2W_1$, $R_1W_2W_1$, $R_1W_2R_1$, and $W_1W_2R_1$. Detecting order violations similarly requires considering these sequences of memory accesses. Our approach takes all of this into account to identify relevant data races, atomicity violations, and order violations on-the-fly during program execution while suppression is being performed.

Prior work on classifying benign and harmful data races [13] has shown that not all data races are harmful. To determine whether a data race is potentially harmful, their key idea is to replay an execution twice for the data race, each time ensuring that the two conflicting memory accesses occur in a different relative order. If both executions produce different results in memory, the data race is classified as potentially harmful. Since our own approach should only report a data race if it can potentially be the root cause for a failure (i.e. if the data race is potentially harmful), we incorporate their key idea into our approach. Specifically, when a data race is detected during one execution, our approach re-executes the program a second time, forcing the conflicting memory accesses to occur in the reverse order. Our approach only reports the data race as the root cause for a failure if the re-execution produces a different result in memory, suggesting that the data race is potentially harmful. Our approach performs similar re-executions to determine whether potential atomicity and order violations are true violations, before actually reporting them as the root cause for a failure.

The remainder of this paper is organized as follows. In the next section, we describe our approach in detail and give an example that illustrates its use and potential benefit. In Section 3, we describe some of the details involved in implementing our approach. Section 4 presents experimental results showing the benefits of our approach in locating bugs in multithreaded programs including the `apache` web server and the `mySQL` database application. Related work is discussed in Section 5, and our conclusions are summarized in Section 6.

2. THE EXECUTION SUPPRESSION APPROACH

2.1. General approach

Our approach for isolating the root cause of a multithreading bug applies to bugs that involve memory corruption, and that can cause a memory-related program failure that reveals a subset of this memory corruption. The approach handles data races, atomicity violations, and order violations, as well as any other memory bugs that involve memory corruption, including single-threaded memory bugs. This includes bugs such as buffer overflows (including corruption of function call return addresses), uninitialized reads, and double frees. We consider memory corruption to occur during execution when corrupt (incorrect) data is written to a memory location, or when an incorrect memory location is accessed. Intuitively, memory corruption occurs during an execution when memory is mishandled in some way.

Memory corruption resulting from a memory bug can propagate (spread) to other memory locations until ultimately a subset of the memory corruption may directly cause a program failure. The goal of our approach is to automatically isolate the root cause of this memory corruption, which—depending on the degree to which the memory corruption has propagated during execution—may be difficult to locate manually from the point of the program failure. Our approach iteratively isolates the root cause through multiple program executions. The approach modifies each execution in a particular way to reveal more of the memory corruption, until finally the first point of the memory corruption (the root cause) is revealed. The key idea is that on each execution, the approach ensures that the original failure is *avoided* and the program can continue executing. The modified execution may or may not result in a new program failure. The approach iterates until either a concurrency bug is found to be the root cause of a failure, or no new program failure is observed.

Each iteration of our approach specifically involves the following. First, the approach determines whether a data race, atomicity violation, or order violation concurrency bug is directly involved in the failure. If so, the approach reports this concurrency bug and terminates. If not, then the memory corruption that directly caused the failure is *suppressed* during the execution. Suppression means that the direct and indirect effects of the program statement that caused the memory corruption are nullified during execution. Intuitively, the result is as if all of the program statements that depend upon the memory corruption in question are simply *not executed*. Since the failure in this case will depend upon the suppressed memory corruption, the failure itself will be avoided during execution, and execution will continue for those statements that do not depend upon any of the known corrupted memory (that is being suppressed). If another failure occurs during execution, then this reveals additional, previously hidden memory corruption. The approach then iterates again to analyze this newly revealed memory corruption. If no new failure occurs in the execution, then the approach assumes that the last program point at which suppression was carried out is the root cause. Thus, our approach iteratively reveals more of the memory corruption in an execution until the first point of corruption (the root cause) is revealed. The approach assumes that if memory corruption exists in an execution, then the execution will produce a memory-related failure.

Our overall approach is composed of three main tasks: (1) ensuring that a multithreaded bug can be faithfully reproduced on multiple program executions; (2) determining on-the-fly whether a particular statement instance during execution is involved in a concurrency bug; and (3) performing execution suppression in order to account for propagation of memory corruption during execution. Our high-level approach is presented in Figure 1. The approach takes as input a single-threaded or multithreaded program and an associated test case execution that causes a failure in the program. The output of the approach is either an identified concurrency bug or program statement(s) that is likely to be the root cause of the failure.

Task 1: Reproducing failures caused by multithreading bugs. When debugging any program, it is necessary to be able to reproduce a program failure so that a failing execution can be analyzed to deduce what is going wrong and to figure out how to fix the problem. For single-threaded (deterministic) programs, it is enough to simply re-execute the program using the same set of input values. However, for multithreaded programs, being able to trigger and repeat a failure is generally not a trivial task. This is because certain multithreading bugs may only manifest themselves under particular thread interleavings, and these interleavings may change between executions even when those executions are based on the same input values. Since our approach involves multiple program executions, it is required that each execution be a faithful reproduction of the original failing execution (except some portions of execution that are necessarily modified by our approach).

Much research work has been done to identify failure-triggering thread interleavings for in-house testing [14, 15] or for reproducing concurrency bugs that occur at the end-user side [16–19]. These approaches either provide a bug-triggering control to a runtime system, such as *Valgrind* or *Pin*, or they generate a whole program execution log, to ensure that a failure can be repeatedly reproduced for debugging. Based on this previous work, our approach provides three ways for programmers to specify a failing execution that can be faithfully repeated on multiple executions. We rely on the *Valgrind* infrastructure [8] for this purpose. *Valgrind* is a user-space dynamic binary translation system that provides a synthetic CPU in software for program execution, and includes its own thread scheduling mechanism that we modified to record a thread interleaving and to force a particular thread interleaving during execution. The first way a programmer can specify a failure-triggering thread interleaving to our approach is to provide a whole program execution trace for the failure. In this case, there is no need to record thread interleavings and subsequent executions are based on the original trace file and use the same thread interleaving that originally triggered the bug. Second, a bug-triggering control produced by other tools can be provided. In this case, *Valgrind* constrains the choices of the thread scheduler at the relevant control points to ensure that the failure occurs, while recording the whole execution trace for the failure. The recorded trace is then used to guide subsequent executions. Finally, since tools to provide such bug-triggering controls were not available to us, our experimental study made use of a third approach: we inserted `sleep` function calls in each program to cause the bug to be triggered, then used *Valgrind* to record the failure-triggering thread interleaving in the first run. This recorded interleaving was then used to guide all subsequent program executions in which the failure needed to be repeated. It is important to note that in general, merely recording a failure-triggering thread interleaving may not be sufficient to reproduce a failure. Other environmental factors may need to be recorded as well,

```
input:
    Single-threaded or multithreaded program $P$, and
    test case execution $t$ causing a memory-related program failure.
output:
    Concurrency bug or statement(s) identified as the root cause of the failure.
algorithm SearchForMemoryFailureRootCause
begin
    // Task 1: Ensure failure can be reproduced during multiple program executions.
1:  analyze execution $t$ on $P$ to ensure that the failure can be faithfully reproduced;
2:  $S_{def} := \{\}$;
3:  $S_{supp} := \{\}$;
4:  while an observable memory failure $f$ occurs during execution $t$ on $P$ do
5:     $S_{use}$ := the statement instance at which $f$ occurs;
6:     $L$ := the used memory location(s) causing $f$ at $S_{use}$;
7:     $S_{def}$ := the corresponding statement instance(s) last defining location(s) in $L$;
       // Task 2: On-the-fly check for concurrency bug directly causing $f$.
8:     re-execute $t$ on $P$ until we reach a statement instance $s \in S_{def}$
          (while monitoring accesses to corresponding memory location $l \in L$);
9:     if $\exists$ potentially-harmful WAR or WAW data race $d$ at $s$ then return $d$;
10:    if $\exists$ $W_1R_2W_1$ or $R_1W_2W_1$ atomicity/order violation $v$ at $s$ then return $v$;
       // Tasks 2 and 3: Perform execution suppression while also continuing to check
       // on-the-fly for concurrency bug directly causing $f$.
11:    resume execution from $s$ while performing the following:
       (1) checking if execution reaches a new $s \in S_{def}$ defining associated $l \in L$:
          if execution reaches a new $s$ defining $l$ then goto line 9 above;
       (2) monitoring accesses to $l$:
          if $\exists$ potentially-harmful RAW data race $d$ then return $d$;
          if $\exists$ $R_1W_2R_1$ or $W_1W_2R_1$ atomicity/order violation $v$ then return $v$;
       (3) suppressing the following statement instances:
          (a) the definition of location $l$ at $s$;
          (b) statement instances in $S_{supp}$; and
          (c) any subsequent statement instances directly or indirectly influenced
             by the definition of $l$ at $s$;
12:    augment set $S_{supp}$ with the additional statements suppressed at line 11 above;
    endwhile
13: return the statement(s) associated with the latest $S_{def}$;
end SearchForMemoryFailureRootCause
```

Figure 1. General approach for isolating a multithreading (or single-threaded) memory bug.

such as data read from an input file, or a value defined as the current system time. However, this was not required by the programs in our experimental study.

In Figure 1, the task of ensuring multithreaded execution repeatability is performed at line 1, before the main iterative loop begins. The inputted failure-triggering test case execution is analyzed to record the information necessary to ensure faithful reproduction of the failure on subsequent executions. This information is then used on each iteration of the approach to ensure that the original execution is faithfully repeated.

The main loop of our approach is shown in lines 4–12 in Figure 1. As long as the program execution ends in a memory failure (loop condition at line 4), our approach analyzes the executed statement instance that is *directly involved* in the failure. Identifying this statement instance is straightforward because the point of the failure is known, and therefore the accessed memory location(s) causing the failure can also be easily found (lines 5–7). In most cases, there will likely be a single used memory location directly causing the failure. However, in the case of an array access directly causing the failure, this can be caused by either the location of the array base address, or the location of the index into the array. Thus, lines 6 and 7 in Figure 1 operate on sets to handle such cases. As shown in line 7, our approach targets the last definition(s) (write access(es)) that directly led to the failure. This statement instance(s) may or may not be involved in the actual root cause of the failure, and so our approach attempts to verify this by re-executing the program (line 8). During re-execution, the next main task of our approach is to determine whether the last definition(s) in question is involved in a concurrency bug that directly caused the observed program failure.

Task 2: On-the-fly checking for a concurrency bug directly associated with the program failure. During program re-execution in our approach (lines 8–11 in Figure 1), memory location accesses are monitored so that on-the-fly checks can be performed to identify concurrency bugs that may have caused the program failure. These on-the-fly checks handle three particular kinds of common concurrency bugs: data races, atomicity violations, and order violations. All of these concurrency bug checks are made during each program re-execution, but we present them here as separate algorithms for ease of understanding. We first discuss checks made for data race bugs, and then later we discuss checks made for atomicity and order violation bugs.

Data race bugs. A *data race* can be defined as the concurrent access of a shared memory location by two or more different threads, such that the following two conditions hold: (1) at least one of those accesses involves a write to that memory location and (2) there is no synchronization specified between those memory accesses. Intuitively, a data race can lead to such problems as a value being unexpectedly overwritten or a stale value being read.

