Detecting, Exposing, and Classifying Sequential Consistency Violations

Mohammad Majharul Islam and Abdullah Muzahid
University of Texas at San Antonio
{*mohammadmajharul.islam, abdullah.muzahid*}*@utsa.com*

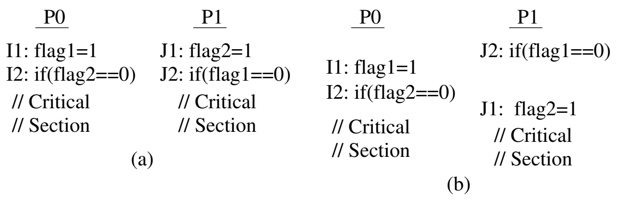
***Abstract*—Sequential Consistency (SC) is the most intuitive memory model for parallel programs. However, modern architectures aggressively reorder and overlap memory accesses, causing SC violations. An SC violation is virtually always a bug. Most prior schemes either search the entire state space of a program, or use a constraint solver to find SC violations. A promising recent scheme uses active testing technique but fails to be effective for SC violations involving larger number of threads and variables, and larger codebases. We propose Orion, the *first* active testing technique that can detect, expose, and classify any arbitrary SC violations in any program. Orion works in two phases. In the first phase, it finds potential SC violation cycles by focusing on racing accesses. In the second phase, it exposes each SC violation cycle by enforcing the exact scheduling order. We present a detailed design of Orion in the paper. We tested different concurrent algorithms, bug kernels, SPLASH2, PARSEC applications, and an open source program, Apache. We experimented with TSO and PSO memory models. We detected and exposed 60 SC violations of which 15 violations involve more than two processors and variables. Orion exposes SC violations quickly and with high probability. Compared to a state-of-the-art active testing technique, it has a much better SC violation detection ability.**

***Keywords*-Memory model; Sequential consistency; Active testing; Parallel programming**

I. Introduction

Among various memory models, Sequential Consistency (SC) [1] is the most intuitive one. SC guarantees a total global order among the memory operations where each thread maintains its program order. However, most commercial architectures sacrifice SC to improve performance. For example, x86 implements a memory model similar to TSO [2] which allows a later load operation to bypass an earlier store operation from the same processor. The overlapping and reordering of memory accesses can lead to non-SC behavior of a program, referred to as an *SC Violation*.

Let us consider Dekker's algorithm in Figure 1(a). Processor P0 first writes *flag1* (I1) and then reads *flag2* (I2) but P1 first writes *flag2* (J1) and then reads *flag1* (J2). Both flags are initially 0. In SC, either I2 or J2 will be the last one to complete. Therefore, either P0 finds *flag2* to be 1 or P1 finds *flag1* to be 1. It is even possible to have both flags to be 1 (e.g., if the completion order is I1, J1, I2, and J2). In any case, we can *never* have both flags to be 0. As a result, only one processor can enter into the critical section at any time. However, if the underlying memory model is TSO, it is possible for the load in J2 to bypass the store in J1 (Figure 1(b)). As a result, the completion order becomes J2, I1, I2, and J1. Both processors find the flags to be 0 and enter into the critical section simultaneously. The same problem can occur if I1 and I2 get reordered.



```
P0                    P1
I1: flag1=1           J1: flag2=1
I2: if(flag2==0)      J2: if(flag1==0)
 // Critical           // Critical
 // Section            // Section
              (a)

P0                    P1
                      J2: if(flag1==0)
I1: flag1=1
I2: if(flag2==0)
                      J1:  flag2=1
 // Critical            // Critical
 // Section             // Section
              (b)
```

Figure 1. (a) shows Dekker's algorithm and (b) shows how an SC violation can occur there.

Detecting SC violations is crucial. Maintaining SC behavior is considered to be one of the correctness criteria for parallel programs. Programmers can ensure SC semantics in any architecture by writing the programs in a data race free manner [3], [4]. However, parallel programs can have occasional data races (intentional or unintentional) and hence, SC violations can occur. The situation gets complicated when memory model specifications of commercial processors from Intel and AMD do not even match with the actual behavior of the machines [5]. Therefore, programmers might not be able to reason about SC behavior with those specifications. Last but not least, a recent study [6] has shown that many real world applications like Apache, MySQL, Mozilla, Gcc, Java, Cilk [7], Splash2 etc. have SC violations. Only 20% of those bugs are detected by existing software testing tools. The rest are discovered by programmers during the analysis of source code. Such findings warrant a tool that can detect, expose, and even classify SC violations in any program.

Significant research has been done to detect SC violations. One line of work [8]–[10] encodes programs and memory model constraints as axioms and use a constraint solver to find SC violations. There are some proposals [11]–[13] to search the state space of a program to find SC violations. Aglave et al. [14] proposed to use static analysis to detect critical SC violation cycles in an attempt to insert necessary fences. Sober [15] and Burnim et al. [16] proposed a run time monitoring system to check an SC execution in an attempt to find SC violations in close-by relaxed executions. Such systems should run along with a model checker to detect all possible SC violations. Recently, Burnim et al. [17] proposed an active testing technique, called Relaxer. It first finds potential SC violations and then, exposes them by buffering some stores while speeding up or stalling certain other thread. Relaxer is effective but cannot expose violations that require complex thread scheduling (e.g., executing a long sequence of accesses in a certain order) or some stores to be pending for a long time. Therefore, it cannot expose SC violations involving more threads (e.g., > 2) and variables (e.g., > 2).

We propose an active testing technique, called Orion, to detect, expose, and classify any SC violation - no matter how many threads and variables are involved or how complex thread interleavings need to be. We name our scheme *Orion* after the Greek God of hunting. Orion starts with a data race detector to find the races. We collect some execution traces of the racing accesses and construct a Happened-before [18] graph to determine cycles [18]. The cycles can be of any length and involve any number of variables. Each cycle can potentially create an SC violation. For each cycle, we run the program with a custom scheduler that enforces the exact order needed to expose SC violation in a particular memory model. Finally, we classify the violation based on the execution outcome. This is the *first* active testing technique that can detect, expose and classify any arbitrary SC violation in any program.

Code	Cycle length	Number of SC violations
harris	4	1
bakery	4	1
init	4	2
snark	4	1
lazylist	4	2
dekker	4	3
ms2	4	1
pthread	4	9
	6	9
	8	1
crypt	4	24
	6	3
msn	4	1
	5	1
	6	1
Total		60

Table I
SC VIOLATIONS DETECTED.

We present a detailed design of Orion in this paper. We developed our profiler and scheduler using a binary instrumentation tool, Pin [19]. We tested different concurrent algorithms, bug kernels, SPLASH2 [20] and PARSEC [21] applications, and a large open source program, Apache. We experimented with TSO and PSO memory models. Table I summarizes the total (i.e., TSO+PSO) SC violations found in different programs. We did not find any SC violation in SPLASH2, PARSEC, and Apache. benchmarks. We compared our scheme against Relaxer [17] to show its superior detection ability.

The paper is organized as follows. Section II gives some background; Section III describes Orion design; Section IV presents experimental results; Section V discusses related work; and finally, Section VI concludes the paper.

