Characterizing Real World Bugs Causing Sequential Consistency Violations

Mohammad Majharul Islam
vjw191@my.utsa.edu
University of Texas at San Antonio

Abdullah Muzahid
abdullah.muzahid@utsa.edu
University of Texas at San Antonio

Abstract

With the ubiquitous availability of parallel architectures, the burden falls on programmers' shoulders to write correct parallel programs that have high performance and portability across different systems. One of the major issues that complicates this task is the intricacies involved with the underlying memory consistency models. Sequential Consistency (SC) is the simplest and most intuitive memory model. Therefore, programmers usually assume SC for writing parallel programs. However, various concurrency bugs can lead to violations of SC. These subtle bugs make the program difficult to reason about and virtually always lead to incorrectness. This paper provides the *first* (to the best of our knowledge) comprehensive characteristics study of SC violation bugs that appear in real world codebases.

We have carefully examined pattern, manifestation, effect, and fix strategy of **20** SC violation bugs. These bugs have been selected from randomly chosen **127** concurrency bugs from a variety of open source programs and libraries (e.g. Mozilla, Apache, MySQL, Gcc, Cilk, Java, and Splash2). Our study reveals interesting findings and provides useful guidance for future research in this area.

1 Introduction

Over the last few years, we have seen parallel architectures to become ubiquitous. This burdens the programmers with the responsibility to write correct parallel programs. Therefore, the issue of programmability and correctness of parallel programs becomes one of the top priorities for today's computing world. A memory model directly affects programmability, performance, portability, and correctness of a parallel program. It defines a set of rules that specifies how the memory behaves with respect to read and write operations. It forms the fundamental basis for writing parallel code.

The memory model that programmers usually have in mind when they program and debug shared-memory parallel programs is SC. SC requires the memory operations of a program to appear in some global sequential order that is consistent with each thread's program order [9]. In practice, however, current processor hardware aggressively overlaps, pipelines, and reorders the memory accesses of a thread. As a result, a program's execution can be very unintuitive.

As an example, consider the simple case of Figure 1(a). In the example, processor *PA* allocates a variable and then sets a flag. Later, *PB* tests the flag and uses the variable. While this interleaving produced expected results, the interleaving in Figure 1(b) did not. Here, since the variable and the flag have no data dependence, the *PA* hardware reorders the statements. In this unlucky interleaving, *PB* ends up using uninitialized data. This is an SC violation.

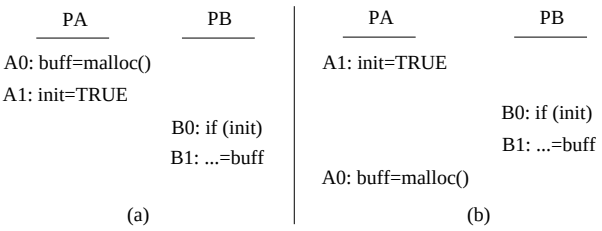


Figure 1: Example of an SC violation.

SC violation bugs are arguably the hardest type of concurrency bugs. First, these bugs complicate the reasoning process of a parallel program. When an SC violation occurs, the program execution loses its intuitive semantic meaning. As a result, the execution becomes harder to verify. Second, SC violation bugs are often architecture dependent. They might manifest under one memory model while completely invisible under a different memory model. Therefore, theses bugs directly affect the portability of a program. Third, often times, these bugs are found in codes that are known to have intentional data races. Examples of such codebases are Dou-

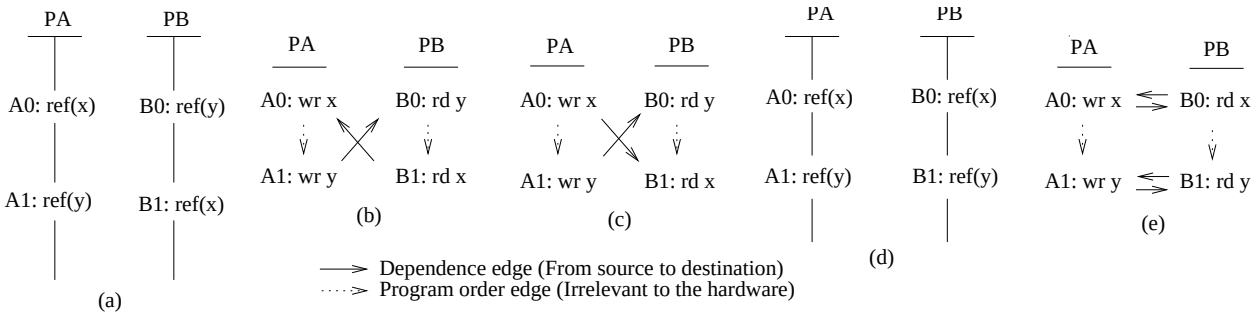


Figure 2: Understanding SC violations.