Given a target statement instance to analyze that performs a write, our approach determines whether a data race exists at this statement instance as follows. At this statement instance, it is known which thread executed the write. Thus, when re-executing the program, the approach monitors on-the-fly any access to this memory location that comes from a different thread. In particular, the approach finds the last access (read or write) to this memory location that comes from a different thread, and checks for any synchronization that may exist between these two accesses (Section 3 gives more details about how we checked for synchronization in our experiments). If no synchronization exists, then this implies that there is either a WAR or a WAW dependence between these two memory accesses that represents a data race. If there is no WAR or WAW data race here, then the execution continues from this point while the approach monitors for any read access to the same location coming from a different thread. Again, if there is no synchronization between two such accesses, this indicates a RAW data race.

Figure 2 shows the pseudocode for this on-the-fly approach for detecting data races. This accounts for the observation by Tallam *et al.* [11] that data races can be represented by either RAW, WAR, or WAW dependencies. Note that in Figure 1, the checking for RAW data races is done on-the-fly while also performing the suppression.

Our approach detects whether a data race exists and occurred during program re-execution at the crucial point at which a failure-triggering memory location is last defined. However, such a data race may not actually be *potentially harmful* [13]. For example, data races that do not affect the state of the executing program are not harmful. Therefore, once a data race is detected by our approach, it is reported only if it is determined to be potentially harmful, because only then can it possibly be the root cause for the failure (lines 7 and 17 in Figure 2).

Our approach determines whether a data race is potentially harmful by altering the sequence of executing threads. Suppose that the two memory accesses involved in a data race are at points $access_1$ and $access_2$ during execution (in that order), and these accesses are performed by threads t_1 and t_2, respectively. Also assume that at points $access_1$ and $access_2$, the values read from or written to the associated memory locations are, respectively, v_1 and v_2. The key idea is to re-execute the program and force the two memory accesses to occur in reverse order, so that $access_2$ in t_2 occurs *before* $access_1$ in t_1 during execution. This requires control over the thread scheduler, which is described in the next section. At the point of the second memory access during re-execution, the approach checks whether either of the read/written values v_1 or v_2 are different (have been changed). If so, the data race is determined to be potentially harmful. If not, then the race is determined to *not* be potentially harmful.

```
input:
    Execution $E$ with target write instruction instance $w$, writing to location $l$ and
      executed by thread $t$.
output:
    An identified data race bug, or NULL if no data race bug is found.
algorithm SearchForDataRaceBug
begin
1:  $foundSync := false$;
2:  $lastAccess :=$ "undefined";
    // Monitor for WAR and WAW data race bugs.
3:  for each statement instance $i$ in execution $E$ taken in order do
4:    if $i == w$ then
5:      if $lastAccess$ != "undefined" && !$foundSync$ then
6.        $d :=$ the data race between $lastAccess$ and $w$;
7:        if $d$ is potentially harmful then return $d$;
8:      break;
9:    else if $i$ accesses location $l$ using thread other than $t$ then
10:     $lastAccess := i$;
11:     $foundSync := false$;
12:   else if $i$ performs thread synchronization related to $t$'s access of $l$ then
13:     $foundSync := true$;
      endif
    endfor
    // Monitor for RAW data race bugs (if no data race bugs found above).
14: for each remaining statement instance $i$ in execution $E$ taken in order do
15:   if $i$ reads from location $l$ using thread other than $t$ then
        // If we reach here, there is no synchronization, otherwise the program
        // would have returned below at line 19.
16.     $d :=$ the data race between $w$ and $i$;
17:     if $d$ is potentially harmful then return $d$;
18:   else if $i$ performs thread synchronization related to $t$'s access of $l$ then
        // There cannot be a RAW data race.
19:     return NULL;
      endif
    endfor
20: return NULL;
end SearchForDataRaceBug
```

Figure 2. On-the-fly checking for data race bugs.

As an example of a harmful data race, suppose there is a read in thread t_1 and a subsequent write in thread t_2 to a particular memory location, and there is no synchronization specified between these two memory accesses. When the program is re-executed, thread t_2's memory access can be forcefully scheduled before thread t_1's memory access, causing the write to occur prior to the read. At the point of the second memory access (which is now the read), it would likely be determined that a different value will have been read from the memory location, as compared to what was read from that memory location in the original execution. As a result, the data race would be identified as potentially harmful.

Atomicity and order violation bugs. Data races occur when two memory accesses to the same location (with at least one write) are not synchronized, such that there is no strict happens-before relationship between the two memory accesses. However, in cases where there may indeed be synchronization between two memory accesses, these two accesses may still not be *atomic* when they are expected to be. In these cases, it is possible that an interleaving thread may access the same memory location in-between the other two memory accesses that were erroneously assumed to be atomic. This can put the program execution into an unexpected state and is known as an *atomicity violation* [12, 20]. Atomicity, also referred to as serializability, means that the data manipulation effect of multiple concurrently executed actions is equal to a serial version of those actions [12].

Such thread interleavings are called *serializable interleavings*. When the programmer's atomic assumption is broken by an unserializable interleaving, we say that atomicity is violated, and an atomicity violation bug manifests itself. For example, suppose thread t_1 reads from a particular memory location at two different statements during execution, but this thread does not perform any writes to this location in-between both reads. In this case, one might think that it is safe to assume the same value would be read at both statements. However, if these two memory accesses are not specified as being atomic, then an interleaving thread t_2 may be unexpectedly scheduled in-between the two reads, which may then write a value to that memory location. In this case, the assumed atomicity of the two reads by thread t_1 has been violated, and we can refer to this situation as an $R_1W_2R_1$ atomicity violation.

For an atomicity violation bug, it is assumed that the bug will be prevented if the atomicity is enforced. In the previous example with the $R_1W_2R_1$ atomicity violation, it is assumed that both of the following memory access sequences will prevent the bug, because they preserve the atomicity of the two memory accesses that were assumed to be atomic: $W_2R_1R_1$ and $R_1R_1W_2$. However, in certain situations it may turn out that only one of these alternate interleavings will prevent the bug. This would imply that even though these memory accesses may be synchronized, the problem is not due to atomicity being violated, but instead, the problem is due simply to the wrong order of memory accesses. In these cases, we refer to such a bug as an *order violation*. This occurs when a programmer assumes an order between two (groups of) operations from two different threads, but the programmer fails to enforce such an order in the implementation. Eventually, an order violation bug happens when one of the two (groups of) operations happens before (or after) the other (group of) operation, inverting the programmer's assumption [21]. Our approach handles the possibility of atomicity violations (involving three memory accesses) and order violations (involving two or three memory accesses) as being the root cause of multithreaded program failures. Note that, like in other previous work, our approach fails to report atomicity violation bugs involving more than three memory accesses to the same variable, or memory accesses to two or more different variables. The same limitations apply to the order violation bugs. However, with these limitations we were still able to detect all the atomicity and order violation bugs in our experiments.

According to work done by Lu *et al.* [12], given three accesses to a memory location in which the middle access is due to an interleaved thread, such that each access is either a read or a write, there are eight possible distinct memory access sequences. However, the authors show that only four of these sequences can lead to atomicity violations, because the other four sequences are actually equivalent to serial accesses in which there is no interleaved thread. The four memory access sequences that can lead to atomicity violations are the following: $W_1R_2W_1$, $R_1W_2W_1$, $R_1W_2R_1$, and $W_1W_2R_1$. Thus, our approach checks for only these particular memory access sequences when looking for possible atomicity violations. Given a target statement instance to analyze that performs a write, this write can only occur as the *final* write in each of the above four sequences. This is because the target write instruction is always guaranteed to be the *last* definition of the memory location prior to the point of the program failure; there can be no later writes to the same memory location that could have been responsible for the failure. As a result, when looking for a potential atomicity violation, our approach executes a program up to the target write instruction instance while simultaneously keeping track of the last memory access to that location by the same thread, as well as the last memory access to that location by a different thread. Once execution reaches the target instruction instance, then a check is performed to see whether the $W_1R_2W_1$ case or the $R_1W_2W_1$ case occurred. If either case occurs, this is a potential atomicity violation. If neither case occurs, then execution proceeds while monitoring for subsequent reads from the memory location in question coming from a different thread. If any such read occurs, then a check is performed for either the $R_1W_2R_1$ case or the $W_1W_2R_1$ case. Either of these cases could suggest a potential atomicity violation.

In general, these sequences of three memory accesses in which the middle access is interleaved could represent an atomicity violation, order violation, or no violation at all. However, order violations may also involve only two memory accesses, similar to the case of data races. The difference between an order violation and a data race is that two memory accesses in a data race

```
input:
    Execution $E$ with target write instruction instance $w$, writing to location $l$ and
      executed by thread $t$.
output:
    An identified atomicity/order violation bug, or NULL if no such bug is found.
algorithm SearchForAtomicityViolationOrOrderViolationBug
begin
1:  $lastAccessSameThread$ := $lastAccessDiffThread$ := "undefined";
    // Monitor for $W_1R_2W_1$ and $R_1W_2W_1$ atomicity/order violation bugs.
2:  for each statement instance $i$ in execution $E$ taken in order do
3:    if $i == w$ then
4:      if $lastAccessSameThread$ and $lastAccessDiffThread$ are defined &&
        $lastAccessSameThread$ occurs before $lastAccessDiffThread$ in $E$ &&
        exactly one last access is a read while the other is a write then
5.        $v$ := the potential atomicity/order violation between $lastAccessSameThread$,
            $lastAccessDiffThread$, and $w$;
6:        if $v$ is found to be a true atomicity or order violation then return $v$;
7:      break;
8:    else if $i$ accesses $l$ using thread $t$ then $lastAccessSameThread$ := $i$;
9:    else if $i$ accesses $l$ using thread other than $t$ then $lastAccessDiffThread$ := $i$;
    endif
  endfor
  // Monitor for $R_1W_2R_1$ or $W_1W_2R_1$ atomicity/order violation bugs
  // (if no atomicity/order violation bugs found above).
10: for each remaining statement instance $i$ in execution $E$ taken in order do
11:   if $i$ reads from location $l$ using thread other than $t$ then
12:     $nextReadDiffThread$ := $i$;
13:     if $lastAccessDiffThread$ is defined && it involves same thread as $i$ then
14.       $v$ := potential atomicity/order violation betw. $lastAccessDiffThread$, $w$, and $i$;
15:       if $v$ is found to be a true atomicity or order violation then return $v$;
16:   else if $i$ is the last instance in $E$ && $lastAccessDiffThread$ is defined then
17:     $v$ := the potential order violation between $lastAccessDiffThread$ and $w$;
18:     if $v$ is found to be a true order violation then return $v$;
19:   else if $i$ is the last instance in $E$ && $nextReadDiffThread$ is defined then
20:     $v$ := the potential order violation between $w$ and $nextReadDiffThread$;
21:     if $v$ is found to be a true order violation then return $v$;
    endif
  endfor
22: return NULL;
end SearchForAtomicityViolationOrOrderViolationBug
```

Figure 3. On-the-fly checking for atomicity violations and order violations.

are not synchronized, whereas the two accesses in an order violation are synchronized (just in the wrong order). If our algorithm to handle order violations only considers situations involving three memory accesses, we may miss some order violation bugs. As a result, in our algorithm to detect atomicity and order violations on-the-fly, we include a special check at the end of execution for the case of only two memory accesses being involved in a potential order violation (assuming no other atomicity/order violations involving sequences of three memory accesses were found during execution).

Figure 3 shows the pseudocode for this approach to on-the-fly checking for potential atomicity and order violations. Our approach checks whether a potential violation is a true violation before reporting it (lines 6, 15, 18, and 21 in Figure 3).

