BAVC: Classifying Benign Atomicity Violations via Machine Learning*

Qichang Chen[1, a, *, **], Zhanfang Chen[1, b], Zhuang Liu[1, c], Xin Feng[1, d], Zhengang Jiang[1, e], Liqiang Wang[2, f], Hongyi Ma[2, g], Ping Guo[2, h], Hao Qian[3, i]

[1]Institute of Computer and Information Technology, Changchun University of Science and Technology, Changchun, Jilin, China

[2]Dept. of Computer Science, University of Wyoming, Laramie, WY, USA

[3] Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

[a]chenqichang10@gmail.com, [b]jeffy2100@126.com, [c]lz1227@live.cn, [d]fengxin@cust.edu.cn, [e]jzg@cust.edu.cn, [f]wang@cs.uwyo.edu, [g]hma3@uwyo.edu, [h]pguo@uwyo.edu, [i]hqian@ksu.edu,

Keywords: Atomicity Violations, Concurrency Errors, Machine Learning, Software Testing, Program Analysis

Abstract. The reality of multi-core hardware has made concurrent programs pervasive. Unfortunately, writing correct concurrent programs is difficult. Atomicity violation, which is caused by concurrent executions unexpectedly violating the atomicity of a certain code region, is one of the most common concurrency errors. However, atomicity violation bugs are hard to find using traditional testing and debugging techniques. In this paper, we investigate an approach based on machine learning techniques (specifically decision tree and support vector machine (SVM)) for classifying the benign atomicity violations from the harmful ones. A benign atomicity violation is known not to affect the program's correctness even it happens. We formulate our problem as a supervised-learning problem and apply these two machine learning techniques to classify the atomicity violation report. Our experimental evaluation shows that the proposed method is effective in identifying the benign atomicity violation warnings.

Introduction

The reality of multi-core hardware has put us at a critical turning point in software development. In order for software applications to benefit from the continued exponential throughput advances in new processors, the applications will need to be well-written multi-threaded software programs. However, writing correct multi-threaded programs is inherently difficult because concurrency can introduce errors that do not exist in sequential programs. In particular, concurrent accesses to shared data must be properly synchronized. Otherwise, concurrency-related errors may happen.

An atomicity violation occurs when an interleaved execution of a set of code blocks (expected to be atomic) by multiple threads is not equivalent to any serial execution of the same code blocks. Figure 1 illustrates a harmful atomicity violation that happens when the two synchronization blocks in thread 2 can execute between the two synchronization blocks in thread 1.

```
                Case 1
thread-1                thread-2
deposit(int val){       deposit(int val){
   int tmp = bal;          int tmp = bal;
   tmp = tmp + val;        tmp = tmp + val;
   bal = tmp;              bal = tmp;
}                       }
```

```
                Case 2
thread-1                thread-2
deposit(int val){       deposit(int val){
   synchronized(o){        synchronized(o){
     int tmp = bal;          int tmp = bal;
     tmp = tmp + val;        tmp = tmp + val;
   }                       }
   synchronized(o){        synchronized(o){
     bal = tmp;              bal = tmp;
   }                       }
}                       }
```

Figure 1 A Java example demonstrating an atomicity violation

A benign warning is a real atomicity violation which does not violate the correctness of the program execution. To be more accurate, some programmer will expect that such kind of atomicity

Published by Atlantis Press, Paris, France.

violation (or rather thread interactions) to happen for some code blocks during execution. However, this subtle thing cannot be captured by our or any existing runtime atomicity violation detection. Thus it will also be reported as an atomicity violation warning. Even given an atomicity violation report, we still need to identify and verify it to be a true warning rather than a false positive or a benign warning by analyzing and reasoning over the program source code.

This may involve significant efforts since it demands that we need to be analyzing those code blocks which caused this atomicity violation and understand the programmer's intention and the complicated interactions among those multiple program components as well. It is very time-consuming to accurately classify these atomicity violation reports on a manual base. To address this limitation, we proposed a benign atomicity violation classifier(*Benign Atomicity Violation Classifier - BAVC*) based on the machines learning techniques for filtering out benign atomicity violations in multi-threaded Java programs and evaluated it on several benchmarks. The technique relieves the programmer from manually sorting out the false alarms in the reported atomicity violations as it does not require any human intervention after the report is generated. It is most useful when there are hundreds of atomicity violations reported for a program that is beyond the human's capability.

BAVC takes a set of structural and semantic properties that are extracted from the code blocks involved in an atomicity violation and outputs a classification label(*i.e.*, true positive, false positive, benign). Its input atomicity violation report is generated from our prior tool HAVE [3].