II. Background

A. Pattern for a Sequential Consistency Violation (SCV)

Shasha and Snir [22] show what leads to an SC violation: overlapping data races that cause dependences to form a Happened-before cycle at runtime. Recall that a data race occurs when two (or more) threads access the same variable without an intervening synchronization and at least one is writing. Figure 2(a) shows the required program pattern for two threads (where each variable is written at least once) and Figure 2(b) shows the required order of dependences observed at runtime for SC violations. We assigned reads and writes to the variables arbitrarily.

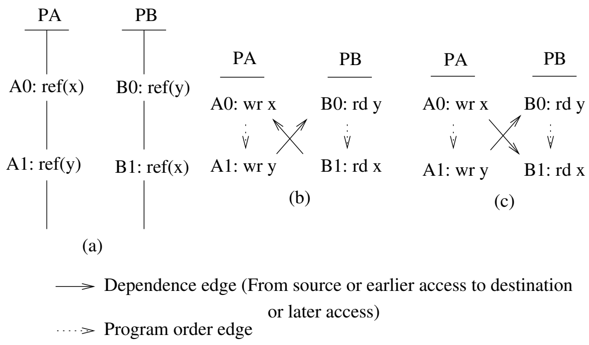


Figure 2. Understanding SC violations.

If at least one of the dependences occur in the opposite direction (e.g., Figure 2(c)), no cycle can form and hence, no SC violation occurs. Given the pattern in Figure 2(a), Shasha and Snir [22] avoid SC violations by placing a fence between A0 and A1 and another between B0 and B1. These fences force at least one dependence arrow to go downward.

B. Total Store Order (TSO)

A TSO machine has a write buffer with each processor. When a store reaches the head of the Reorder Buffer (ROB), it retires into the write buffer. From there, the stores are performed in order. Whenever a load reaches the head of the ROB and the data is returned from the local cache, it is allowed to retire (i.e., complete) even if the write buffer contains some earlier stores. When a load completes while the write buffer is non-empty, the load essentially bypasses those earlier stores in the buffer. If the processor executes a load that has the same memory address as one of the pending stores in the write buffer, TSO allows the load to get data from the write buffer even if the store has not been completed yet.

C. Partial Store Order (PSO)

PSO is similar to TSO except that while a store is waiting in the write buffer, a later store to a different address can bypass the earlier store and complete. As in TSO, a processor can load data from a pending store in the write buffer.

III. Overview of Orion

Orion works in 2 phases - detecting potential SC violations and exposing them. We implement the first phase by analyzing some execution traces of the program. For the second phase, we impose some specific scheduling constraints to expose the SC violations. If the violations can be exposed, we further classify them as benign or harmful based on their impact on program execution. If, on the other hand, some violations cannot be exposed, they are discarded as infeasible ones.

A. Phase 1: Detecting Potential SC Violations

The goal of this phase is to find a set of potentials SC violations that we can expose in the second phase. The steps are as follows:

1) Detecting Data Races: An SC violation requires two or more data races to overlap in execution [22]. So, the first step of Phase 1 is to find all data races. There has been significant research on detecting data races [23]–[26]. Therefore, instead of reinventing the wheel, we use an existing race detector such as Intel Parallel Inspector [26]. This is a dynamic data race detector based on Happened-before [18] algorithm. In order to find the data races, we run the tool with a given program multiple times (e.g., 10) with different inputs. At the end, we collect all the data races. Each race is represented by a pair of instructions and a memory location.

2) Collecting Execution Traces: For this step, we write a profiler using a binary instrumentation tool such as Pin [19]. From the previous step, we get a list of data races. For each data race, we know the instructions that are involved. Our profiler takes a list of such instructions as its input. We run the program with the profiler. When the program executes one of the racing instructions, the profiler records the instruction and memory address, thread id, and type of access (i.e., read/write). The profiler also records synchronization operations (e.g., lock, unlock, barrier, thread creation, thread join etc.) along with the id of the thread that executes them. The profiler records everything in an output file. We run the program multiple times (e.g., 10). Each run results in a separate output file. A long running program with many data races may end up creating a large output file which may not be feasible to process further. In order to deal with this issue, we configure our profiler such that for each thread, it records up to a maximum number of dynamic instances of a racing instruction. A programmer can tune this configuration parameter based on his/her timing budget.

3) Creating Race Graphs: For each trace, we create a race graph. A race graph is a directed graph. We construct it using the racing accesses as nodes and data race and program order relations as edges. Two accesses have a data race if they do not have any Happened-before relation and at least one of them is a store operation. Two accesses have a program order relation if they are from the same thread and one appears before the other during program execution.

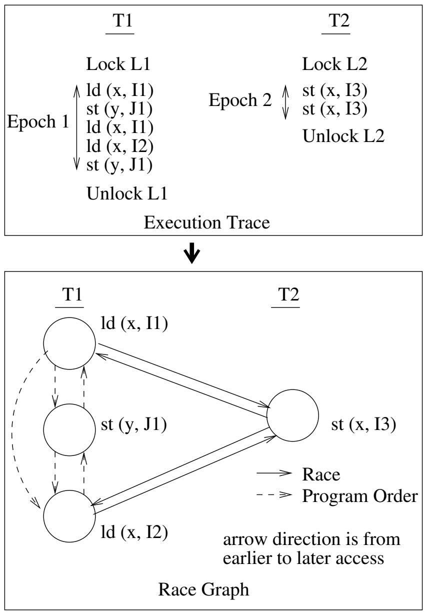


Figure 3. Example race graph created from an execution trace (Phase 1).

Let us consider the execution trace shown in Figure 3. A load is shown as *ld(a, ins),* where *a* is the memory address and *ins* is the instruction address of the load. Similarly, a store is shown as *st(a, ins)*. Let us refer to a thread's execution from one synchronization operation to the next one as an *Epoch*. In each epoch, we create one node for all the loads that access the same memory address and have the same instruction address. The rationale is that if one of those loads has a data race with another access, other loads will also have data race with the same access. All the stores of an epoch that access the same memory address and have the same instruction address also result in a single node in the graph. Thus, Epoch 1 contributes 3 nodes labelled as $ld(x, I_1)$, $st(y, J_1)$, and $ld(x, I_2)$ and Epoch 2 contributes 1 node labelled as $st(x, I_3)$. For each thread, we add a program order edge between 2 nodes if the corresponding accesses in the trace have program order relation (direct or transitive). For example, we add a program order edge from node $ld(x, I_1)$ to node $st(y, J_1)$ because according to the execution trace, $ld(x, I_1)$ appears before $st(y, J_1)$ in program at least in one instance. We add a program order edge from $st(y, J_1)$ to $ld(x, I_1)$ for the same reason. However, we add a program order edge from $st(y, J_1)$ to $ld(x, I_2)$ because the store has a transitive program order relation with the load. Thus, the program order edges of the race graph essentially form a transitive closure of program order relations.