To determine whether a potential violation is a true violation, for the case of three involved memory accesses, our approach alters the sequence of executing threads by *moving the interleaving thread* that occurs between the first and third memory accesses. This is done in two ways. First, the interleaving thread is performed *before* the other two memory accesses. Second, the interleaving thread is performed *after* the other two memory accesses. This requires two separate program executions to force each of the two alternate thread interleavings. The effect of these alternate thread interleavings is then examined in terms of whether or not the program failure is now avoided. If *both* alternate thread interleavings cause the failure to be avoided, then an atomicity violation is reported because this suggests that the interleaving thread interrupted the assumed atomicity of the other two memory accesses. If *only one* alternate thread interleaving causes the failure to be avoided, then an order violation is reported because this suggests that the order of the memory accesses may simply be incorrect. Finally, if *neither* alternate thread interleaving causes the failure to be avoided, then no violation is reported. In the case of only two involved memory accesses representing a potential order violation, then one additional program execution is performed to force the alternate memory access order; if the program failure is then avoided, an order violation is reported.

Task 3: Performing execution suppression. The final task performed by our approach is to carry out execution suppression. This occurs in the event that the write statement instance(s) directly causing a failure is not found to be associated with any data race, atomicity violation, or order violation concurrency bug. Suppression is performed simultaneously with monitoring memory accesses during execution for on-the-fly checking of some concurrency bugs (line 11 in Figure 1).

Suppression is performed because the approach does not initially know whether the memory corruption directly causing a failure is associated with the root cause, or whether this memory corruption is due simply to propagation. Suppression allows the approach to determine which case is more likely. To perform the suppression, the approach continues execution from the corrupted definition (from the point at which the algorithm had left off in line 8 in Figure 1). During this execution, the approach *suppresses* the effect of the corrupted definition statement instance by simply not executing that statement. Additionally, any other statement instances that directly or indirectly depend upon the first suppressed statement instance are themselves suppressed. Finally, any other statement instances that were suppressed during prior iterations of the approach are suppressed as well. By performing suppression in this way, the failure observed in the execution is guaranteed to be avoided, because the failure depends upon the suppressed statement instances. If no additional failures occur during the execution, then the most recently suppressed statement instance(s) is likely to be associated with the root cause, and is reported (line 13 in Figure 1). This is because the absence of a failure indicates that there is likely no more memory corruption present in the execution. On the other hand, if another failure occurs, this implies that memory corruption remains in the execution and so the root cause has still not yet been revealed. In this case, the approach iterates again (back to line 4). Note that while suppression is being carried out, any necessary checks for concurrency bugs at the point of a memory access are performed *before* the associated statement instance is potentially suppressed.

Comments about our approach. Our approach is designed to effectively locate the root causes of memory-related program failures for single-threaded as well as multithreaded programs. The approach accounts for three important types of concurrency bugs: data races, atomicity violations, and order violations.

An important requirement of our approach is that it assumes a failure-triggering thread interleaving is specified as the input. The current work does not focus on how to initially identify such an interleaving, but existing techniques can be used for this purpose. For example, Sen *et al.* have done work [22–24] on random testing for data race and atomicity violation detection, which involves choosing thread schedules at random. This technique can be used to automatically search for failure-triggering thread interleavings. Related work was done by Edelstein *et al.* [25], in which a Java program is randomly seeded with `sleep`, `yield`, and `priority` primitives at particular program points, and during runtime, decisions are made as to whether or not these primitives are executed. Similarly, the work by Lu *et al.* [26] proposes a hierarchy of concurrent program thread interleaving coverage criteria that can be used to systematically explore the interleaving space and effectively expose concurrency bugs.

Another important requirement of our approach is that a memory-related failure must manifest itself in such a way that it directly reveals a corrupt memory location. This can occur when a program crashes at the point at which a corrupt memory location is accessed. Our work focuses on these kinds of failures because they allow us to easily and automatically identify a corrupt memory location at the point of a failure. For other kinds of failures, such as incorrect program output, assertion violations, or raised exceptions, our approach can apply if there is a way to automatically link the failure to a corrupt memory location. In general, however, this may be difficult or even impossible for certain instances of these kinds of failures, because they may not involve memory corruption or they may not be readily linked to a corrupt memory location. Therefore, we cannot claim that our approach applies to these other kinds of failures in general. Also, since our approach only works for memory-related program failures that are caused by memory corruption, our approach cannot be used to identify memory leak errors. A memory leak is a type of memory error in which unnecessary allocated memory is never freed; this type of error does not involve corruption of memory during execution.

Our approach for checking whether a data race is potentially harmful, or whether a potential atomicity or order violation is a true violation, involves re-executing the program and altering the sequence of executing threads to change the relative order of the involved memory accesses. This approach—while relatively simple and working effectively in our experiments—can lead to both false positives and false negatives. For example, an identified data race involved at the point of a known corrupt memory location might indeed be potentially harmful, but may not necessarily be the root cause of a failure (though it may be something worthwhile for a developer to check in case it can lead to other failures). Also, forcefully altering the sequence of executing threads may sometimes put the execution in an unexpected state, such as when a thread needs to be forcefully scheduled when it may not normally have been executable. This can make it appear that a concurrency bug is harmful when it may not actually be. Moreover, in the case of concurrency bugs that involve more complicated thread interleavings than those considered, our approach may miss these concurrency bugs. Using more accurate concurrency bug detection techniques would likely require more runtime cost, but we chose the current approach for its simplicity, and because it worked effectively for our experimental subjects.

Since our approach may sometimes identify more than one used memory location that could be directly responsible for a program failure, our approach may output more than one statement as the likely root cause of the failure. Although we do not expect that this situation will be very common in practice, it should be noted that when it does occur, then a user may have to examine more than one outputted statement to identify the true root cause.

Finally, it is possible that our approach may terminate prematurely in cases where memory corruption exists in an execution but does not cause the program to exhibit a failure. In cases such as this, our approach will identify a statement that is likely to be closer to the root cause than the point of the original failure, but it may not be the root cause itself.

2.2. Illustrative example

Example part 1: failure is caused by a data race bug. Consider the example multithreaded code snippet shown in Figure 4. In this piece of code, x and y are pointers to shared memory locations, each containing an integer. Lines 1–2 are erroneously not protected in a critical section, despite the fact that they involve a read and subsequent write to shared location x. A potentially harmful data race, therefore, exists at this point. Lines 3–4, which involve a write and subsequent read to shared location y, are protected by a critical section. However, the write into location y at line 3 may write the wrong value, due to the data race in lines 1 and 2 involving location x. Since $*y$ is then used as an index into an array in line 4, there is potential for a crash at this point. Similarly, there is potential for a crash at line 5, which uses $*x$ as an array index. In this example, we will assume that these potential crashes will in fact occur during execution.

If the example code is executed using two parallel threads, the execution may end in a failure (crash) as depicted in Figure 5(A). In this execution, thread 1 executes line 1 first, then this is immediately followed by thread 2 executing line 1 (this is possible since lines 1–2 are not protected by synchronization). In this case, thread 2 reads a stale (incorrect) value from location x, since it was expected that this read would not occur until after thread 1 updates x in line 2. Next, thread

```
Let x and y be pointers to shared memory locations.
Let foo, foo2 be functions that take an integer input and return an integer.
Let structArray be a heap array of pointers to structs with a field named data.

// Assume *x has been initialized to some value.
// x not protected by critical section in lines 1–2.
1:  int a = foo(*x);
2:  *x = *x + a;
    start_critical_section();
       // Defined value of y may be incorrect.
3:     *y = foo2(*x);
4:     int b = structArray[*y]−>data;
    end_critical_section();
5:  int c = structArray[*x]−>data;
```

Figure 4. Example multithreaded code snippet with a data race bug.

2 writes to *x* in line 2; this writes an unexpected value to *x* since this value is computed using the stale value for *x* read from the same thread in line 1. Now, thread 1 resumes and writes to *x* in line 2; again, this writes an incorrect value to *x* since it is computed using an unexpected value for *x* obtained from the write in thread 2. If thread 1 continues executing, line 3 reads the wrong value for *x* and then defines an incorrect value for *y*, which is then used in line 4. At line 4, execution crashes due to a buffer overflow.

Since the array index **y* is directly related to the crash at line 4, our approach determines that the last definition of this location occurred at line 3. The next iteration of our approach is shown in Figure 5(B). During this execution, it turns out that there is no data race involved at line 3. This is because there are no prior accesses to location *y* before line 3 (so it is not involved in any WAR or WAW dependence), and there is no subsequent read from location *y* after line 3 from another thread that is not protected by synchronization. Similarly, there are no potential atomicity or order violations involved here. Thus, the definition of *y* at line 3 is suppressed, and the definition of *b* at line 4 is also suppressed since it uses the suppressed definition at line 3. Thus, the original crash is avoided. Now, execution continues as thread 2 resumes and executes lines 3 and 4 (these lines are not suppressed in this execution since they do not depend on the previously suppressed instances of lines 3 and 4 from the other thread). A crash occurs at this new instance of line 4 in thread 2. Again, the last definition of *y* is at line 3. During the next execution, shown in Figure 5(C), there is again no concurrency bug detected at this point. Lines 3 and 4 are then suppressed in the execution of thread 2. Now, execution is able to reach line 5 in thread 2. A crash occurs here due to the use of incorrect **x* as an array index.

Our approach determines that the last definition to location *x* occurred at line 2 of thread 1. The next execution is shown in Figure 5(D). When control reaches line 2 of thread 1, a data race check is first performed. In this case, it is discovered that the last access to location *x* by another thread is the write in line 2 of thread 2. Since there is no synchronization specified between these two memory accesses, our approach identifies a data race at this point (WAW dependence). Next, the approach checks whether this data race is potentially harmful. In the original execution, thread 2, line 2 is executed *before* thread 1, line 2. Thus, our approach re-executes the program and forces thread 1, line 2 to be executed before thread 2, line 2 (the reverse order). This effectively alters the interleaving of these two threads so that the behavior matches what is expected. As a result, the value written into location *x* at the second access is now changed, and the data race is determined to be potentially harmful. This data race, which is associated with the true root cause of the failure, is then reported.

Example part 2: failure is caused by a non-concurrency bug. Suppose we slightly modify the example code from Figure 4 to obtain the example code in Figure 6. This example is identical to the previous example, except that in this case, we have added lines -1 and 0 to the beginning of

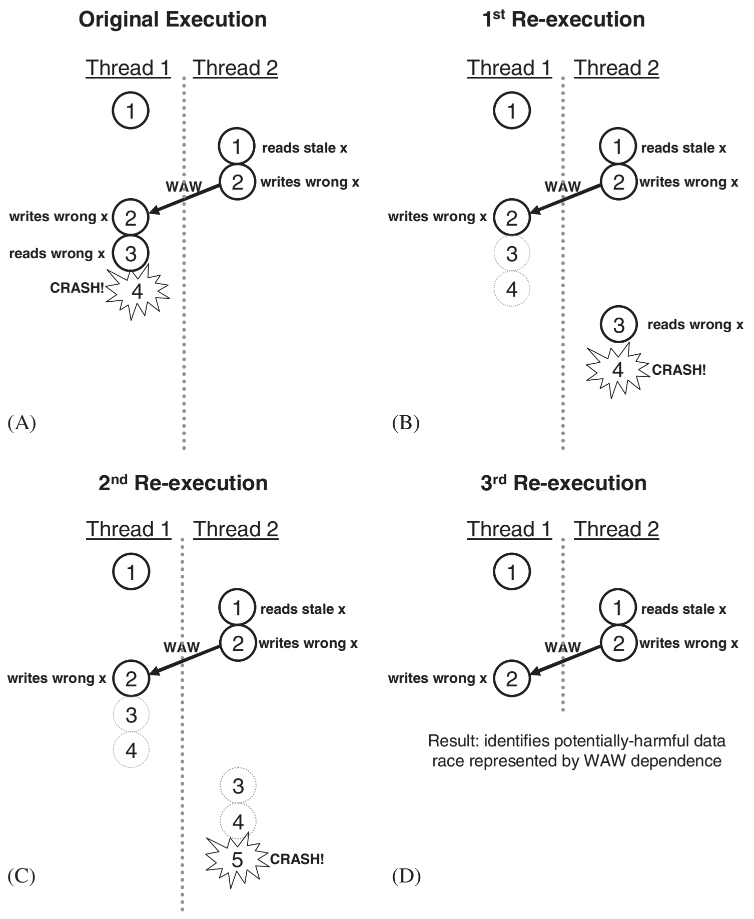

Figure 5. Example executions for running approach with a data race bug, to accompany Figure 4.

the code snippet, and line 6 to the end of the code snippet. Line −1 defines a shared variable z to be an incorrect constant value (this is the root cause of the failure), which in turn is used in line 0 to initialize x to an incorrect (also constant) value. Line 6 involves an assertion that ensures z contains the correct value; if not, then the program terminates with an error at this point. Note that the same data race exists here as in the previous example. However, we assume in this case that for the constant value used to define location x in line 0, the value returned by the call to foo in line 1 is actually 0. Then in this situation, the data race left over from the previous example is *not* potentially harmful. This is because the value written into location x at line 2 does not change the constant value $*x$, and this is true for all threads, regardless of the thread interleaving.