ble Checked Lock (DCL) constructs [19], synchronization and concurrent libraries, and codes that implement lock-free data structures. Therefore, traditional data race detectors cannot detect SC violation bugs from these codebases. Fourth, the presence of SC violation bugs can make various software debugging tools inadequate. These tools usually assume SC behavior of a program. Therefore, if a program has SC violation bugs, the tools may loose their applicability. This makes SC violation bugs an important category of concurrency bugs, even for the sake of detection of other bugs. Last but not the least, SC violation bugs require very specific reordering of memory accesses to manifest. Therefore, they cannot be reproduced with existing debugging and testing techniques.

Despite the importance and difficulty of SC violation bugs, there have been very few attempts to understand the characteristics of these bugs in real world programs and libraries. Most of the previous work [3, 1] focuses on SC violation bugs in small kernels and programs. Vulcan [16] provides a preliminary study of SC violation bugs in various open source programs but the study is far from comprehensive, considering the many dimensions of the bugs. This work performs the *first* (to the best of our knowledge) comprehensive characteristics study of SC violation bugs found in various popular codebases. More specifically, we examine bug pattern, effect, fix strategy, and other characteristics of real world SC violation bugs. Our study is based on **20** SC violation bugs out of a total of **127** randomly chosen concurrency bugs. These bugs are selected from Mozilla, Apache, MySQL, Gcc, Cilk, Java, and Splash2 programs. Our study reveals interesting findings which can act as guidelines for future research in this area.

This paper is organized as follows: Section 2 describes some background and related work; Section 3 explains the methodology of the study; Sections 4 describes the findings of our work; Section 5 explains the limitations of our study; and finally, Section 6 concludes the paper.

2 Background and Related Work

Shasha and Snir [20] show what causes an SC violation: overlapping data races where, at runtime, the dependences end up ordered in a cycle. Recall that a data race occurs when two threads access the same memory location 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 the dependences observed at runtime for SC violations (where we assigned reads and writes to the references arbitrarily).

If at least one of the dependences occur in the opposite direction (e.g., Figure 2(c)), no SC violations occur. In addition, if the code of the two threads references the two variables in the same order (Figure 2(d)), no SC violation is possible — no matter how these references end up being reordered at runtime. For example, in Figure 2(e), no SC violation can occur — no matter the execution order of the instructions within a thread, or the direction of the inter-thread dependences. All of these patterns can be generalized to include more than two threads and variables.

Given the pattern in Figure 2(a), Shasha and Snir [20] avoid the SC violations by placing a fence between the references *A0* and *A1* and another between the references *B0* and *B1*. Their algorithm to find where to put the fences has been called the Delay Set.

There have been several proposals [6, 14, 13, 16, 18] for detecting SC violations. However, most of them have focused on detecting data races as proxies for SC violations. This includes the work of Gharachorloo and Gibbons [6], DRFx [14], and Conflict Exceptions [13]. Specifically, they all look for a data race between two accesses from different threads that occur within a short time distance. The actual detection scheme varies among them. Recently, Vulcan [16] and Volition [18] focus on detecting actual SC violations by uncovering the required cycle as in Figure 2(a) and (b) among the memory accesses.