After adding the program order edges, we add data race edges in the race graph. For each epoch, we determine the epochs that are parallel (i.e., they do not have any Happened-before relation). Then, we add data race edges between

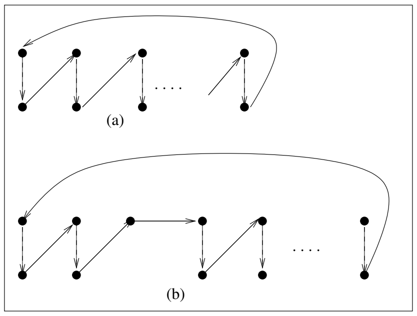


Figure 4. Two types of minimal SC violation cycle.

the conflicting access nodes of the parallel epochs. Recall that two accesses conflict with each other if they access the same address and at least, one of the accesses is a store operation. *Epoch 1* of Figure 3 can execute in parallel with *Epoch 2* (since they are critical sections protected by different locks). Therefore, we add data race edges between conflicting access nodes $ld(x, I_1)$ and $st(x, I_3)$. We add one edge in each direction since any one of $ld(x, I_1)$ and $st(x, I_3)$ can execute first.

4) Detecting SC Violation Cycles: An SC violation cycle is a Happened-before cycle in the race graph. It consists of both program order edges and data race edges. A minimal SC violation cycle can be in one of the two forms [22] as shown in Figure 4. Each cycle can be represented as $A_1 \rightarrow_P A'_1 \rightarrow_R A_2 \rightarrow_P A'_2 ... A'_n \rightarrow_R A_1$, where $\rightarrow_P$ and $\rightarrow_R$ are program order and data race edges respectively, any A_i or A'_i is a node in the race graph, both A_i and A'_i are executed by the same thread, and A'_i and A_{i+1} access the same memory address for any $1 \leq i < n$. For the cycle in Figure 4(a), we have a sequence of nodes like $A_j \rightarrow_P A'_j \rightarrow_R A_{j+1}$ for any $1 \leq j < n$. For the cycle in Figure 4(b), we have a sequence of nodes like $A_j \rightarrow_R A_{j+1} \rightarrow_R A_{j+2} \rightarrow_P A'_{j+2}$ for some $1 \leq j < n-1$ such that A_j is a load, A_{j+1} is a store and A_{j+2} is a load access node. For the other values of j, the sequence is similar to the one found in the cycle of Figure 4(a).

We use a depth-limited version of depth-first search (DFS) algorithm to find cycles of different length from the race graph. Since the smallest SC violation cycle contains 4 nodes from two different threads, we start by finding cycles of length 4. DFS finds all possible cycles of length 4. We discard the ones that do not conform with the categories of Figure 4. After length 4, we find cycles of length 5, 6, ..., up to some maximum length.

At this point, it is worth mentioning why we choose to add all possible (i.e., transitive closure) program order edges between any pair of nodes in the race graph. It makes the cycle detection algorithm simpler in the sense that if we want to find SC violation cycles of length n, we just need to use DFS up to depth n. Without the transitive closure property, we need to search all possible depths.

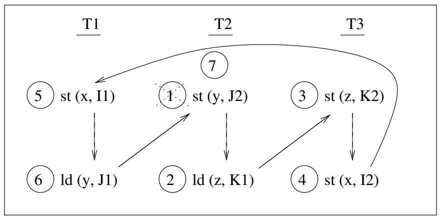


Figure 5. Determining the order of accesses to expose an SC violation.

5) Classifying TSO and PSO Cycles: The last step of Phase 1 is to classify the cycles according to the memory models i.e., TSO and PSO. For a cycle $A_1 \rightarrow_P A'_1 \rightarrow_R A_2 \rightarrow_P A'_2 ... A'_n \rightarrow_R A_1$ to cause an SC violation in TSO, A_i and A'_i have to be a store and load access node respectively for, at least, one instance of $1 \leq i < n$. Otherwise, the cycle cannot cause any SC violation in TSO. Any cycle that can cause an SC violation in TSO, can also cause an SC violation in PSO. In addition, the cycles where both A_i and A'_i are store access nodes for some $1 \leq i < n$ can also cause SC violations in PSO.

B. Phase 2: Exposing and Classifying SC Violations

This phase takes the list of potential SC violations found in Phase 1. It, then, exposes each violation and classifies it according to the execution outcome. If we cannot expose the violation, the violation is marked as infeasible.

1) Exposing SC Violations: Researchers have proposed many schemes to expose SC violations. One such scheme buffers a store as late as possible to expose SC violations [16]. Another scheme, Relaxer, buffers a store and then, speeds up or stalls certain thread to expose SC violations [17]. There is some work that relies on constraint solver [8] or explores the state space of a program to find SC violations [11]. In this work, we choose a different approach where, given an SC violation cycle, we expose it by enforcing the exact order required for the cycle.

We start by explaining with an example in Figure 5. If we consider TSO memory model, $st(x, I_1)$ followed by $ld(y, J_1)$ in thread T1 can get reordered. Similarly, $st(y, J_2)$ followed by $ld(z, K_1)$ in T2 can get reordered. However, the accesses of T3 cannot get reordered in TSO. Thus, in order to have an SC violation, we need to reorder at least one of the pairs - the access pair of T1 or T2. Without loss of generality, let us assume that $st(y, J_2)$ appears before $st(x, I_1)$ during the execution. Next, we assign an order in which different accesses of the cycle need to be executed to create the violation. We assign 1 for $st(x, J_2)$ and use topological sorting to assign a number to each access of the cycle. Note that topological numbers are correct for the accesses except the last one i.e., $ld(y, J_1)$. If we execute the accesses according to the number, $ld(y, J_1)$ will execute after $st(y, J_2)$. As a result, the (racing) dependence arrow will point downward and there will not be any SC violation. Instead, if we buffer the first access i.e., $st(y, J_2)$, execute the rest according to the assigned number and then, execute the buffered store $st(y, J_2)$ at the end, the cycle completes and an SC violation occurs. In a sense, the execution order for $st(y, J_2)$ changes from 1 to 7. Thus, Orion exposes an SC violation by reordering only one pair of accesses.

The complete algorithm for exposing an SC violation cycle in TSO is shown in Algorithm 1. At the high level, we fast forward (i.e., execute normally without enforcing memory model or any other constraint) until a thread is about to execute any access a_i in I (Line 5). Fast forwarding is a popular technique used in various processor simulators [27]. I contains accesses that can be bypassed by the next access in the cycle. So, for TSO and PSO, I contains only store accesses. The executing thread buffers a_i and assigns an execution order for the rest of the accesses of the cycle (Line 7). The threads execute them in that order and at the end a_i is flushed from the write buffer to expose the cycle. Any time a thread finishes its accesses of the cycle, it stalls until the cycle completes or certain amount of time (e.g., d seconds) has passed (Line 15). If a situation arises where the cycle may not occur, we restart (i.e., we apply the algorithm from the beginning on the rest of the execution). This occurs in Line 16, 31, 38, 41, 51, and 65. An example of such situations could be a wrong thread executing one of the accesses of the cycle (Line 35). Similarly, if a thread executes one of the already executed accesses (or addresses) of the cycle, we also restart (Line 38). For TSO, we keep buffering stores until the write buffer is full. When the buffer is full and we have a new store, we execute the earliest store from the buffer (Line 64). Thus, we create maximum reordering window. If a_i is flushed at any point before the cycle completes, we restart the algorithm (Line 51 and 65). Every load is executed right away (Line 54-58). This implies that even if the write buffer contains a store to the same address as a load, the load executes before the store. This behavior is allowed in TSO [3]. A fence always flushes the corresponding write buffer (Line 12).