Now consider running our approach on this modified example. Suppose that the original execution and the first two re-executions are the same as how they appear in Figures 5(A)–(C) (with the only minor difference being that lines −1 and 0 now precede execution of line 1 in each of the two threads). The crashes at lines 4 and 5 from the previous example can still occur in the current example because location x still contains an incorrect value in the current example.

The approach eventually gets to the end of the second re-execution as shown in Figure 5(C). Then the crash at thread 2, line 5 directly depends upon the store to location x in thread 1, line 2. The next execution, shown in Figure 7(A), is where the behavior of the approach in this example differs from the previous example. In this execution, control reaches thread 1, line 2. The same data race as was found in the previous example is also found here. However, in this case the data race is found to *not* be potentially harmful, as was explained earlier. Similarly, another data race is found here, which is a RAW race involving the write in thread 1, line 2, and the subsequent read in

```
Let x, y, and z be pointers to shared memory locations.
Let foo, foo2 be functions that take an integer input and return an integer.
Let structArray be a heap array of pointers to structs with a field named data.

// Assume z is defined to be an incorrect constant.
-1:  *z = some_incorrect_constant_value;
0:   *x = 2 * (*z);
// x not protected by critical section in lines 1–2.
// Assume a is defined to be 0 given constant *x.
1:   int a = foo(*x);
2:   *x = *x + a;
     start_critical_section();
        // Defined value of y may be incorrect.
3:      *y = foo2(*x);
4:      int b = structArray[*y]−>data;
     end_critical_section();
5:   int c = structArray[*x]−>data;
6:   assert_correct_value(*z);
```

Figure 6. Example multithreaded code snippet with a non-concurrency bug.

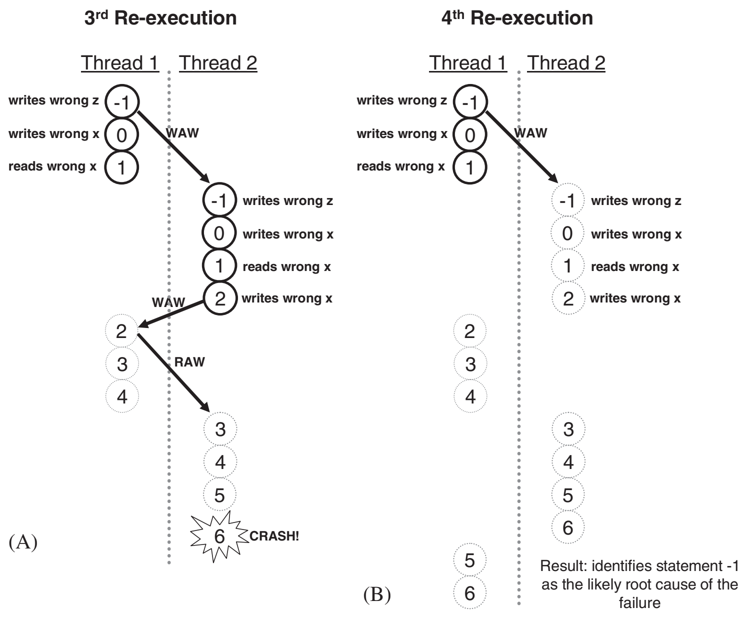

Figure 7. Example executions for running approach with non-concurrency bug, to accompany Figure 6. Assume that the original execution and the first two re-executions are similar to those in Figures 5(A)–(C).

thread 2, line 3. However, again this race is found to *not* be potentially harmful, since the write to location *x* in line 2 does not actually change its value. Thus, no potentially harmful data races are identified here. There are also no atomicity or order violations identified by the approach, so the approach continues by suppressing the definition of *x* at thread 1, line 2, and anything dependent upon it. This allows execution of thread 2 to arrive at line 6, which crashes because the sanity check for the value in location *z* fails (since *z* contains an incorrect value). The last definition of location *z* occurs at thread 2, line −1.

In the next execution, shown in Figure 7(B), control arrives at thread 2, line −1. In this case, there exists a previous access to location *z* from the other thread (thread 1, line −1), so there is a data race involved here represented by a WAW dependence. However, since *z* is initialized to a constant

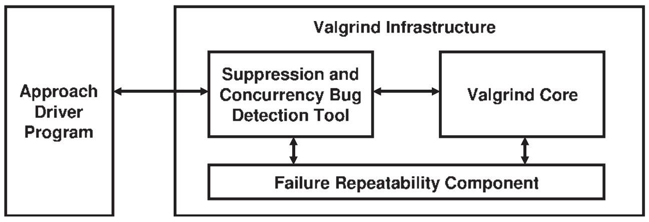
Approach Driver Program
Valgrind Infrastructure
Suppression and Concurrency Bug Detection Tool
Valgrind Core
Failure Repeatability Component

Figure 8. High-level design for the implementation of our approach. Arrows represent interactions between components.

value, again this data race is not potentially harmful. There are no additional subsequent reads of z prior to the point of the crash, so there are no RAW data races at this point. Similarly, there are no atomicity or order violations associated with location z. Thus, the approach continues by suppressing the definition of z in thread 2, line -1, and all of the subsequent statement instances that are dependent upon it (including all of the statement instances suppressed in previous iterations). In this example, all remaining executed statements in both threads are suppressed, resulting in no more program failures. The approach then terminates and identifies statement -1, the statement associated with the most recent point of suppression, as the root cause of the failure.

3. IMPLEMENTATION

The high-level design of our approach is shown in Figure 8. Our implementation is primarily written in C within the Valgrind infrastructure [8]. Valgrind is a user-space dynamic binary translation framework that provides a synthetic CPU in software for program execution. Valgrind allows a program to be dynamically instrumented and modified at runtime. Since Valgrind allows for modifying an execution at the binary instruction level, our implementation also works at this level. We map the results of our approach back to the associated program statements (in the source code) prior to reporting them to the user. Also, we assume that the multithreaded programs under consideration are run on a single-processor system, so that only one thread is running at any given time.

The Valgrind infrastructure we used is composed of three components: (1) the *Valgrind Core*, which is the core functionality already provided by Valgrind; (2) the *Failure Repeatability Component*, which is a component we implemented within Valgrind that ensures a failure in a multithreaded execution can be repeated on subsequent executions; and (3) the *Suppression and Concurrency Bug Detection Tool*, a tool we implemented to carry out a program execution while performing the tasks necessary to conduct execution suppression as well as on-the-fly concurrency bug detection. Finally, a fourth component implemented outside of Valgrind called the *Approach Driver Program* works in conjunction with Valgrind to actually run our approach. We now describe some implementation details for each of these components.

The Valgrind Core. Because Valgrind provides a synthetic CPU in software for program execution, it includes its own thread scheduler. We customized the thread scheduler in the Valgrind core to work in conjunction with the *Failure Repeatability Component* to ensure that the order in which threads are scheduled in a failing execution is repeated on subsequent executions. In our experiments, this ensured that the same failure could be repeated on multiple program executions. This is because we preserved the sequence of scheduled threads on each execution, and for our experimental subjects, no other environmental information needed to be recorded in order to ensure that a failure could be repeated. The *Valgrind Core* also works with the *Suppression and Concurrency Bug Detection Tool* to ensure that particular thread interleavings can be forced as necessary to check for harmful concurrency bugs.

The Failure Repeatability Component. This component works in conjunction with the *Suppression and Concurrency Bug Detection Tool* and the *Valgrind Core* to ensure that a failure can be repeated on multiple program executions. This component works as follows. First, it is assumed that a failing execution (with corresponding thread interleaving) is inputted to our approach. In our experiments, this was actually done by adding `sleep` function calls to the program execution as necessary to ease repeats of the failures. When the *Suppression and Concurrency Bug Detection Tool* carries out a failing execution for the first time, the *Failure Repeatability Component* records the order in which threads are scheduled during the execution. Then, for all subsequent executions conducted by our approach to isolate the same bug, this component communicates with the *Valgrind Core* to guarantee that the threads are scheduled in the same order as in the original execution. This is accomplished by forcing the thread scheduler in Valgrind to schedule threads according to our own specified schedule.

To ensure a failure could be repeated in our experiments, it was sufficient to record only the thread interleaving of the original failing execution. In general, however, other environmental information associated with the original failing execution may also need to be recorded.

The Suppression and Concurrency Bug Detection Tool. We implemented this tool within the Valgrind infrastructure to handle the two main tasks required for each program execution in our approach: (1) execution suppression and (2) on-the-fly concurrency bug detection. The tool takes as input a program with thread interleaving for a failure-triggering execution, a set of instruction instances whose effects should be suppressed during execution (called 'suppression points'), and a target write instruction instance (with associated memory location and value, and the ID of the thread performing the write) that must be checked to determine whether it involves a potentially harmful data race, atomicity violation, or order violation.

Execution suppression. To perform execution suppression, the *Suppression and Concurrency Bug Detection Tool* performs tracing in addition to performing the suppression itself. The tracing is required to see which memory locations are accessed and when, to determine the latest definition that directly causes a failure. The suppression is required to nullify the effects of the appropriate instructions during execution.

For tracing during a given execution, the tool records a trace of the memory locations accessed (loaded from and stored to) during the execution. To do this, the tool instruments each non-suppressed load and store instruction to record the current program counter, its associated instance number, the type of instruction (i.e. load or store), and the address of the accessed memory location. This information makes it possible, at the point of a failure, to identify the accessed memory location that directly caused the failure, and the corresponding instruction instance that last defined this memory location. The identified instruction instance can then be added to the list of suppression points for the next iteration of the approach, if necessary.

For suppression during a given execution, the *Suppression and Concurrency Bug Detection Tool* performs 'suppression information flow tracking' at all instructions, as well as 'actual suppression' at the appropriate load and store instructions. To do the suppression information flow tracking, the tool associates every memory location and register with a *shadow location* that contains information about whether or not the associated location needs to have its effects suppressed during the execution. Initially, all memory locations are marked as 'not suppressed'. Memory locations first become 'suppressed' when they are used in an instruction instance that is specified as a suppression point. At an instruction instance, if at least one of the used memory locations or registers is marked as 'suppressed', then any defined memory locations or registers are also marked as 'suppressed'. On the other hand, if none of the used locations are marked as suppressed, then any defined locations are marked as 'not suppressed'. Tracking this information during execution ensures that any instruction instance that directly or indirectly uses a suppressed location can have its effects suppressed as well.