App	Bug Info	Brief Description
Gcc	2644	In *pthread_cancel_init()*, *libgcc_s_getcfa* is used as a flag without proper fences.
Gcc	11449	In *__init_des_r()*, *small_tables_initialized* is used as a flag without proper fences.
Gcc	10418	Fences are misplaced inside the code of *atomic_compare_and_exchange*, resulting in an incorrect implementation of pthread_mutex_unlock.
Gcc	55492	The implementation of *__atomic_load* places a fence before the load. This can cause later memory accesses to get reordered before the load.
Gcc	48076	The use of *emutls_key* can get reordered with *obj→loc.offset* because of unsafe DCL pattern.
Gcc	48126	Misplaced fences can cause later memory accesses to get reordered before *compare_and_swap* operation.
Mozilla	225525	The use of *decoding* in a pattern similar to DCL can lead to reordering.
Mozilla	622691	*defaultCompartmentIsLocked* is updated outside a critical section without using proper fences.
Mozilla	554860	*takeSample* is used as a flag without enough fences.
Mozilla	124923	The use of *oid_d_hash* in a pattern similar to DCL can lead to reordering.
MySQL	Duan et al.[4]	Clean up code may start before slave_running is cleared because of insufficient fences.
MySQL	Duan et al. [4]	The use of *cell→object* as a flag without proper fences can lead to reordering with prior memory accesses, increasing the possibility of a crash.
MySQL	45058	The use of *charset_initialized* in a pattern similar to DCL can lead to reordering.
Apache	49972	The use of *currentDateGenerated* in a pattern similar to DCL can lead to reordering.
Apache	49986	The use of *reload* in a pattern similar to DCL can lead to reordering.
Apache	47158	The use of *currentMillis* in a pattern similar to DCL can lead to reordering.
Apache	44178	*queue* is used as a flag without proper fences.
Java	6633229	The use of unsafe DCL pattern in *Math.random()* can lead to similar random values in two threads.
Cilk	Duan et al. [4]	The use of *store-store* fence instead of a full fence in *Cilk_unlock* can lead to reordering with prior load operations, resulting in an incorrect critical section.
fmm	Vulcan [16]	*construct_synch* is used as a flag without proper fences.

Table 1: Summary of SC violation bugs.

Some researchers have used the compiler to identify race pairs that can cause SC violations, typically using the Delay Set algorithm, and then insert fences to prevent cycles [4, 5, 8, 10, 11, 21]. There has been some work on program testing and verification that either checks semantic correctness or collects traces of a program and then, off-line, applies reordering rules to detect SC violations [1, 2, 3]. Such techniques are typically limited to small sized and intentionally buggy codes.

3 Methodology

3.1 Applications

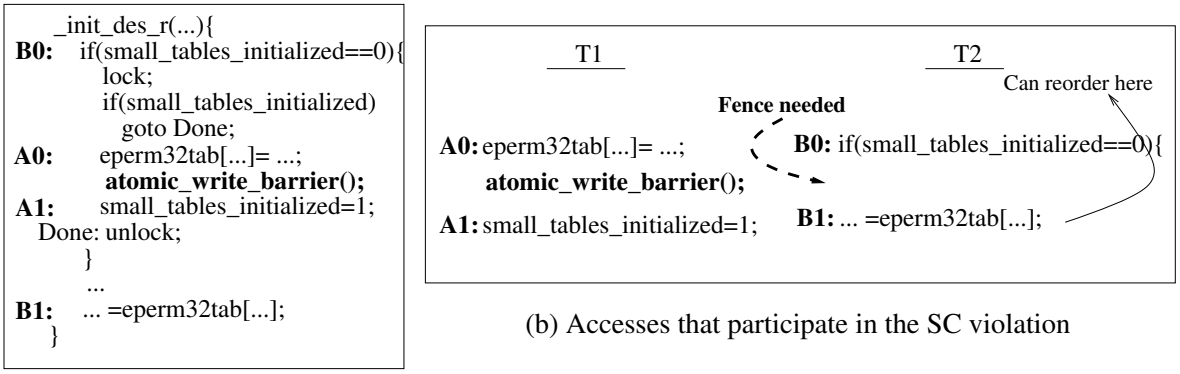
We select several open source applications, libraries and benchmarks in our study: Mozilla, Apache, MySQL, Gcc, Cilk, Java, and Splash2. These are all mature concurrent applications and libraries. They represent different type of server applications, client applications, programming libraries, threading libraries, and multithreaded benchmarks. Concurrency is used for different purposes in different programs. The server applications use from hundreds to thousands of threads to handle incoming requests. The client applications use concurrency to handle multiple GUI sessions and background worker threads. The libraries implement a variety of functionalities, a significant portion of which is meant to be thread-safe. The Splash2 benchmark contains different multithreaded scientific programs. We believe that these programs and libraries represent a variety of concurrent programs that are in use today.