The algorithm for PSO would be very similar to Algorithm 1 except if a_i is followed by another store in the cycle (i.e., we need to reorder a pair of stores), we keep a_i in the write buffer until we encounter a fence or the cycle is complete. If we need to flush the write buffer for any other reason, we will flush everything except a_i.

2) Classifying SC Violations: After exposing an SC violation, we let the program run to completion or failure. If a failure occurs, we classify the violation as a harmful one. We also classify the violation as a harmful one if the program runs to completion but the output is not correct. On the other hand, if the program runs to completion and the output is correct, we classify it as a benign one.

IV. Evaluation

A. Experimental Setup

We used Pin [19] to develop a profiler and a thread scheduler. The profiler was used for collecting execution

Algorithm 1 Code for exposing an SCV cycle in TSO

```
 1: Assume that the cycle to expose is $C = A_1 \rightarrow_P A'_1 ... \rightarrow_R A_1$
 2: Let $E$ and $A$ be the set of edges and accesses of C
 3: Calculate $S = \{A'_i | A'_i \rightarrow_R A_{i+1} \in E\ for\ 1 \le i < n\}$
 4: Calculate $I = \{A_i | A_i \rightarrow_P A'_i \in E\ and\ A'_i\ can\ bypass\ A_i\}$
 5: BEGIN: Fast forward until thread $T_i$ tries to execute $a_i \in I$.
 6: Buffer $a_i$ in $T_i$'s write buffer, $WB_{T_i}$
 7: Calculate execution order of $b \in A$ such that $order_{a_i} = 1$.
 8: Initialize $Addr = NIL$ and $Rest = A - \{a_i\}$
 9: $Turn = 2$ and $NextThread = T_i$
10: for any access $a_j$ or fence by thread $T_j$ do
11:     if The instruction is a fence then
12:         Flush $WB_{T_j}$
13:     else
14:         if $a_j \in Rest$ then
15:             Stall $T_j$ until $Turn = order_{a_j}$ or $d$ seconds have elapsed
16:             if $d$ seconds have elapsed then RESTART
17:             end if
18:             if $NextThread = ANY\ or\ NextThread = T_j$ then
19:                 FLUSH_ACCESS($a_j$, $T_j$)
20:                 $Turn = Turn + 1$
21:                 $Rest = Rest - \{a_j\}$
22:                 if $Turn = |A| + 1$ then
23:                     Flush $WB_{T_i}$
24:                     Flush other write buffers
25:                     Fast forward the rest of the execution
26:                 else if $a_j \in S$ then
27:                     $NextThread = ANY$
28:                     $Addr = addr_{a_j}$
29:                     Stall $T_j$ until the $a_i$ is flushed or $d$ seconds have elapsed
30:                 else if $a_j \in I$ then
31:                     $NextThread = T_j$
32:                     $Addr = NIL$
33:                 end if
34:             else
35:                 RESTART
36:             end if
37:         else if $a_j \in A - Rest$ then
38:             RESTART
39:         else
40:             if $addr_{a_j} = Addr$ then
41:                 RESTART
42:             else
43:                 ACCESS($a_j$, $T_j$)
44:             end if
45:         end if
46:     end if
47: end for
48: function FLUSH_ACCESS($a_j$, $T_j$)
49:     if $a_j$ is a store then
50:         Flush $WB_{T_j}$ and then, execute $a_j$
51:         if $a_i$ is flushed then RESTART
52:         end if
53:     else
54:         if $WB_{T_j}$ contains a store to $addr_{a_j}$ then
55:             Execute $a_j$ to return the value of the store
56:         else
57:             Execute $a_j$
58:         end if
59:     end if
60: end function
61: function ACCESS($a_j$, $T_j$)
62:     if $a_j$ is a store then
63:         if $WB_{T_j}$ is full then
64:             Remove the earliest store and execute it
65:             if $a_i$ is executed then RESTART
66:             end if
67:         end if
68:         Buffer $a_j$ into $WB_{T_j}$
69:     else
70:         if $WB_{T_j}$ contains a store to $addr_{a_j}$ then
71:             Execute $a_j$ to return the value of the store
72:         else
73:             Execute $a_j$
74:         end if
75:     end if
76: end function
77: function RESTART
78:     Flush all write buffers,
79:     Reset $Addr$, $Rest$, $Turn$ and $NextThread$
80:     Start from BEGIN
81: end function
```

Codes	Approx. LoC	# of racy var	# of racy ins	Phase 1						Phase 2						Relaxer
				# of dyn. accesses	# of edges	Size of graph(KB)	Initial cycles	Pruned cycles		WB 16		WB 32		WB 64		
								TSO	PSO	TSO	PSO	TSO	PSO	TSO	PSO	
harris	160	2	2	12	8	0.83	1	0	1	0	0	0	0	0	1	1
bakery	30	2	4	164	409	41.4	2556	1	0	1	0	1	0	1	0	1
init	65	2	7	20	96	9.75	534	1	1	1	1	1	1	1	1	2
snark	150	2	4	45	173	20.6	1655	0	3	0	1	0	1	0	1	1
lazylist	120	3	3	14	12	1.21	2	0	2	0	1	0	2	0	2	2
dekker	20	3	6	27	18	1.82	10	3	0	3	0	3	0	3	0	3
ms2	80	4	5	49	164	16.5	32958	0	2	0	0	0	1	0	1	1
pthread	66	4	7	23	102	10.3	4072	17	3	16	3	16	3	16	3	9
crypt	88	3	12	28	168	17	18549	29	10	19	8	19	8	19	8	24
msn	80	4	6	63	247	25.2	27659	3	5	1	0	1	2	1	2	1
httpd-2.4.20	228314	12	32	2250	92	9.95	0	-	-	-	-	-	-	-	-	-
bodytrack	14354	2	4	94	548	71.4	0	-	-	-	-	-	-	-	-	-
streamcluster*	1769	2	3	120	1971	229	1	-	-	-	-	-	-	-	-	-
raytrace	13841	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-
vips	142959	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-
canneal	2825	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-
sp2.raytrace	6050	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-
swaptions	1119	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-
fluidanimate	4343	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-
blackscholes	914	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-
dedup	3347	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-
Total	192380	37	70	659	3916	445.01	87997	54	27	41	14	41	18	41	19	45
								81		55		59		60		

Table II
DETAILED RESULTS OF DIFFERENT PHASES OF ORION.

traces and the scheduler was used to control thread interleaving. All experiments were performed on a 4-core system with Intel core i5-3570 3.0 GHz processor and 8 GB of RAM. The system had Ubuntu 12.04. We used three sets of benchmarks (Table III). The first set has implementations of concurrent data structures and mutual exclusion algorithms that have potential SC violations [8], [16]. The second set has some reported SC violation bugs from open source programs and libraries (e.g., MySQL, GCC). Finally, we used one application from SPLASH-2 [20], nine applications from PARSEC [21] and an open source server, Apache [28].