Besides tracking suppression information, 'actual suppression' is performed at memory load and store instructions. At a store instruction instance that uses a suppressed location, the effect of the store is suppressed by *not* writing to the destination location. The effect is as if the store never occurred and the destination location retains whatever value was originally contained there. Similarly, for a load instruction instance that uses a suppressed location, the load is suppressed by *not* reading from the source location. The effect is as if the load never occurred and the destination register being loaded into retains whatever arbitrary data was originally contained there. Note that this arbitrary data will never be used since anything affected by it would be suppressed as well.

When suppressing, it is possible that a memory location marked as 'suppressed' can be used in a conditional check. In this case, our implementation suppresses the entire conditional structure associated with the conditional check. This is because all statements contained within the conditional structure are dependent upon the suppressed location. Special consideration is also made in the event that the return address of a function call is marked as 'suppressed'. In this case, our implementation relies on profiling information to force the function to return to a known, valid address. Finally, our approach refrains from making system calls when at least one of the input values for a system call is suppressed.

On-the-fly concurrency bug detection. Our implementation accounts for detection of data races, atomicity violations, and order violations, as being the possible root causes of memory-related program failures.

Our algorithm for on-the-fly checking for data races was previously shown in Figure 2. To implement this, the approach requires as input a target write instruction instance i, with information about the thread t that performs the write, whose memory location l is written, and the value v contained in the memory location after the write. During execution, all load and store (i.e. read and write) instruction instances executed prior to the target write instruction instance are instrumented as follows: if the accessed location is l and the thread performing the access is *different than* t, then the current instance is saved as the most recent access to l from a thread other than t, and a global flag *sync* is set to *false*, indicating that no synchronization on accesses to l has yet been found. Later, if another instruction instance is executed that accesses l by a thread other than t, then all prior access information will be discarded and only information about this most recent access will be recorded. If any synchronization point related to l is encountered during execution, then the flag *sync* is set to *true*. When execution eventually reaches the target instruction instance i, then the approach knows which instruction instance last accessed the associated memory location from another thread, and whether any relevant synchronization was encountered between the two memory accesses. As a result, the approach can easily determine at this point whether a WAR or WAW data race is involved. Next, the approach sets the *sync* flag to *false*, and resumes execution at the instruction instance immediately following i. Again, if any synchronization related to l is encountered during execution, then *sync* is set to *true*. In the event that a load (read) occurs from location l by a thread other than t, and the *sync* flag is *false*, then a RAW data race has been found.

When we instrument the executing program in our implementation, we assume that synchronization points are clearly marked in the program so that they can be easily identified. For our experiments, it was sufficient to automatically monitor for standard library function calls related to synchronization via locks (such as those provided by the `pthread` API). However, in general other kinds of synchronization may exist, including user-defined synchronization such as flag or barrier synchronization. Being able to accurately detect synchronization during execution can have a significant impact on the accuracy of the data race detection results. Thus, for other programs that may involve user-defined synchronization that is not explicitly marked in the code, existing techniques may need to be incorporated into our approach in order to automatically and accurately detect this synchronization. Tian *et al.* [27, 28] have recently described a dynamic technique to accomplish this.

Our algorithm for on-the-fly checking for atomicity and order violations is shown in Figure 3. The implementation here is similar to that of data race checking in terms of how the program is instrumented at runtime, with a few important differences. First, there is no need to check for synchronization points because atomicity/order violations may involve synchronized memory accesses (unlike data races in which memory accesses must not be synchronized). Next, since atomicity violations involve sequences of three memory accesses, then for a given target write instruction instance, the approach needs to monitor for *two* other accesses to that same memory location (not one other access like for the case of data races). Also, since order violations may involve sequences of two memory accesses, they may not be captured when sequences of three memory accesses are considered. As a result, once the end of execution is reached and no violations have yet been found involving sequences of three memory accesses, then a special check is performed for possible order violations involving only two memory accesses.

In our implementation, if any data race is detected, then it is checked to determine if it is *potentially harmful* before being reported as the root cause for a failure. Similarly, if any potential atomicity or order violation is detected because it involves a certain sequence of memory accesses, then it is checked to determine if it is a *true* violation before being reported. In both cases, this is implemented by re-executing the program and altering the sequence of executing threads such that the relative order in which the involved memory accesses occur during execution is changed. This is possible since the *Suppression and Concurrency Bug Detection Tool* is implemented within the *Valgrind* infrastructure, which gives us control of the thread scheduler during program execution. Importantly, when altering the relative order of involved memory accesses, any associated acquirements/releases of locks around these memory accesses have to be moved correspondingly to avoid deadlock. For data races, we execute the program one additional time and force the two involved memory accesses to occur in reverse order. For atomicity and order violations involving three memory accesses in which the middle access is from an interleaved thread, we execute the program two additional times, once each for moving the interleaved thread either before or after the other two memory accesses. For the special case of an order violation involving only two memory accesses, this is checked in the same way as for data races, by executing the program one additional time and reversing the relative order of the two accesses. The approach then checks whether the behavior of the program under the alternate thread interleavings indicates a harmful data race or violation, and reports the concurrency bug if so.

The Approach Driver Program. This is the main driver program for carrying out our approach. Given a faulty program and an associated thread interleaving that causes a failure, this program invokes the *Suppression and Concurrency Bug Detection Tool* initially using an empty set of suppression points and no specified target instruction instance. The *Suppression and Concurrency Bug Detection Tool* then records memory access tracing information from the execution, and simultaneously invokes the *Failure Repeatability Component* to record the order in which threads are scheduled during execution. Based on this information, an initial suppression point (and a target write instruction instance at which to search for a concurrency bug) is identified. This information is then passed to another invocation of the *Suppression and Concurrency Bug Detection Tool*. If no concurrency bug is reported but another failure occurs, then the process iterates again. Eventually, either a concurrency bug will be reported as the likely root cause, or else the execution will terminate without any failure, in which case the most recent suppression point is reported as the likely root cause.

4. EXPERIMENTS

To study the effectiveness of our approach in isolating the root causes of multithreading bugs that involve memory corruption, we selected a set of multithreaded programs and their associated bugs as shown in Table I. This set of programs involves six data race bugs (one in `apache`, two in `pbzip2`, two in `mysqld`, and one in `mozilla`), two atomicity violation bugs (one in `mysqld` and one in `mozilla`,), one order violation bug (in `mozilla`), as well as four other memory bugs: one uninitialized read (in `mysqld`), as well as three stack buffer overflows (in programs `prozilla` and `axel`). We selected these subject programs because they are relatively well known and widely used, and they are multithreaded (although not all executions of each

Table I. Subject programs and their associated bugs used in our experiments.

Program name	# Lines of code	Program description	Root cause of failure
apache	191 K	HTTP server (ver. 2.0.48)	Data race [29]
pbzip2-1	2 K	Parallel file compressor (ver. 0.9.4)	Data race [30]
pbzip2-2	2 K	Parallel file compressor (ver. 0.9.4)	Data race [15]
mysqld-1	924 K	MySQL database server (ver. 5.1.25)	Data race [21]
mysqld-2	508 K	MySQL database server (ver. 3.23.56)	Data race [31]
mysqld-3	508 K	MySQL database server (ver. 3.23.56)	Atomicity violation [32]
mysqld-4	508 K	MySQL database server (ver. 3.23.56)	Uninitialized read [33]
mozilla-1	2647 K	Web browser (ver. 1.9.1)	Data race [34]
mozilla-2	1433 K	Web browser (ver. M11)	Atomicity violation [12]
mozilla-3	2454 K	Web browser (ver. 0.9.9)	Order violation [35]
prozilla-1	16 K	Download accelerator (ver. 1.3.5.1)	Stack buffer overflow [36]
prozilla-2	16 K	Download accelerator (ver. 1.3.5.1)	Stack buffer overflow [37]
axel	3 K	Download accelerator (ver. 1.0a)	Stack buffer overflow [38]

Table II. Results for isolating the root causes of the bugs in our experiments.

Program name	# Executions required (orig+supp+race+vio)	Max # entries in trace (size)	# Threads created	Identifies root cause?
apache	3 (1+1+1+0)	16.3 M (391 MB)	28	Yes
pbzip2-1	3 (1+1+1+0)	18.7 M (468 MB)	10	Yes
pbzip2-2	3 (1+1+1+0)	29.1 M (749 MB)	10	Yes
mysqld-1	3 (1+1+1+0)	4.5 M (112 MB)	10	Yes
mysqld-2	3 (1+1+1+0)	2.2 M (51 MB)	5	Yes
mysqld-3	4 (1+1+0+2)	2.1 M (51 MB)	5	Yes
mysqld-4	2 (1+1+0+0)	2.1 M (50 MB)	4	Yes
mozilla-1	3 (1+1+1+0)	15.7 M (394 MB)	3	Yes
mozilla-2	4 (1+1+0+2)	23.4 M (585 MB)	3	Yes
mozilla-3	4 (1+1+0+2)	39.5 M (987 MB)	2	Yes
prozilla-1	2 (1+1+0+0)	2.1 M (52 MB)	1	Yes
prozilla-2	4 (1+3+0+0)	0.75 M (18 MB)	1	Yes
axel	3 (1+2+0+0)	0.27 M (6 MB)	1	Yes

program necessarily involve more than 1 thread). Further, the bugs we selected for our experiments are real errors that have been discovered in practice.

For each of the 13 bugs listed in Table I, we identified an execution that would trigger the bug. This execution was provided as the input to our execution suppression approach to try to isolate the root cause. The results for each of our 13 analyzed bugs are shown in Table II. In this table, the first column shows the program name. The second column shows the total number of executions required by our approach to isolate the root cause. This number is broken down into four components: the original execution to trigger the failure ('orig', which is always 1); the number of suppression executions required to detect concurrency bugs and/or carry out suppression ('supp'); the number of re-executions required to force a different thread interleaving to check whether a detected data race is potentially harmful ('race'); and the number of re-executions required to check potential atomicity/order violation bugs to see if they are true violations ('vio'). The third column in the table shows the maximum trace length recorded by our approach among all re-executions for each benchmark program. The number of entries is specified in millions; each entry contains the relevant information necessary to be recorded at each load and store instruction during execution (the values in parentheses are the number of megabytes of space required to store these traces). The fourth column in the table shows the total number of threads created during execution. The fifth column describes whether the true root cause is identified by our approach. We now describe the results for each of our analyzed bugs in detail.

Program apache. In this subject program, there is a data race bug in which there is a write to a buffer, followed by an update to a buffer count variable. These two memory accesses involve a shared variable that is not protected in a critical section. Further, this piece of code is related to writing information to a server log. In the event that multiple requests are writing to the server log simultaneously, it is possible that the data in the server log will get corrupted due to the data race. In this case, the root cause can be represented by either a WAR or a WAW data race associated with the unprotected instructions.

When our approach is run on this program using two conflicting requests that trigger the bug, a failure occurs when the server log is found to be corrupted. The associated instruction instance of the corrupting write to the log is identified. This instruction instance uses two variables, a pointer to the buffer itself, as well as a count variable. Since either of these can be corrupted, our approach identifies the last instruction instances in the execution in which these two variables were defined. It turns out that the count variable was last defined in the update instruction mentioned above that is associated with a data race. Our approach then performs the first re-execution in which concurrency bug detection (and possibly suppression) will be carried out. The first target instruction reached during the execution is the update to the buffer count variable. At this point, the approach determines that the last access to this shared variable by a *different* thread occurs at the same instruction (but an earlier instance), representing a WAW dependence. Further, there is no synchronization between the executions of these two accesses, so a data race occurs here. The approach performs one more execution to force the different thread interleaving to occur, and determines that the final value in the shared variable in question, has changed at the point of the second memory access. As a result, the approach determines that the identified data race is potentially harmful, and reports it to the user. This data race is in fact directly associated with the expected root cause of the failure. Overall for this subject program, even though the execution involved 28 threads in total, there were only three program executions required by the approach: the original execution to repeat the failure, the first suppression execution in which a data race was detected, and a final execution to determine that the data race was potentially harmful.