3.2 Bugs

We collect SC violation bugs randomly from bug databases and previously published literature. The bug databases of the selected programs contain thousands of bug reports. In order to effectively collect SC violation bugs, we first search concurrency bugs using various keywords like 'double checked lock', 'synchronization', 'race', 'lock', 'concurrent', 'volatile', 'atomic', 'mutex', 'consistent', 'memory model, 'violation', and their variations. These keywords return several hundred bug reports. We discard the bug reports that have not been confirmed and fixed. So, the remaining bugs are clear-cut harmful ones. Among them, we randomly select 127 bugs. We manually check these bug reports, detailed discussions, relevant source codes, and patches to filter out the concurrency bugs that do not cause SC violations. After that, we finally get 20 bugs that can cause SC violations. It should be noted that not all of these bugs can cause SC violations in every memory model. We consider a bug to be an SC violation bug if it can cause SC violations at least in the most relaxed memory model (e.g. Release Consistency [7] or IBM PowerPC [15] model). Table 1 lists a summary of SC violation bugs that we have finally selected.

4 Findings

We analyze the bugs based on three main characteristics - pattern of the bugs, fix strategy, and time to fix. Besides these main issues, we also collect information about the



(a) Code from crypt_util.c

(b) Accesses that participate in the SC violation

Figure 3: Example of a DCL pattern.

number of threads and variables involved, impact of the bugs, and initial detection mechanism.

4.1 Bug Pattern

Different bug patterns require different diagnosis and fix approaches. We classify the bugs into three main patterns - DCL, improper flag synchronization, and incorrect fence. Figure 3 and 4 show a representative example of each pattern.

In Figure 3, an unsafe DCL pattern is used to initialize *eperm32tab*. If two threads call the initialization routine simultaneously, one thread (e.g. T2) might end up using uninitialized *eperm32tab* because of an SC violation caused by memory reordering.

In Figure 4(a), *construct_synch* has been used as a flag without declaring it as an *atomic* variable. This can cause prior memory accesses of thread T2 to get reordered after the flag is set. As a result, the computation can be incorrect. The third pattern is shown in Figure 4(b). Here, *cilk_membar_storestore* fence inside *Cilk_unlock* function cannot prevent the read operation of variable *b* to get reordered outside the critical section. This can lead to a violation of mutual exclusion property of critical sections.

Apps	DCL	Impro. Flag	Incor. Fence	Total
Gcc	2	1	3	6
Mozilla	2	1	1	4
MySQL	1	1	1	3
Apache	3	1	0	4
Java	1	0	0	1
Cilk	0	0	1	1
fmm	0	1	0	1
Overall	9	5	6	20

Table 2: Patterns of SC violation bugs.

Table 2 shows the number of different patterns found in different programs. According to this, DCL and improper flag synchronization patterns constitute 70% of all SC violation bugs. Therefore, if we target only these two patterns, the majority of SC violation bugs can be eliminated. Fortunately, these two patterns can be easily handled using some static analysis technique. The rest (i.e. 30%) require more advanced detection mechanism. Some type of model checking based approach [1, 3] or hardware based approach [16, 18] appears to be more appropriate for them.

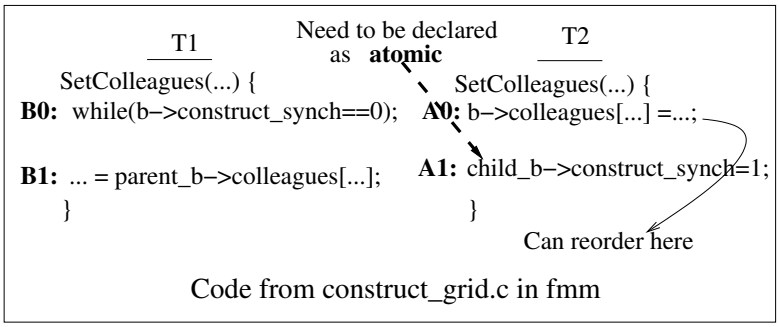
4.2 Fix Strategy

After carefully examining the fix strategies of all SC violation bugs, we classify them into four categories - correcting fences (Cor. Fence), using atomic variables (Atomic), using locks (Locks), and restructuring of the code (Res. Code). Figure 4(a) shows how an SC violation bug can be fixed by using an atomic variable. Figure 3(b) and 4(b) show how the bugs can be fixed by correcting a fence.