Set	Program	Description
Conc. Algo.	dekker	Algo. mutual exclusion.
	bakery	Algo. mutual exclusion.
	snark	Non-blocking double-end. queue.
	msn	Non-blocking queue.
	ms2	two-lock queue.
	harris	Non-blocking set.
	lazylist	List-based concurrent set.
Bug kernels	pthread_cancel from glibc	Unwind code after canceling thread needs a fence [29].
	crypt_util from glibc	Small table initialization code needs a fence [29].
	init from MySQL	Available charsets initialization code needs a fence [6].
Full Apps	SPLASH-2	1 programs form SPLASH-2.
	Parsec	9 programs form Parsec.
Open Src.	httpd-2.4.20	Apache HTTP Server.

Table III
APPLICATIONS ANALYZED.

For Phase 1 of Orion, we used Intel Inspector [26] to detect data races from each program. We recorded all Happened-before cycles up to length 8. Although we restricted ourselves up to length 8, the restriction is arbitrary. Programmers can use our tool to find any length cycles. For Phase 2, we tried to expose each cycle to create an SC violation. We ran 100 experiments for each cycle.

B. Characterization of Orion

Table II summarizes results for Phase 1 & 2. For each application listed, we show the number of unique racing variables and instructions detected by the data race detector in Column 3 and 4 respectively. Column 5 shows the total number of dynamic accesses in the profile and Column 6 shows the total number of edges (both program order and conflict edges) in the graph. The size of the graph is shown in Column 7. Column 8 reports the number of detected cycles. The cycles are pruned based on the criteria in Section III-A4 and classified into TSO and PSO cycles. They are reported in Column 9 and 10 respectively. Column 11 and 12, 13 and 14, 15 and 16 report the number of bugs exposed with write buffer of 16, 32, and 64 entries respectively. We compared our scheme against a state-of-the-art active testing technique, Relaxer [17]. It is shown in the last column. Before explaining the results, we should note that for streamcluster, we profiled only the first 20 accesses per racing variable per thread to limit the size of the profile and graph. Although this can cause false negatives, it is essential to work with large applications having sizable number of dynamic racing accesses.

In summary, Orion finds 81 Happened-before cycles from Phase 1 — 54 for TSO and 27 for PSO. Phase 2 can expose

55 SC violations with 16 entry write buffer. 41 out of 55 violations are for TSO and the rest are for PSO. With 32 and 64 entry write buffer, Phase 2 can expose 59 and 60 violations respectively. With increased buffer size, there is no change in the number of TSO violations that Orion can expose. However, PSO violations increase with write buffer size. This is expected since larger buffer allows more stores to get reordered in PSO memory model. We choose 64 as the default size of the write buffer. Compared to Orion, Relaxer can expose 45 SC violations out of 81 cycles. This is less than the violations exposed by Orion(which is 60). This is due to the fact that Relaxer cannot expose any of the 15 violations that have length greater than 4. It should be noted that the numbers shown here for Relaxer are much smaller than those reported in the original paper. This is because unlike the paper which counts all dynamic instances of SC violations, we count only the unique SC violations (each violation is identified by the addresses of the instructions involved).

C. Exposure Probability

Table IV describes empirical probability of confirming a cycle with different stalling time. We experimented with stalling time 0.5s, 1s, 2s, and 10s. Column 4 to 7 show the average empirical probability of exposing an SC violation cycle. The data shows that larger cycles usually have less probability of getting exposed. This is expected since all accesses need to be executed in a specific order to create the cycle. On average, with stalling time 2s, we achieved the highest exposure probability of 0.65 and we selected it as the default stalling time. With stalling time 0.5s, 1s, and 10s, the average exposure probability is 0.59, 0.58, and 0.47 respectively.

Codes	Cycle length	# of SCV	0.5s (p)	1s (p)	2s (p)	10s (p)
harris	4	1	0.43	0.4	0.36	0.31
bakery	4	1	1	1	1	1
init	4	2	0.89	0.94	0.91	0.96
snark	4	1	0.88	0.83	0.92	0.84
lazylist	4	2	0.58	0.56	0.51	0.55
dekker	4	3	0.68	0.74	0.67	0.66
ms2	4	1	0.1	0.17	0.17	0.09
pthread	4	9	0.72	0.73	0.9	0.75
	6	9	0.65	0.6	0.65	0.49
	8	1	0.08	0.03	0.02	0.02
crypt	4	24	0.46	0.53	0.56	0.28
	6	3	0.9	0.4	0.91	0.7
msn	4	1	0.37	0.39	0.31	0.32
	5	1	0.85	0.9	0.9	0.91
	6	1	0.37	0.17	0.12	0
Average	-	-	0.59	0.58	0.65	0.47

Table IV
EMPIRICAL PROBABILITY OF EXPOSING A CYCLE.

D. Time Analysis

Table V demonstrates average running time (seconds) of Phase 1 & 2. The table lists only 12 programs because the others have none or at most one data race (recall that for an SC violation, we need at least two data races). Phase 1 time is divided into profiling time, graph creation time, and cycle detection time. They are shown in Column 2, 3, and 5 respectively. Phase 2 time is the average cycle exposure time. It is shown in Column 6. If no cycle is found in a race graph for a given length, the corresponding exposure time is not shown. From the data, it is obvious that cycle detection is the most time consuming stage. Detection time increases for larger cycle length. For our experiments, we limited our cycle detection time to a maximum of 8 hours.

Codes	Profiling time (s)	Graph creation time (s)	Cycle length	Cycle detection time (s)	Exposure time (s)
harris	0.735	0.004	4	0.004	1.792
			5	0.005	-
bakery	0.741	0.035	4	66.404	3.487
			5	2777.116	-
init	0.638	0.007	4	0.359	2.472
			5	3.611	-
snark	0.726	0.009	4	1.181	13.698
			5	32.564	-
lazylist	0.703	0.004	4	0.005	1.684
			5	0.004	-
			6	0.004	-
			7	0.004	-
dekker	0.713	0.009	4	0.006	1.355
ms2	0.718	0.012	4	0.49	2.114
			5	5.85	-
			6	173.184	-
			7	5100.765	-
			8	28800*	-
pthread	0.648	0.006	4	0.151	1.843
			5	0.476	-
			6	4.08	2.2
			7	40.043	-
			8	355.437	2.948
crypt	0.647	0.01	4	1.092	4.268
			5	18.149	-
			6	439.572	4.9
			7	9157.144	-
msn	0.703	0.014	4	1.41	3.874
			5	35.629	10.813
			6	2623.907	3.695
			7	28800*	-
			8	28800*	-
httpd-2.4.20	1.192	0.017	4	0.07	-
			5	0.13	-
			6	0.19	-
			7	0.251	-
			8	0.324	-
body-track	10.947	0.035	4	1.439	-
			5	4.366	-
stream-cluster	5.714	0.087	4	42.85	-
			5	375.423	-

Table V
TIME ANALYSIS OF DIFFERENT PHASES.

For each stalling time, we ran a total of $81 \times 100 = 8100$ experiments to expose 81 cycles. We experimented with 4 different stalling times. Figure 6 summarizes the time required to execute these experiments. More than half of the experiments were completed within 2s in most cases. For example, 80.25%, 76.72%, 55.28% and 34.42% experiments were finished within 2s for stalling time of 0.5s, 1s, 2s, and 10s respectively. Average exposure time for those stalling times are 2.33s, 2.44s, 3.9s, and 13.6s respectively.