Program pbzip2-1. In this subject program, there is a data race bug in which one writer thread writes NULL to a shared heap object named fifo, but fails to acquire a lock on this shared object. Meanwhile, another reader thread (which has the correct lock) reads from this shared object. The program can crash when the reader thread attempts to access this shared object after the writer thread has unexpectedly nullified the object. In this case, the programmer has wrongly assumed that all reader threads will finish before the writer thread executes.

When applying our approach on bug-triggering input for this program, the program crashes when the reader thread tries to access the shared heap object fifo, which has unexpectedly been nullified by the writer thread. It is determined that the last definition (write access) of fifo occurs when the writer thread nullifies it. Our approach then re-executes the program to search on-the-fly for concurrency bugs while also carrying out suppression as necessary. Once execution reaches the identified definition of shared object fifo where it is prematurely nullified, then it is determined that the last access to this shared object by a *different* thread happens to be an earlier read operation in the reader thread. In this case, there is no synchronization between the earlier read and the current write, since the writer thread fails to acquire the correct lock on the object, even while the reader thread has the correct lock. Thus, this is a data race representing a WAR dependence. Our approach next executes the program a third time and forces the write to fifo in the writer thread to be executed *before* the associated read in the reader thread. This alternate thread interleaving causes the resulting value within the shared object to change, so the data race is determined to be potentially harmful and is reported to the user. In this case, the reported data race is indeed associated with the root cause of the failure because it involves the unexpected nullification of object fifo by the writer thread, which fails to acquire the necessary lock on the object.

Program pbzip2-2. This subject program has a similar data race bug to that in program pbzip2-1. Here, a main writer thread performs resource deallocation, nullifying a shared condition variable named `fifo->notFull`. Later, a reader thread attempts to access this shared condition variable, resulting in a program crash since the shared variable was previously unexpectedly deallocated. Synchronization is missing between these two memory accesses.

When our approach is run on this program with input to decompress one large file that will trigger the bug, the program crashes when the reader thread tries to access the condition variable `fifo->notFull` that was previously deallocated by the main writer thread. On the next execution that monitors for concurrency bugs and performs suppression as necessary, execution reaches the point of the last definition of `fifo->notFull`, in which the variable is nullified. At this point, it is discovered that the last access to this shared variable by a *different* thread occurs from a reader thread, and there is no synchronization between these two memory accesses. The next execution forces these two memory accesses to occur in reverse order, and the data race is reported because it is determined to be potentially harmful.

Program `mysqld-1`. This program involves a crashing data race bug. One thread accesses a shared variable named `thd->proc_info` after first testing to ensure that it is not `NULL`. However, after the `NULL` test but prior to accessing the variable, another thread can unexpectedly assign `NULL` to this shared variable, causing a program crash when `NULL` is then subsequently dereferenced. The problem can occur here because there is no synchronization specified around these memory accesses.

When our approach is run on this program using three simultaneous client connections, the program crashes when one thread accesses variable `thd->proc_info`, immediately after another thread unexpectedly nullifies it. The last definition of this accessed memory location occurs when it is unexpectedly nullified. When our approach re-executes this program, execution reaches the unexpected `NULL` definition of variable `thd->proc_info`. At this point, it is found that the last access to this location from another thread is when it is tested to determine if it is `NULL`. Since there is no synchronization between these two accesses, this represents a WAR data race. On the next execution of our approach to determine the potential harmfulness of this data race, the `NULL` definition of the shared memory location is forced to occur *before* the `NULL` test by the reader thread. This results in a modified program state so the data race is determined to be potentially harmful and is reported.

Program `mysqld-2`. In this program, there is a data race bug in which a close and subsequent open of a log file is correctly protected in a critical section, while a read to the log file status is unexpectedly interleaved in-between these two operations because it does not acquire the appropriate lock. This kind of data race is called an *asymmetric race* [39], which occurs when one thread correctly protects a shared variable using a lock, while another thread accesses the same variable *without* correct synchronization. The interleaving read in this case checks the log file status to determine whether it is open or closed, and if open, then writes some messages to the log. It is possible with two conflicting threads that one thread closes the log file, but before re-opening it, another thread notices that the log is closed and therefore does not write its messages to the log. The effect is that some database updates may be silent and will not be recorded in the log.

When executing our approach on bug-triggering input for this program, a failure occurs when it is determined that the log file is closed when it was expected to be open. Our approach determines that the last definition (write access) to the log file status occurred when the log file was closed. On the next execution of our approach, execution proceeds until it reaches the point at which the log file is marked as closed. At this point, it turns out that there are no prior accesses to this variable from a *different* thread (only from the same thread), so there are no WAR or WAW data races, and no atomicity or order violations, up to this point. Thus, execution continues from this point while performing suppression and also monitoring accesses to this variable for possible RAW data races and other violations. Once execution reaches the point at which the log file is accessed (read) and determined to be closed, then our approach discovers that no synchronization was specified between the current point and the last point at which the log was marked as closed (by a different thread). Thus, a RAW data race is detected at this point. Next, the approach executes the program again and forces the update to the log to occur *before* the log is closed by the other thread. It turns out that this new interleaving changes what is contained in the log, so it is determined that the RAW data race is potentially harmful, and it is reported as the root cause of the failure.

Program `mysqld-3`. In this program, there is an atomicity violation bug in which an update to the database state and an associated update to the log file are not protected in a critical section. When two requests are being handled simultaneously, the effect of this bug is that the order of entries in the log file may not match the actual order in which updates occurred to the database.

When our approach is run on this program using two simultaneous connections that trigger the bug, a failure occurs when it is determined that the log entries have become out of order. For this subject program, we treat the log itself as shared memory, so our approach identifies the last write to the log that caused the out-of-order entry. It is assumed that this write has been corrupted in some way. During re-execution, the program proceeds while the approach monitors for accesses to the log to look for possible concurrency bugs. In this case, the involved memory accesses are protected by proper synchronization, so no data races are detected. However, a potential violation is detected that involves two memory accesses from one thread, interleaved by the offending write from another thread that causes the out-of-order log. Our approach then executes the program two additional times, each time forcing the interleaved access to occur either before or after the other two memory accesses. It turns out that both of these executions cause the log file entries to become in proper order, so the potential violation is determined to be a true atomicity violation, and is reported as the root cause of the failure.

Program `mysqld-4`. In this program, there is a memory bug in which the act of loading data from an input file into a database table can cause a segmentation fault, if no database has been first selected. The bug in this case is an uninitialized read of the variable `thd->db` that should be associated with an open database, but is instead unexpectedly `NULL`.

When our approach is run on bug-triggering input for this program, it is determined that a segmentation fault occurs at a call to `strlen` when the pointer passed to it is `NULL`. Our approach determines that the last definition of `thd->db` occurs at an earlier instruction instance in the execution. When our approach re-executes the program, control reaches the initial definition of the `NULL` variable. At this point, our approach determines that there are no prior accesses to this variable from other threads, so there are no concurrency bugs up to this point. Thus, execution continues until it reaches the point of the second memory access (the read of the value `NULL`). Our approach determines that the write and subsequent read are from two different threads, but in this case, there exists specified synchronization between the two memory accesses, so no RAW data race occurs here. No potential atomicity or order violations occur here either. Thus, execution continues by suppressing all instruction instances directly or indirectly affected by this `NULL` value. It turns out that at the end of this execution, no additional failures occur. Thus, the definition of the `NULL` value is reported as the root cause of the failure. This can help developers to understand that the execution unexpectedly tried to dereference the value `NULL` that was defined at this point.

Program `mozilla-1`. This program contains a data race bug in the `mozilla` Javascript engine. The program can crash when a thread nullifies a shared pointer variable, which is followed by a dereference of this pointer by another thread.

When our approach is run on this program using two threads that trigger the bug, the Javascript engine crashes when one thread tries to access shared pointer `rt->scriptFilenameTable`, which was previously deallocated by another thread. On the next iteration of our approach, execution reaches the point at which variable `rt->scriptFilenameTable` was last defined (where it was assigned the value `NULL`). At this point, it is discovered that the last access to this variable from a *different* thread is found to be a read operation from the other conflicting thread. A WAR data race is identified here since there is no synchronization specified between these two memory accesses. The next execution of our approach forces the alternate order of these two memory accesses to occur during execution, and it is determined that the data race is potentially harmful. This race is reported as being associated with the root cause of the failure.

Program `mozilla-2`. In this program, there is an atomicity violation bug in which initialization of a script protocol and the associated compiling and execution are not protected in a critical section. Due to lack of atomicity protection, the program can crash when one thread unexpectedly sets the script protocol to `NULL` (variable `mCurrentScriptProto` in the program), just after the initialization phase but before the compilation phase in another thread.

When our approach is run on this program using two threads to load, compile, and execute Javascript code that triggers the bug, the program crashes when one thread tries to dereference `mCurrentScriptProto`, which was previously set to `NULL` by another thread. The last definition of this shared variable occurs when it was set to `NULL` by the other thread. Upon re-execution, no data races are identified because synchronization is found to exist between the memory accesses associated with this last definition. However, a $W_1W_2R_1$ memory access pattern related to this shared variable is identified in which the interleaving write is the `NULL` definition of variable `mCurrentScriptProto`. To determine whether this potential violation is a true atomicity or order violation, our approach executes the program two additional times, each time forcing the interleaving write to occur either before or after the other two memory accesses. In this case, *both* re-executions cause the program failure to be avoided, so the violation is identified as a true atomicity violation, and it is reported as the root cause of the failure.

Program `mozilla-3`. An order violation bug is present for this subject program in `mozilla XPCOM`, a cross-platform component object model. In a rare situation, the program can hang forever when `TimerThread::Shutdown` sends an `exit` signal *before* `TimerThread::Run` goes to sleep and waits for that signal. This scenario breaks the programmer assumption that the `Run` thread always finishes its computation and waits for the exit signal *before* the `Shutdown` thread sends this signal. For our approach to work with this program that can result in a hanging execution, we modified our tool to kill a program execution if the same instruction executes repeatedly for a long threshold length of time.

When our approach is run on this program using two threads that trigger the bug, the program is terminated by our tool when the `Run` thread waits for a long time, continually reading for the `exit` signal implemented by the shared condition variable named `mCondVar`. It is determined that the last definition of this shared variable occurs when it is set by the `Shutdown` thread. Upon re-execution, control reaches this last definition but no data races are detected up to this point since the associated memory accesses are synchronized. Also, no potential violations are identified at this point either. Execution then continues by suppressing while also monitoring for other data races or potential violations. Once execution reaches the read operation in the `Run` thread at which the program begins to hang, a $W_1W_2R_1$ memory access pattern is identified for variable `mCondVar`, representing a potential violation. Our approach, therefore, executes the program two additional times to determine whether this violation is a true atomicity or order violation bug. First, the program is re-executed while forcing the last definition of the shared variable in the `Shutdown` thread to occur *before* the other two memory accesses. In this case, the same program failure occurs. Next, the program is re-executed while forcing the last definition of the shared variable in the `Shutdown` thread to occur *after* the other two memory accesses. In this case, the failure is avoided. Since only one of these two alternate thread interleavings causes the failure to be avoided, an order violation bug is reported as the root cause of the failure for this program.