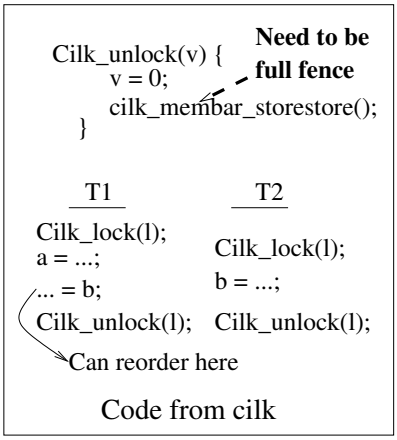
Apps	Cor. Fence	Atomic	Locks	Res. Code	Total
Gcc	5	0	1	0	6
Mozilla	1	1	1	1	4
MySQL	2	1	0	0	3
Apache	0	3	1	0	4
Java	0	0	1	0	1
Cilk	1	0	0	0	1
fmm	0	1	0	0	1
Overall	9	6	4	1	20

Table 3: Fix strategies of SC violation bugs.

Table 3 shows a break down of different fix strategies. It shows that 45% of SC violation bugs can be fixed by adding or modifying fences whereas 30% of the bugs require atomic variables. Together these two strategies fix 75% of all SC violation bugs. Rest of the bugs are fixed by adding locks or restructuring the code. This finding implies that the fix strategies of SC violation bugs are



(a) Improper flag synchronization



(b) Incorrect fence

Figure 4: Example of (a) an improper flag synchronization, and (b) an incorrect fence pattern.

quite different from those of data races (which are fixed mainly by adding or modifying locks). Hence, we need an approach that can determine when certain data races can lead to SC violations and suggest a fix accordingly.

4.3 Time to Fix

We collect information about how much time it takes to fix SC violation bugs. This gives us an idea about the complexity associated with these bugs. Since not all of the programs maintain official bug databases, we can gather this statistics only for 15 SC violation bugs. 7 of them require from 90 days up to several years to get fixed. This is in contrast to other concurrency bugs which require an average of 73 days to get fixed [12]. In fact, in one instance (Gcc bug #2644), the bug requires 5 years to finally get fixed. This gives us an idea about the complexity of analyzing SC violation bugs.

4.4 Other Characteristics

Number of threads and variables: All of the bugs that we selected, involve only two threads and two variables. Therefore, it suffices to have a detection mechanism that can handle SC violation bugs between any two threads. However, it is quite possible to have multiple SC violations between the same pair of threads.

Bug impacts: Whenever an SC violation occurs, a thread observes an undefined or stale value of a variable. This can eventually lead to dereferencing a NULL pointer, use of uninitialized data structures, incorrectness of a calculation, and even, violation of mutual exclusion property of a critical section. All of these affect the correctness of a program. All the bugs in our study have been confirmed as harmful by the developers.

Initial detection: Duan et al. [4] have detected three SC violation bugs. Among the other bugs, Mozilla bug #622691 has been initially detected by Helgrind [17] as a data race. The rest of the bugs have been initially detected by programmers reading the source code. This implies that programmers lack effective tools for detecting and reproducing SC violation bugs.

5 Limitations

Similar to previous work [12, 22], our characteristics study is subject to a validity problem. Potential threats to the validity of our work are representativeness of the programs and bugs, and our examination methodology.

As for application representativeness, our study chooses three open source applications, three open source libraries, and one multithreaded benchmark. We believe that these programs represent various concurrent applications and libraries used in today's computing world. However, our study may not reflect the characteristics of other types of applications like HPC applications, operating systems, or cloud applications.

To address bug representativeness, the SC violation bugs that we studied are randomly selected from the bug databases of the chosen applications or previously published work [4, 16]. Therefore, they provide a good sample of fixed bugs in those programs. However, the characteristics of non-fixed and non-reported SC violation bugs might be different.