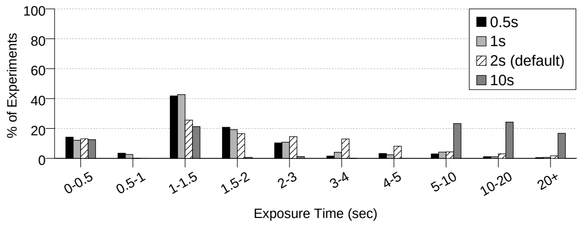


Figure 6. Exposure time analysis.

E. Bug Categorization

We observe SC violations' impact and classify them into 3 categories - harmful, benign, and infeasible SC violations. Recall that infeasible violations are the cycles which Orion cannot expose. We tested with two versions (according to the number of reordering enforced) of Phase 2 - 1 reordering (default) and all reordering. In 1 reordering version, we reordered exactly 1 access pair no matter how many pairs can be reordered in the cycle. In case of all reordering, we reordered all possible (allowed by the memory model constraints) access pairs. Overall, we found similar exposure probability for both versions - 0.65 and 0.61 for 1 and all reordering version (Figure 7). In addition, both versions classify SC violations into roughly the same fractions (i.e., 0.64 and 0.60 for the two versions respectively).

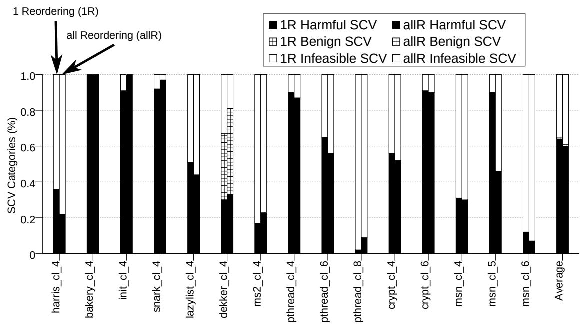


Figure 7. Bug impact.

F. Case Studies

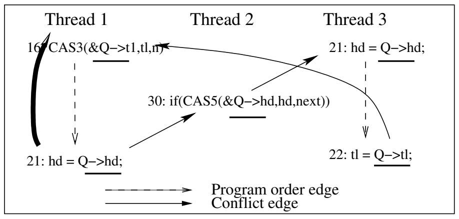
SC violation cycles can be categorized based on cycle length, memory model, and bug impact. We use examples to discuss each case.

1) Cycles of Different Length: Orion detected SC violation cycles of length 4, 5, 6, and 8. Figure 8 demonstrates a length-4 and length-8 cycle in pthread_cancel_init.c from glibc [29]. Here *f1*, *f2*, *f3* and *f4* corresponds to *libgcc_s_resume*, *libgcc_s_personality*, *libgcc_s_getcfa* and *libgcc_s_handle* respectively. Figure 8(a) presents the simplified code where each thread tries to initialize all four pointers if it finds *f4* to be null. Figure 8(b) shows a trace of thread 1 and thread 2. Under PSO memory model, the write of *f4* at line 7 can bypass both the write of *f2* and *f3* at line 5 and 6 in thread 1. Thread 2 first finds *f2* to be null at line 15 but inside init, it sees *f4* to be not null and returns. Then, at line 17 thread 2 tries to read *f2* which is still uninitialized. Figure 8(c) shows only the participating accesses in this violation with the execution sequence (5, 7, 2, 17, and 5). Figure 8(d) shows an SC violation of length 8 where the read of *f1*, *f2*, *f3* can bypass the write of *f2*, *f3*, *f4* in thread 1, 2, and 3 respectively under TSO. Under PSO, the write of *f4* can bypass the write of *f1* in thread 4. Any of these four reorderings can cause the violation. For example if read of *f1* bypasses write of *f2* then the execution sequence (5, 12, 6, 17, 7, 22, 4, 7, and 5) violates SC.

Figure 9 lists simplified enqueue and dequeue methods from benchmark msn in which Orion detected an SCV of length 5. Bypassing the write of $Q \rightarrow tl$ at line 16 by read of $Q \rightarrow hd$ at line 21 in thread 1 can start an SC violation cycle (16, 21, 30, 21, 22, and 16). Here, the single access in thread 2 is a write operation and both of its racy accesses are read. This is a necessary constraint for a violation of odd length. Figure 10(b) shows a violation of length 6 detected in crypt_util.c (Figure 10(a)). Details of this example is described in Section IV-F2.

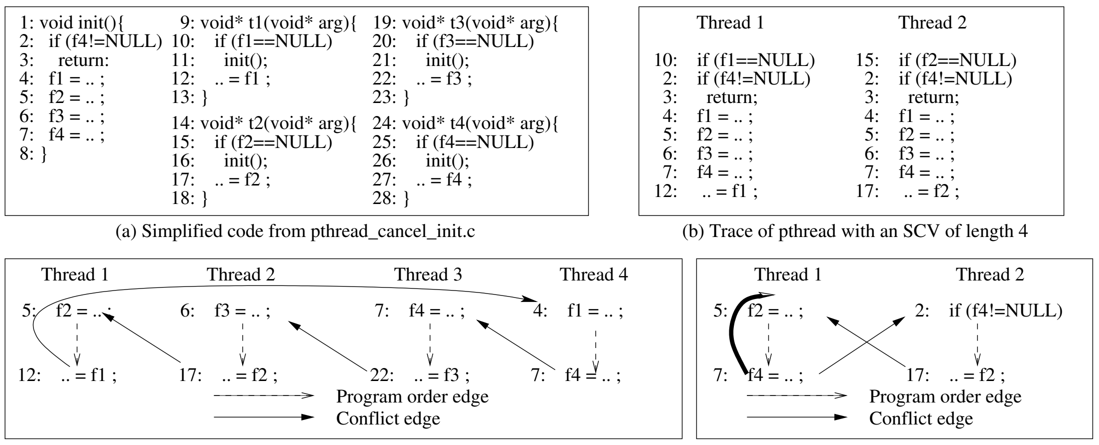
```
struct Q{
  node *hd, *tl;
 }

 1: void enqueue(Q, data){
 2:   node *n, *tl, *next;
 3:   n = new_node();
 4:   n->data = data;
 5:   n->next = NULL;
 6:   while(TRUE){
 7:     tl = Q->tl;
 8:     next = tl->next
 9:     if(tl==Q->tl)
10:       if(next==0){
11:         if(CAS1(&tl->next,next,n))
12:           break;
13:       }else
14:         CAS2(&Q->tl,tl,next);
15:   }
16:   CAS3(&Q->tl,tl,n)
17:}

struct node{
  node *next;
  void* data;
 }
18: void dequeue(Q){
19:   node *hd, *tl, *next;
20:   while(TRUE){
21:     hd = Q->hd;
22:     tl = Q->tl;
23:     next = Q->next;
24:     if(hd==Q->hd){
25:       if(hd==tl){
26:         if(next==0)
27:           return FALSE;
28:         CAS4(&Q->tl,tl,next);
29:       }else
30:         if(CAS5(&Q->hd,hd,next))
31:           break:
32:     }
33:   }
34:   free(hd);
35:   return TRUE;
```