Program `prozilla-1`. In this program, there is potential for a stack buffer overflow at an unchecked call to `strncpy`; in this case, no check is made to ensure that the source string will actually fit into the allocated destination buffer.

When our approach is run on an input that triggers the stack overflow, a crash occurs at the return of a function called `parse_html_mirror_list`, due to corruption of the return address of the function call on the stack. Our approach determines that the last definition of the memory location associated with this corrupted return address is in fact from the unchecked call to `strncpy`. Upon re-execution, there are no data race, atomicity violation or order violation bugs identified since all instructions associated with the failure are from the same thread. As a result of the suppression conducted on this execution, it turns out that no additional failures occur during execution. Our approach then reports the faulty call to `strncpy` as the root cause of the failure.

Program `prozilla-2`. In this program, there is potential for a stack buffer overflow of a variable called `buffer`, which is declared to be of fixed maximum size on the stack. A call to `sprintf` using this buffer can potentially write more data into the buffer than what will fit into the allocated size.

When our approach is run on this program using an input that triggers the overflow, a segmentation fault first occurs when dereferencing a pointer that is declared on the stack. This is due to the pointer variable on the stack being corrupted by the stack overflow. It is determined that the last definition of this corrupted memory location occurred in the unchecked `sprintf`. No concurrency bugs are detected during the first re-execution because there is only one thread present in the execution. When our approach performs suppression during the first re-execution, another crash occurs at the return from a function named `http_fetch_headers` (due to the function call return address being corrupted by the stack overflow). It is determined that the corrupted return address was also last defined in the unchecked `sprintf`. The program is then executed again to conduct further suppression, but a third failure occurs at the return of a function called `get_http_info`. The corrupted return address in this case is again last defined at the unchecked `sprintf`. Finally, when the program is executed once more to conduct suppression, no additional failures occur. Thus, the root cause is identified to be the unchecked call to `sprintf`. In all of these suppression executions, no concurrency bugs were found since only one thread was ever present during execution.

Program `axel`. In this program, there is potential for a stack buffer overflow in an unchecked call to `sscanf`. In this case, the destination stack buffer is declared to be of fixed size 256 bytes. Since the source string can be of arbitrary size in this case, the stack buffer may overflow.

When our approach is run on this program, a first segmentation fault occurs when dereferencing a pointer that is declared on the stack. The last definition of the corrupted memory location was in the unchecked call to `sscanf`. On the first re-execution, no concurrency bugs are detected since only one thread is present. After suppressing during this execution, a new crash occurs at the return of a function called `conn_info`, due to a corrupted function call return address. Again, the last definition of the corrupted location was in the unchecked call to `sscanf`. Upon the next suppression execution, no additional failures occur, and the unchecked call to `sscanf` is identified as the root cause of the failure. No concurrency bugs were detected in these suppression executions since only one thread was ever present during execution.

Overall, we found that our approach could accurately identify the root cause of the failures in all of our subject programs. On the other hand, these subject programs happened to involve little propagation of corrupted memory. In general, some multithreading bugs may involve significant propagation of corrupted memory, and in these cases, we expect that our approach may continue to be effective due to the iterative nature of the execution suppression technique.

In terms of the number of threads, program `apache` required the most at 28 threads during execution. However, only two threads were involved in the data race that caused the failure for this program, and our approach focused only on these relevant threads when identifying the root cause. In terms of trace size, program `mozilla-3` involved the largest trace for any single execution: just under 40 million entries, requiring just under 1 GB of space for storage. We stored the necessary tracing information naively in our experiments, but we believe that more sophisticated compression schemes could be used to significantly reduce this storage requirement if necessary.

All subject programs in our experiments required only between two and four program executions to identify the root cause using our approach. However, the time required to run each execution using

Table III. Time to run the longest execution, and total time to run the approach, for each program.

Program name	Time to perform longest execution (in s)			Total time (s)
	Native execution	'Stock' Valgrind	Tracing Valgrind	
apache	12.5	18.0	26.5	80.5
pbzip2-1	0.6	2.0	28.2	36.2
pbzip2-2	1.0	5.7	44.2	67.3
mysqld-1	3.4	10.9	112.2	155.7
mysqld-2	1.1	2.8	38.4	49.6
mysqld-3	1.0	3.1	39.2	58.6
mysqld-4	1.3	2.6	36.6	41.8
mozilla-1	0.7	10.1	35.2	63.1
mozilla-2	1.0	2.1	8.4	21.0
mozilla-3	2.5	4.3	122.5	148.3
prozilla-1	2.2	2.5	3.7	7.5
prozilla-2	0.03	0.17	0.83	1.85
axel	0.002	0.122	0.387	0.753

our approach varied widely from among the subject programs. Table III shows the time required to run the *longest* execution (in terms of trace size) for each benchmark program in our experiments, as well as the total time required to run all executions of our approach for each program. The left-most column shows the program name. The next three columns show the time required to run the single *longest* execution for each benchmark program, broken up into three types of executions: (1) the native execution time when the program is run on the original machine, outside of the Valgrind environment; (2) the execution time for running in 'stock' Valgrind, that is basic Valgrind without any additional instrumentation added during execution; and (3) the execution time within Valgrind when the execution is instrumented to record the tracing information necessary to run our approach. Finally, the right-most column shows the total time required to run all executions of our approach for each subject program, including the time required to perform suppression as well as to perform analysis for concurrency bugs. In the worst case, program `mysqld-1` required about 2.5 min in total to run our approach. Most programs required less than 2 min, and three programs required less than 10 s. We believe these running times are reasonable in a debugging context. Given this, we also observe that the use of Valgrind can impose a very high runtime overhead over native execution times. This is because Valgrind was designed for ease of program instrumentation and not for runtime efficiency. For example, in program `mozilla-1`, the native execution time is about 0.7 s, but the time to execute the program while tracing in Valgrind is over 35 s, a slowdown factor of 50×. The slowdown factor is less in other programs: in `apache`, the native execution time is about 12 s, but the Valgrind tracing time is about 26 s, a slowdown factor of just over 2×. One reason the slowdown factor is so low in `apache` is because much of the runtime in both the native and Valgrind tracing executions is spent waiting for many server connections to be made. During this time, the native execution time increases relative to the Valgrind tracing execution time, since Valgrind is not performing any tracing while the execution waits. Thus, it is the nature of the given program that determines the slowdown factor observed under Valgrind, and this varies for different programs.

5. RELATED WORK

We first proposed the idea of execution suppression for locating memory bugs in [40]. In this prior work, a basic execution suppression approach was described, and some experimental results were given that highlight the effectiveness of execution suppression. However, this prior work considered only single-threaded programs and their associated memory bugs. The current work is more comprehensive because it generalizes the execution suppression approach to be applicable and effective for multithreaded programs. In addition to handling memory bugs such as buffer overflows and uninitialized reads in multithreaded code, our approach can now also detect when a harmful data race, atomicity violation, or order violation is the likely root cause of a failure in a multithreaded program.

Our current work that generalizes our approach to handle multithreading bugs is inspired by several key observations taken from prior work. The first of these comes from Tallam *et al.* work on extending dynamic slicing to capture data race bugs [11]. In this work, the authors observed that dynamic slicing traditionally considers only RAW data and control dependencies, and this may result in dynamic slices failing to capture relevant data race bugs in an execution. To remedy this limitation, the authors proposed a generalized dynamic slicing algorithm that also considers dynamic WAR and WAW dependencies. In our current work, we have taken this observation about the three important kinds of dependencies to consider for data races, and have incorporated it into our algorithm for detecting data races. We have also incorporated into our algorithm for detecting atomicity and order violations, the observation by Lu *et al.* [12] that only four particular sequences of memory accesses need to be considered when detecting atomicity violations involving three memory accesses (in which the middle access is caused by an interleaving thread). Finally, the work by Narayanasamy *et al.* on classifying benign and harmful data races [13] has provided the key observation that not all data races are potentially harmful, and one way to check the potential harmfulness of a data race is to repeat an execution twice, each time forcing the two involved memory accesses to occur in a different relative order. If the order of the two memory accesses influences the resulting values of variables in memory, then the data race is potentially harmful. In our approach, we use this key idea of repeating an execution while altering the relative order of memory accesses, to determine whether identified data races or potential atomicity/order violations are indeed harmful and should be reported as the likely root cause for a failure.

Handling concurrency bugs. Lu *et al.* [21] conducted a comprehensive study of real-world concurrency bug characteristics, which serves as a useful guide for concurrency bug testing and detection. Zhang *et al.* [30] also conducted a study of concurrency bugs that result in program crashes to identify common thread interleavings that are typically associated with memory bugs. Based on this study, they developed a technique to predictively detect concurrency bugs that result in crashes. Park *et al.* [15] developed a technique that focuses on exposing atomicity violation bugs that may be difficult to expose in practice. Their approach looks for particular types of interleavings that are inherently correlated with atomicity violations, and uses trace analysis to systematically identify such interleavings that are likely to be feasible in a program, but are unlikely to occur in practice. Farzan *et al.* [41] developed another technique for detecting atomicity violations, and argued in their work that if a concurrent program uses *nested locking*, then the task of predicting common types of atomicity violations can be solved efficiently. Recent works [39, 42] have focused on detecting and tolerating *asymmetric data races*, race conditions in which one thread correctly acquires and releases a lock for a shared variable, while another ill-behaved thread disobeys the locking discipline for this variable.

Other approaches for detecting general data races in multithreaded programs can be classified into those that use the *happens-before* algorithm [1, 43, 44], the *lockset* algorithm [3, 5, 45, 46], or a hybrid algorithm that combines both approaches [2, 4, 47]. The key idea behind lockset-based algorithms is to check whether shared variables are protected by at least one lock. The algorithm employs heuristics that can cause false positives to be reported. On the other hand, the key idea behind happens-before algorithms is to check whether accesses to shared variables in a program are explicitly ordered through synchronization operations. This approach may miss some data races, but all identified data races will be true data races (though some of them may be benign). Hybrid algorithms generally attempt to achieve coverage close to that of lockset-based algorithms, while reducing false positives. In the current work, our approach for detecting data races is based on a happens-before relationship because it identifies conflicting memory accesses for which there is no explicit synchronization between them. Thus, all identified data races in our approach are true data races. However, our approach further checks to see if a data race is potentially harmful before reporting it as a likely root cause of a failure.

Handling other memory bugs. Besides the current work, there has been other work that focuses on detecting memory bugs. *Valgrind* [8] and *Purify* [9] can be used to detect memory bugs, but are restrictive in that they look for particular kinds of memory bugs. Our approach is more general and can be used for any memory bugs involving corrupted memory, such that they exhibit failures revealing at least a subset of the memory corruption. However, *Purify* can be used to detect memory leaks, while our approach cannot be directly applied to memory leaks since leaks do not involve corrupted memory. *CCured* [10] is an approach for verifying type-safety of pointers both statically and during runtime, which can be used to find potential memory bugs.

There has been recent work on protecting against heap-based memory errors to improve program reliability. *DieHard* [48] provides memory safety with high probability by randomizing the location of objects in a large heap and by replicating execution. *Archipelago* [49] allocates heap objects far apart in virtual address space to combat buffer overflows, and protects against dangling pointer errors by preserving freed objects after they are freed. *Exterminator* [50] pinpoints heap-based memory errors and derives runtime patches to avoid them in the current and subsequent executions. Unlike these approaches that are targeted toward heap-based memory errors, our current work targets a more general class of memory errors that involve corrupted memory, which may involve memory other than the heap, and which may also include concurrency bugs. While our current work attempts to isolate general memory corruption in an execution, the *Samurai* system [51] provides safeguards against corruption of critical data through a memory model called *critical memory*. Their system uses replication and forward error correction to ensure that non-critical updates do not corrupt critical data. However, the system requires that critical memory be explicitly identified by a programmer.