Benign Atomicity Violations

Figure 4 shows the four summarized various benign atomicity violation patterns that were collected from in our prior experiments. The intuition behind it is that many benign atomicity violations fall into only a few categories, that similar ones share some similar traits that can be generalized and identified. For example, the spin lock example shown in the bottom left graph of Figure 3 is a benign atomicity violation pattern that is not a concurrency error and will not affect the program's correctness. The other three examples shown in Figure 3 are also benign atomicity violation patterns that are known to be part of a correct concurrent program.

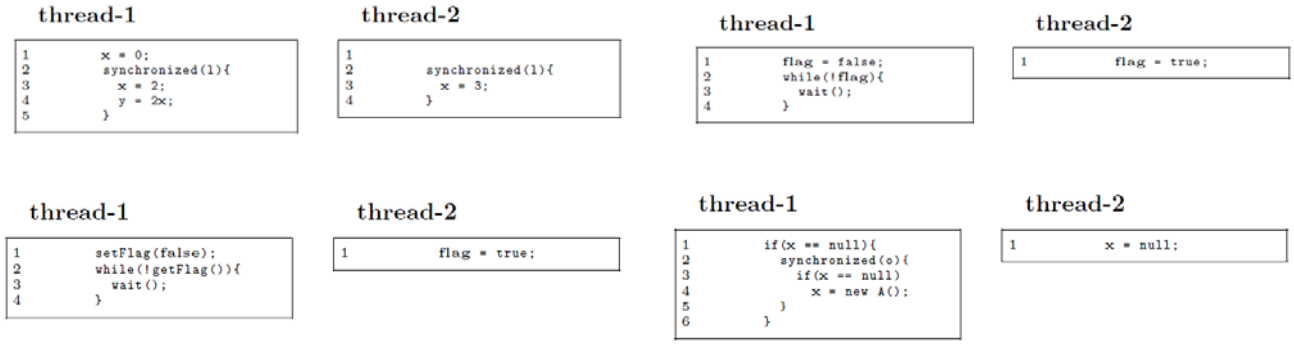

Figure 3 Four summarized benign atomicity violation patterns

Design of BAVC

Our approach consists of two steps: training and classification. The training stage involves generating the machine learning models from sample data set which consists of the four above-mentioned types of atomicity violations with correct labels. In the classification step, we apply the computed machine learning model to several atomicity violation reports which are obtained from benchmarks. Figure 3 illustrates the concept of BAVC and its workflow.

Published by Atlantis Press, Paris, France.

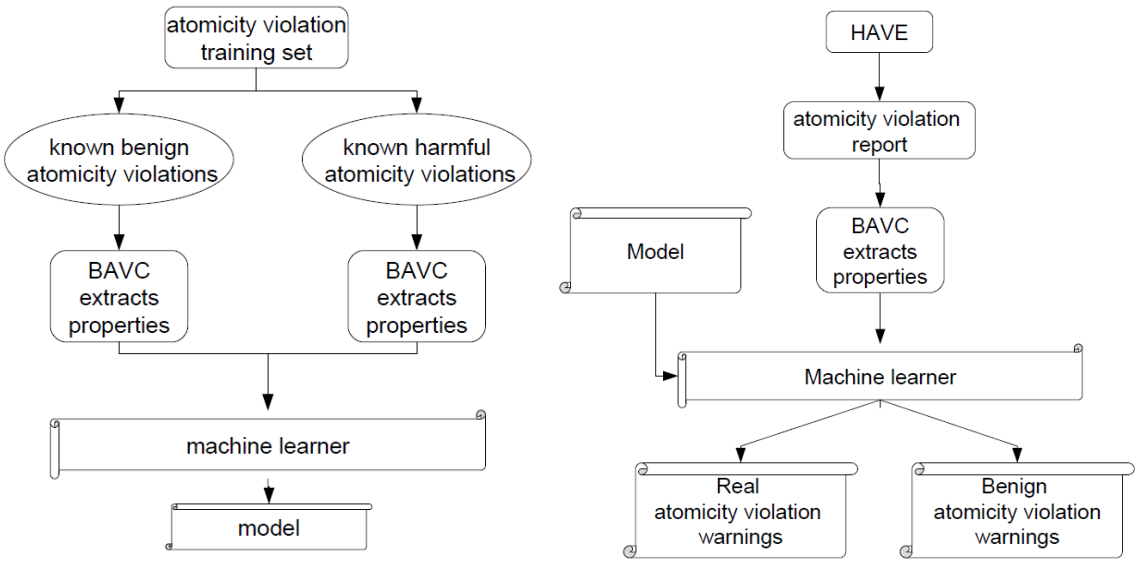

Figure 4 BAVC's machine learner training model in which we feed the known atomicity violations to the machine learners and BAVC's workflow