(a) Core of msn benchmark



(b) Accesses that participate in length−5 SCV

Figure 9. Understanding the SC violation in msn.



```
1: void init(){          9: void* t1(void* arg){   19: void* t3(void* arg){
2:   if (f4!=NULL)       10:   if (f1==NULL)        20:   if (f3==NULL)
3:     return:           11:     init();            21:     init();
4:   f1 = .. ;           12:   .. = f1 ;            22:   .. = f3 ;
5:   f2 = .. ;           13: }                      23: }
6:   f3 = .. ;
7:   f4 = .. ;           14: void* t2(void* arg){   24: void* t4(void* arg){
8: }                     15:   if (f2==NULL)        25:   if (f4==NULL)
                         16:     init();            26:     init();
                         17:   .. = f2 ;            27:   .. = f4 ;
                         18: }                      28: }
```

(a) Simplified code from pthread_cancel_init.c

```
      Thread 1                 Thread 2

10:  if (f1==NULL)       15:  if (f2==NULL)
 2:  if (f4!=NULL)        2:  if (f4!=NULL)
 3:    return;            3:    return;
 4:  f1 = .. ;            4:  f1 = .. ;
 5:  f2 = .. ;            5:  f2 = .. ;
 6:  f3 = .. ;            6:  f3 = .. ;
 7:  f4 = .. ;            7:  f4 = .. ;
12:    .. = f1 ;         17:    .. = f2 ;
```

(b) Trace of pthread with an SCV of length 4

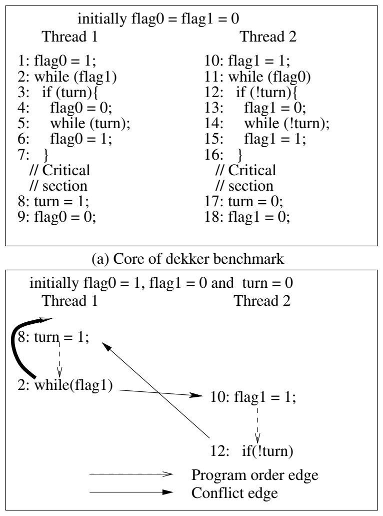
(d) Accesses that participate in the SCV of length 8 in pthread

(c) Accesses that participate in length−4 SCV

Figure 8. Understanding SC violations in pthread_cancel from glibc.

2) *Cycles of Different Memory Models:* An SC violation can occur due to write-write reordering, write-read reordering or both. If Orion detects a violation with write-write reordering only then we categorize it as an SC violation under PSO. Figure 8(c) shows an SC violation that is possible under PSO memory model due to the write-write reordering. On the other hand, the length 5 violation detected in msn (Figure 9(b)) is possible under TSO memory model due to the write-read reordering. If multiple reordering is possible in a violation where at least one of them is a read bypassing a write, then we categorize it as TSO since one (TSO allowed) reordering is enough to cause the bug. However, impact can vary according to the reordering. An example of such violation is shown in Figure 10(b). In thread 1, the write of *initialized* can bypass the write of *a_ptr* leaving *a_ptr* and *b_ptr* uninitialized. Thread 2 sees *initialized* set and tries to read *b_ptr* but finds it to be null. Note that bypassing the write of *b_ptr* by the read of *a_ptr* in thread 3 can also cause the violation, but impact will be different.

3) *Cycles with Different Outcomes:* We classify the exposed SC violations into harmful and benign ones according to the program outcome. Except for two SC violations from dekker, all other exposed SC violations are classified as harmful. For example, SC violations in Figure 8(c) and 10(b) cause the program to crash. A benign SC violation example from dekker is shown in Figure 11. Consider a potential Happened-before cycle (8, 2, 10, 12, and 8) in Figure 11(b) extracted from the code of dekker. Initially, thread 1 is in its critical section and thread 2 is waiting to enter. Thread 1 exits its critical section and then, attempts to re-enter. The write of 1 to *turn* at line 8 is buffered and the read of *turn* at line 12 gets an old value. This violation is benign because thread 2 waits (at line 14) for thread 1 to modify *turn* and thread 2 eventually sees the write of 1 to *turn* and enters into its critical section. More generally, we can see that there is no need to restrict the read of *flag1* at line 2 from bypassing the write of *turn* at line 8. The same is true for the read of *flag0* at line 11 and the write of *turn* at line 17. Note that, these benign SC violations are different from the ones detected in Relaxer [17].



```
        initially flag0 = flag1 = 0
    Thread 1                  Thread 2

1: flag0 = 1;             10: flag1 = 1;
2: while (flag1)          11: while (flag0)
3:   if (turn){           12:   if (!turn){
4:     flag0 = 0;         13:     flag1 = 0;
5:     while (turn);      14:     while (!turn);
6:     flag0 = 1;         15:     flag1 = 1;
7:   }                    16:   }
  // Critical               // Critical
  // section                // section
8: turn = 1;              17: turn = 0;
9: flag0 = 0;             18: flag1 = 0;
```

(a) Core of dekker benchmark

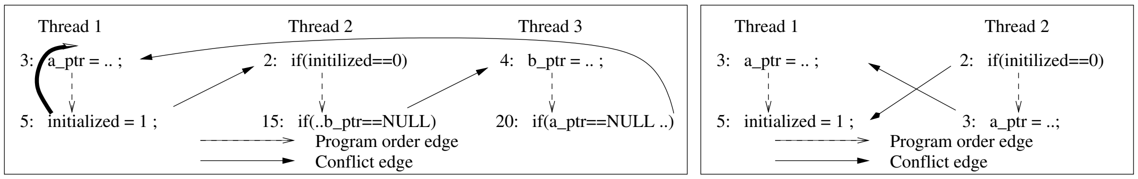
(b) Trace of dekker with a benign SCV

Figure 11. A benign SC violation in dekker.

4) *Infeasible Cycles:* All the racing edges in the graph are conservatively made bidirectional. However, in some cases, both directions are not possible during the actual execution. Phase 1 of Orion detected 21 cycles, each consisting of one infeasible conflict order. These false SC violation cycles are eventually discarded by Phase 2. Figure 10(c) demonstrates such a scenario. In the profile, the execution sequence is (3, 2, 5, and 3). Since we make the racing edge between 2 and 5 bidirectional, Phase 1 detects a cycle (3, 5, 2, and

```
1: void init(){                8: void* t1(void* arg){                  13: void* t2(void* arg){                  18: void* t3(void* arg){
2:   if (initialized==0){       9:   init();                             14:   init();                             19:   init();
3:     a_ptr = .. ;            10:   if(a_ptr==NULL || b_ptr==NULL)      15:   if(a_ptr==NULL || b_ptr==NULL )     20:   if(a_ptr==NULL || b_ptr==NULL)
4:     b_ptr = .. ;            11:     Error ;                           16:     Error ;                           21:     Error ;
5:     initialized = 1 ;       12: }                                     17: }                                     22: }
6:   }
7: }
```

(a) Simplified code from crypt_util.c



(b) Accesses that participate in an SC violation of length 6 (c) SC violation is not possible

Figure 10. Understanding the SC violation in crypt.