In terms of examination methodology, we have gone through detailed bug reports, discussions, patches, and source codes. Since SC violation bugs are architecture dependent, it is possible that not all of the bugs might manifest in the same architecture. However, we believe that each of these bugs can manifest in at least one architecture.

Overall, while our findings cannot be generalized to all concurrent programs and libraries today, we believe

that our study captures the characteristics of SC violation bugs in major classes of current programs and libraries. Still, we would like to suggest the readers to take our findings with above methodology and applications in mind.

6 Conclusion

This paper presents the *first* (to the best of our knowledge) comprehensive characteristics study of SC violation bugs in large open source programs and libraries. Our study is based on **20** SC violation bugs out of randomly chosen **127** concurrency bugs. These bugs have been collected from three open source concurrent applications, three open source libraries, and one multithreaded benchmark. These bugs can serve as *a standard benchmark* for SC violations. Our study uncovers many interesting findings and implications of SC violation bugs. For example, DCLs and improper flag synchronizations are the major source of SC violation bugs. Therefore, some future research can be done by targeting only these two patterns. If we can provide some IDE plugin for writing these two patterns correctly or design some static analysis technique for them, then the majority of SC violation bugs can be avoided. Our findings also show that the fix strategies of SC violation bugs are quite different from those of data races. Therefore, we need an approach that can determine when some data races can lead to SC violations and suggest a fix accordingly.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable suggestions and feedback for this project. This work has been supported by the Faculty Start-up Fund from University of Texas at San Antonio.

References

[1] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. 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* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 12–21.

[2] BURCKHARDT, S., AND MUSUVATHI, M. Effective program verification for relaxed memory models. In *CAV* (Jul 2008).

[3] BURNIM, J., SEN, K., AND STERGIOU, C. 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* (Berlin, Heidelberg, 2011), TACAS'11/ETAPS'11, Springer-Verlag, pp. 11–25.

[4] DUAN, Y., FENG, X., WANG, L., ZHANG, C., AND YEW, P.-C. 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* (Washington, DC, USA, 2009), CGO '09, IEEE Computer Society, pp. 25–34.

[5] FANG, X., LEE, J., AND MIDKIFF, S. P. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 285–294.

[6] GHARACHORLOO, K., AND GIBBONS, P. B. Detecting violations of sequential consistency. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 1991), SPAA '91, ACM, pp. 316–326.

[7] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture* (New York, NY, USA, 1990), ISCA '90, ACM, pp. 15–26.

[8] KRISHNAMURTHY, A., AND YELICK, K. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing 38*, 2 (Nov. 1996), 130–144.

[9] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer 28*, 9 (Sept. 1979), 690–691.

[10] LEE, J., AND PADUA, D. A. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computer 50*, 8 (Aug. 2001), 824–833.

[11] LIN, C., NAGARAJAN, V., AND GUPTA, R. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 295–306.

[12] LU, S., ET AL. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS* (Mar 2008).

[13] LUCIA, B., CEZE, L., STRAUSS, K., QADEER, S., AND BOEHM, H.-J. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 210–221.

[14] MARINO, D., SINGH, A., MILLSTEIN, T., MUSUVATHI, M., AND NARAYANASAMY, S. 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* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 351–362.

[15] MAY, C., SILHA, E., SIMPSON, R., AND WARREN, H. *The PowerPC Architecture: A Specication for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc, 1994.

[16] MUZAHID, A., QI, S., AND TORRELLAS, J. Vulcan: Hardware support for detecting sequential consistency violations in programs dynamically. In *Proceedings of the 45th annual ACM/IEEE international symposium on Microarchitecture* (December 2012), MICRO '12.

[17] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[18] QIAN, X., SAHELICES, B., TORRELLAS, J., AND QIAN, D. Volition: Precise and Scalable Sequential Consistency Violation Detection. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems* (March 2013), ASPLOS '13.

[19] SCHMIDT, D. C., AND HARRISON, T. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Conf. on Patt. Lang. of Prog.* (1996).

[20] SHASHA, D., AND SNIR, M. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems 10*, 2 (Apr. 1988), 282–312.

[21] SURA, Z., FANG, X., WONG, C.-L., MIDKIFF, S. P., LEE, J., AND PADUA, D. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 2–13.

[22] YUAN, D., PARK, S., AND ZHOU, Y. Characterizing logging practices in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 102–112.