General fault localization techniques. There has been significant prior research on techniques for locating bugs that often does not explicitly deal with multithreading bugs. *Slicing-based techniques* are one such thread of research (though it has recently been extended [11] to handle data race bugs). *Static slicing* [52] identifies a subset of program statements that *may* influence the value of a variable at a program location. *Dynamic slicing* [53–55] finds the statements that actually *do* influence a variable value in a particular execution. *Relevant Slicing* [56, 57] is similar to dynamic slicing, but additionally finds statements that can *potentially* influence a variable value in an execution, if a predicate were to evaluate to a different outcome. All slicing-based approaches identify a subset of program statements that must be examined by a user to locate a bug. This set of statements can potentially be very large. Unlike slicing-based techniques, our current approach seeks to identify only a single statement (a few statements on rare occasions) or concurrency bug that is likely to be the root cause for a failure.

Another thread of research in this area involves *state-altering approaches*. These approaches attempt to modify the state of program executions in particular ways to gain insight about the likely location of a bug. In *Delta Debugging*, failure-inducing input is identified [58] that allows for the computation of cause–effect chains for failures [59], which can in turn be linked to faulty code [60]. This approach involves substituting state (the values of variables) between passing and failing runs. A related *Value Replacement* idea was proposed [61] that attempts to replace the values used at certain statement instances with alternate sets of values; if any value replacement causes a failing run to become successful, then the statement associated with the value replacement may be erroneous. *Predicate Switching* [62] attempts to isolate erroneous code by identifying 'critical' predicates whose outcomes can be altered during a failing run to cause it to become successful.

Still other approaches make use of statistical information [63–66]. For example, the *Nearest Neighbor* approach [66] compares the spectra for two similar executions (one successful and one failing) to identify the most suspicious parts of a program. *Tarantula* [63] ranks program statements according to suspiciousness values determined by how many failing versus passing tests exercise each statement.

6. CONCLUSIONS

We have presented a general approach for isolating the root cause of a failure in a multithreaded program. Our approach is designed to be general so that it can be applied to any multithreading bugs that involve memory corruption during execution, and that produce a program failure revealing at least a subset of this memory corruption. The approach can identify either a harmful concurrency bug (data race, atomicity violation, or order violation) or a statement causing memory corruption that is the likely root cause for a failure. Our approach can also be automated if a failure-triggering thread interleaving for a faulty program is provided. Moreover, our approach includes an iterative technique called *execution suppression* that can ensure effectiveness of our approach even in the presence of significant propagation of corrupted memory during execution. Our experimental results on a set of real bugs in large-scale multithreaded programs have shown that our approach can be very effective at precisely identifying the root causes of failures caused by multithreading and other memory-related bugs.

REFERENCES

1. Christiaens M, Bosschere KD. TRaDe: A topological approach to on-the-fly race detection in Java programs. *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX Association: Berkeley, CA, U.S.A., 2001; 105–116.
2. OCallahan R, Choi JD. Hybrid dynamic data race detection. *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., June 2003; 167–178.
3. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 1997; **15**(4):391–411.
4. Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. *ACM SIGOPS Operating Systems Review* 2005; **39**(5):221–234.
5. Zhou P, Teodorescu R, Zhou Y. HARD: Hardware-assisted lockset-based race detection. *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*. IEEE Computer Society: Los Alamitos, CA, U.S.A., February 2007; 121–132.
6. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 2006; **32**(3):176–192.
7. Yang J, Sar C, Engler DR. EXPLODE: A lightweight, general system for finding serious storage system errors. *Seventh Symposium on Operating Systems Design and Implementation*. USENIX Association: Berkeley, CA, U.S.A., November 2006; 131–146.
8. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 2007; 89–100.
9. Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter Technical Conference*. USENIX Association: Berkeley, CA, U.S.A., 1992; 125–136.
10. Necula GC, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*. ACM: New York, NY, U.S.A., January 2002; 128–139.
11. Tallam S, Tian C, Gupta R. Dynamic slicing of multithreaded programs for race detection. *International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos, CA, U.S.A., September 2008; 97–106.
12. Lu S, Tucek J, Qin F, Zhou Y. AVIO: Detecting atomicity violations via access interleaving invariants. *Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., October 2006; 37–48.
13. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B. Automatically classifying benign and harmful data races using replay analysis. *International Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 2007; 22–31.
14. Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. *PLDI '07*: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2007; 446–455.
15. Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. *Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., 2009; 25–36.
16. Altekar G, Stoica I. Odr: Output-deterministic replay for multicore debugging. *SOSP '09*: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM: New York, NY, U.S.A., 2009; 193–206.
17. Narayanasamy S, Pokam G, Calder B. BugNet: Continuously recording program execution for deterministic replay debugging. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE Computer Society: Washington, DC, U.S.A., June 2005; 284–295.
18. Park S, Zhou Y, Xiong W, Yin Z, Kaushik R, Lee KH, Lu S. Pres: Probabilistic replay with execution sketching on multiprocessors. *SOSP '09*: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM: New York, NY, U.S.A., 2009; 177–192.

19. Weeratunge D, Zhang X, Jagannathan S. Analyzing multicore dumps to facilitate concurrency bug reproduction. *ASPLOS '10*: *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., 2010; 155–166.
20. Flanagan C, Freund SN. Atomizer: A dynamic atomicity checker for multithreaded programs. *POPL '04*: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM: New York, NY, U.S.A., 2004; 256–267.
21. Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News* 2008; **36**(1):329–339.
22. Park C, Sen K. Randomized active atomicity violation detection in concurrent programs. *International Symposium on Foundations of Software Engineering*. ACM: New York, NY, U.S.A., 2008; 135–145.
23. Sen K. Effective random testing of concurrent programs. *International Conference on Automated Software Engineering*. ACM: New York, NY, U.S.A., 2007; 323–332.
24. Sen K. Race directed random testing of concurrent programs. *ACM SIGPLAN Notices* 2008; **43**(6):11–21.
25. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. *IBM Systems Journal* 2002; **41**(1):111–125.
26. Lu S, Jiang W, Zhou Y. A study of interleaving coverage criteria. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM: New York, NY, U.S.A., September 2007; 533–536.
27. Tian C, Nagarajan V, Gupta R, Tallam S. Dynamic recognition of synchronization operations for improved data race detection. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ACM: New York, NY, U.S.A., July 2008; 143–154.
28. Tian C, Nagarajan V, Gupta R, Tallam S. Automated dynamic detection of busy-wait synchronizations. *Software*: *Practice and Experience* 2009; **39**(11):947–972.
29. Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y. BugBench: Benchmarks for evaluating bug detection tools. *Workshop on the Evaluation of Software Defect Detection Tools Co-located with PLDI*. ACM: New York, NY, U.S.A., June 2005.
30. Zhang W, Sun C, Lu S. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. *ACM SIGPLAN Notices* 2010; **45**(3):179–192.
31. http://bugs.mysql.com/bug.php?id=791.
32. http://bugs.mysql.com/bug.php?id=169.
33. http://bugs.mysql.com/bug.php?id=110.
34. https://bugzilla.mozilla.org/show_bug.cgi?id=515403.
35. Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. *ACM SIGARCH Computer Architecture News*. ACM: New York, NY, U.S.A., June 2009; 325–336.
36. http://www.securityfocus.com/bid/12635.
37. http://bugs.gentoo.org/show_bug.cgi?id=70090.
38. http://www.securityfocus.com/bid/13059.
39. Ratanaworabhan P, Burtscher M, Kirovski D, Zorn B, Nagpal R, Pattabiraman K. Detecting and tolerating asymmetric races. *Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., 2009; 173–184.
40. Jeffrey D, Gupta N, Gupta R. Identifying the root causes of memory bugs using corrupted memory location suppression. *IEEE International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos, CA, U.S.A., September 2008; 356–365.
41. Farzan A, Madhusudan P, Sorrentino F. Meta-analysis for atomicity violations under nested locking. *Proceedings of the 21st International Conference on Computer Aided Verification*. Springer-Verlag: Berlin, Heidelberg, 2009; 248–262.
42. Rajamani S, Ramalingam G, Ranganath V, Vaswani K. ISOLATOR: Dynamically ensuring isolation in concurrent programs. *Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., 2009; 181–192.
43. Mellor-Crummey J. On-the-fly detection of data races for programs with nested fork-join parallelism. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. ACM: New York, NY, U.S.A., November 1991; 24–33.
44. Ronsse M, Bosschere KD. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems* 1999; **17**(2):133–152.
45. Flanagan C, Freund SN. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming* 2008; **71**(2):89–109.
46. Krena B, Letko Z, Tzoref R, Ur S, Vojnar T. Healing data races on-the-fly. *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems*: *Testing and Debugging*. ACM: New York, NY, U.S.A., July 2007; 54–64.
47. Dinning A, Schonberg E. Detecting access anomalies in programs with critical sections. *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM: New York, NY, U.S.A., May 1991; 85–96.
48. Berger ED, Zorn BG. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 2006; 158–168.
49. Lvin VB, Novark G, Berger ED, Zorn BG. Archipelago: Trading address space for reliability and security. *13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., March 2008; 115–124.

50. Novark G, Berger ED, Zorn BG. Exterminator: Automatically correcting memory errors with high probability. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 2007; 1–11.
51. Pattabiraman K, Grover V, Zorn B. Samurai: Protecting critical data in unsafe languages. *EuroSys '08*. ACM: New York, NY, U.S.A., April 2008; 219–232.
52. Weiser M. Program slicing. *IEEE Transactions on Software Engineering* 1984; **10**(4):352–357.
53. Agrawal H, Horgan JR. Dynamic program slicing. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 1990; 246–256.
54. Korel B, Laski J. Dynamic program slicing. *Information Processing Letters* 1988; **29**(3):155–163.
55. Zhang X, Gupta N, Gupta R. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2006; 169–180.
56. Agrawal H, Horgan JR, Krauser EW, London S. Incremental regression testing. *IEEE International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos, CA, U.S.A., September 1993; 348–357.
57. Gyimothy T, Beszedes A, Forgacs I. An efficient relevant slicing method for debugging. *ACM/SIGSOFT Foundations of Software Engineering*. Springer-Verlag: London, U.K., September 1999; 303–321.
58. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**(2):183–200.
59. Zeller A. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*. ACM: New York, NY, U.S.A., November 2002; 1–10.
60. Cleve H, Zeller A. Locating causes of program failures. *27th International Conference on Software Engineering*. ACM: New York, NY, U.S.A., May 2005; 342–351.
61. Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*. ACM: New York, NY, U.S.A., July 2008; 167–178.
62. Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. *Proceedings of the 28th International Conference on Software Engineering*. ACM: New York, NY, U.S.A., 2006; 272–281.
63. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering*. ACM: New York, NY, U.S.A., May 2002; 467–477.
64. Liblit B, Naik M, Zheng A, Aiken A, Jordan M. Scalable statistical bug isolation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., June 2005; 15–26.
65. Liu C, Yan X, Fei L, Han J, Midkiff S. SOBER: Statistical model-based bug localization. *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM: New York, NY, U.S.A., September 2005; 286–295.
66. Renieris M, Reiss S. Fault localization with nearest neighbor queries. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society: Los Alamitos, CA, U.S.A., October 2003; 30–39.