3). During Phase 2, thread 1 sets initialized to 1 at line 5, thread 2 reads 1 from it at line 2, and never executes the write of *a_ptr* at line 3 leaving the cycle incomplete. Other unexposed cycles are also infeasible due to the same issue of bidirectional edge. The infeasible cycles are eliminated after Phase 2.

V. Related Work

Most of the software based techniques are either exhaustive search based or constraint solver based approaches [8], [9], [30]. There are some dynamic approaches based on data race detection [17], [31]. Atig et al. [32] proved several decidability results for verification of finite state concurrent programs under different relaxed memory models. Exhaustive and constraint solver based approaches work well for small programs and kernels. However, they cannot handle large applications with many shared memory accesses. Orion can be applied to large applications by restricting the number of dynamic accesses to profile. Orion applies active testing [24], [33] technique like the earlier work [17], [34] to predict and expose SC violations in parallel programs. For example, Racefuzzer [24] combines race detection with a randomized thread scheduler in order to find real race conditions in a concurrent program with high probability and to discover if the detected real races could cause an exception or an error in the program. Bensalem et al. [35] used a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. Other proposals [36], [37] also use active testing to confirm potential bugs in parallel programs.

Run time monitoring algorithms such as Sober [15] and [16] scale moderately in practice. They rely on a model checker in order to find all violations of sequential consistency. They check SC executions to find SC violations in close enough relaxed executions. However, unlike Orion, they do not enforce the exact order to expose a violation; rather, those approaches rely on the worst case reordering to find violations. Several proposals [15], [32] including Relaxer have used operational definitions for TSO, PSO and other relaxed memory models. Orion also uses a conservative operational model along with some exact scheduling order to expose a violation.

Recent concurrency bug fixing scheme like CFix [38] automatically inserts synchronization operations to enforce the desired orderings and mutual-exclusions. Joshi et al. [39] proposed a property driven technique that introduces reorder-bounded exploration to identify the smallest number of program locations for fence placement. Any specialized algorithms to automatically insert fences based on static analysis [22], [40], [41] can guarantee memory-safety in principle. However, doubts remain about their precision in the presence of aliasing and loops. Besides, performance also degrades due to conservative fence insertion.

Most hardware based approaches [4], [42]–[47] detect data races as proxies for SC violations. Recent proposals [29], [48], [49] focus on detecting actual violations. These schemes are not suitable for real machines.

VI. Conclusion

An SC violation is almost always a bug. This paper proposed Orion, the *first* active testing technique that can detect, expose, and classify any arbitrary SC violations in any program. Orion works in two phases. In Phase 1, it finds potential cycles by focusing on racing accesses. In Phase 2, it exposes each cycle by enforcing the exact scheduling order. We presented a detailed design of Orion. We detected and exposed 60 SC violations of which 15 involve more than two processors and variables. Compared to a state-of-the-art active testing technique, Orion has a much better SC violations detection ability.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We also thank the members of PALab for the discussions and comments. This work is supported by the startup package provided by University of Texas at San Antonio and NSF under Grant No. 1319983.

References

[1] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computer*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[2] D. Weaver and T. Germond, *The SPARC Architecture Manual Version 9*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[3] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[4] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFx: a simple and efficient memory model for concurrent programming languages," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010.

[5] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1785414.1785443

[6] M. Islam and A. Muzahid, "Characterizing real world bugs causing sequential consistency violations," in *Workshop on Hot Topics in Parallelism*, June 2013.

[7] "Intel Cilk Plus," http://cilkplus.org/.

[8] S. Burckhardt, R. Alur, and M. M. K. Martin, "Checkfence: checking consistency of concurrent data types on relaxed memory models," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250737

[9] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind, "Nemos: a framework for axiomatic and executable specifications of memory consistency models," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.

[10] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, *QB or Not QB: An Efficient Execution Verification Tool for Memory Orderings*. Springer Berlin Heidelberg, 2004, pp. 401–413.

[11] T. Q. Huynh and A. Roychoudhury, "Memory model sensitive bytecode verification," *Formal Methods in System Design*, vol. 31, no. 3, pp. 281–305, 2007.

[12] S. Park and D. L. Dill, "An executable specification, analyzer and verifier for rmo (relaxed memory order)," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995, pp. 34–41. [Online]. Available: http://doi.acm.org/10.1145/215399.215413

[13] D. L. Dill, S. Park, and A. G. Nowatzyk, "Formal specification of abstract memory models," in *Proceedings of the 1993 Symposium on Research on Integrated Systems*. Cambridge, MA, USA: MIT Press, 1993, pp. 38–52. [Online]. Available: http://dl.acm.org/citation.cfm?id=163429.163442

[14] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, 2014.

[15] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV*, Jul 2008.

[16] J. Burnim, K. Sen, and C. Stergiou, "Sound and complete monitoring of sequential consistency for relaxed memory models," in *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, 2011, pp. 11–25.

[17] ——, "Testing concurrent programs on relaxed memory models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[20] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[22] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 2, pp. 282–312, Apr. 1988. [Online]. Available: http://doi.acm.org/10.1145/42190.42277

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, 1997.

[24] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[25] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[26] Intel, "Intel parallel studio," https://software.intel.com/en-us/intel-parallel-studio-xe, 2015.

[27] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[28] "Apache Web Server," http://www.apache.org/.

[29] A. Muzahid, S. Qi, and J. Torrellas, "Vulcan: Hardware support for detecting sequential consistency violations dynamically," in *Proceedings of the 45th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO '12, December 2012.

[30] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, "Qb or not qb: An efficient execution verification tool for memory orderings," in *In Computer-Aided Verification (CAV)*, 2004.

[31] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew, "Detecting and eliminating potential violations of sequential consistency for concurrent c/c++ programs," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[32] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," *SIGPLAN Not.*, vol. 45, no. 1, Jan. 2010.

[33] Z. Lai, S. Cheung, and W. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, May 2010.

[34] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient data race detection for distributed memory parallel programs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[35] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier, "Confirmation of deadlock potentials detected by runtime analysis," in *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2006.

[36] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," in *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, 2001.

[37] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," *SIGPLAN Not.*, Mar. 2009.

[38] G. Jin, W. Zhang, D. Deng, B. Liblit, , and S. Lu, "Automated concurrency-bug fixing," in *10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[39] S. Joshi and D. Kroening, "Property-driven fence insertion using reorder bounded model checking," in *Formal Methods (FM)*, ser. LNCS, vol. 9109. Springer, 2015, pp. 291–307.

[40] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence: A static analysis approach to automatic fence insertion," in *Computer Aided Verification (CAV)*, 2014.

[41] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[42] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ISCA*, 2009.

[43] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?" in *Proceedings of the 26th annual international symposium on Computer architecture*, 1999.

[44] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[45] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[46] K. Gharachorloo and P. B. Gibbons, "Detecting violations of sequential consistency," in *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, 1991.

[47] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[48] X. Qian, B. Sahelices, J. Torrellas, and D. Qian, "Volition: Precise and Scalable Sequential Consistency Violation Detection," in *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '13, March 2013.

[49] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient sequential consistency via conflict ordering," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.