Discussions

We tested BAVC on the following programs: elevator, tsp, sor, and hedc from [15] and Vector, Stack, HashTable class source programs from JDK 1.4. The experiment is performed on a machine with 1.8 GHz Intel dual-core CPU, 2GiB memory, Windows XP SP3, and Sun JDK 1.6. Our experimental evaluation of BAVC indicates that it is very effective in identifying benign atomicity violations and both the decision tree and the SVM models adopted in BAVC are able to locate 12 out of the total 13 the benign atomicity violations (92.3% accuracy) that have been classified manually by the programmers in those benchmarks.

Related Work

There are many existing research papers about detecting the various concurrency errors including deadlocks, data races and atomicity violations. Chen et al. [5] presents a combined static and dynamic analysis, a new algorithm based on conflict-edges to detect and report atomicity violations. In [22], Wang et al. proposed the reduction- based and block-based algorithms. Flanagan and Freund [9] independently proposed a reduction-based algorithm. Xu et al. [23] proposed inferring computation units based on data dependence and control dependence, then atomicity is checked on the computation units [20]. Lu et al. [14] used access interleaving invariants as indications of programmers' assumptions about the atomicity of certain code regions. There are few prior research results about applying machine learning techniques with program analysis reports and our paper is innovative in this aspect.

Conclusion and Future Work

In this paper, we proposed and designed an innovative approach that applies the machine learning techniques to classifying the benign atomicity violations from harmful ones. Our experimental evaluation indicates this approach is very effective in identifying the benign ones and therefore very instrumental to the refining atomicity violation reports.

Directions for our future work include tuning the machine learning parameters to improve the overall accuracy, incorporating fine-grain transaction boundaries information in the machine learning models to reduce the false negatives and positives, extending our experiment to other concurrency benchmarks for thorough evaluation.

Published by Atlantis Press, Paris, France.

Acknowledgment

*Corresponding author.

**The research has been supported by Changchun University of Science and Technology New Faculty Startup Fund, NSF Grants 0941735 and CAREER-1054834.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM Press, Nov. 2005.

[2] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD). Springer-Verlag, Nov. 2005.

[3] Q. Chen, L. Wang, Z. Yang, and S. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In International Conference on Fundamental Approaches to Software Engineering FASE), European Joint Conferences on Theory and Practice of Software (ETAPS), York, UK, March 2009. Springer-Verlag.

[4] Q. Chen, L. Wang, and Z. Yang. Heat: A combined approach for thread escape analysis. International Journal of Systems Assurance Engineering and Management (IJSAEM) Special Issue on Advances in Software Testing, 1(1), 2011.

[5] Q. Chen, L. Wang, and Z. Yang. SAM: Self-adaptive Dynamic Analysis for Multithreaded Programs. In Haifa Verification Conference (HVC) 2011. Haifa, Israel. Dec. 2011. Springer-Verlag.

[6]. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In Proc. ACM Symposium on Principles of Programming Languages (POPL), pages 256{267. ACM Press, 2004.

[7]. C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. IEEE Transactions on Software Engineering, 31(4), Apr. 2005.

[8]. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In Proc. of ACM SIGPLAN conference on Programming language design and implementation (PLDI). ACM Press, 2008.

[9]. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press, 2003.

[10] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifcations for concurrent object-oriented software using model-checking. volume 2937 of LNCS. Springer-Verlag, Jan. 2004.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM Press, 2006.

[12] R. Majumdar and K. Sen. Hybrid concolic testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007.

[13] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press.

[14] Christoph von Praun and Thomas R. Gross. {Object race detection}. In Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), volume 36(11) of SIGPLAN Notices, pages 70–82. ACM Press, October 2001..

[15] C. von Praun and T. R. Gross. Object race detection. volume 36(11) of SIGPLAN Notices, pages 70-82. ACM Press, Oct. 2001.

[16] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press, June 2005.

[17] L.Wang and S. D. Stoller. Accurate and effcient runtime detection of atomicity errors in concurrent programs. In Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press, 2006.

[18] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. Transactions on Software Engineering, 32(2):93{110, Feb. 2006.

[19] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press, 2005.

Published by Atlantis Press, Paris, France.