An Integrated Framework for Checking Concurrency-related Programming Errors*

Qichang Chen and Liqiang Wang
Department of Computer Science
University of Wyoming
{qchen2, wang}@cs.uwyo.edu

Abstract

Developing concurrent programs is intrinsically difficult. They are subject to programming errors that are not present in traditional sequential programs. Our current work is to design and implement a hybrid approach that integrates static and dynamic analyses to check concurrency-related programming errors more accurately and efficiently. The experiments show that the hybrid approach is able to detect concurrency errors in unexecuted parts of the code compared to dynamic analysis, and produce fewer false alarms compared to static analysis. Our future work includes but is not limited to optimizing performance, improving accuracy, as well as locating and confirming concurrency errors.

1 Introduction

Multicore/multiprocessor hardware has become ubiquitous, which enforces concurrent programming to become a common technique. Although for the past decade we have witnessed incrementally more programmers writing concurrent programs, the vast majority of current applications are still sequential and can no longer benefit from the hardware improvement without significant redesign. In order for software applications to benefit from the continued exponential throughput advances in new architectures, the applications will need to be well-written concurrent software programs. However, developing concurrent programs is intrinsically hard due to the fact that concurrency introduces a whole new class of errors that do not exist in sequential programs.

Three of the most common concurrency errors are deadlock, data race and atomicity violation. Deadlock and data race are well-known and have been studied for a long time. A deadlock occurs when all threads are blocked, each waiting for some action by one of the other threads. A data race occurs when two concurrent threads perform conflicting accesses (*i.e.*, accesses to the same shared variable and at least one access is a write) and the threads use no explicit mechanism to prevent the accesses from being simultaneous. An example is shown in Figure 1, which is adopted from [3]. In Program 1 of Figure 1, conflicting accesses to the shared variable `bal` can happen simultaneously without any protecting lock, hence a data race occurs. Atomicity violation is not as well-known as deadlock and race condition. 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. Program 2 in Figure 1 eliminates the data race in Program 1 by adding a lock `o`. However, Program 2 is still incorrect if the `deposit` method is required to be atomic. An atomicity violation occurs in Program 2 when the two synchronization blocks in thread 2 execute between the two synchronization blocks in thread 1, which leads the result of `bal` to be incorrect.

```
                    Program 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;
}                          }
```

```
                    Program 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. Examples in Java demonstrating data races and atomicity violations.

Detecting concurrency-related software errors are based

*The work was supported in part by ONR Grant N000140910740.

on three main techniques of program analysis: dynamic analysis, static analysis, and model checking. Dynamic analysis reasons about behavior of a program through observations of its executions. To detect concurrency errors, dynamic analysis extends the traditional testing techniques. It tries to look for potential concurrency errors by searching specific patterns based on the current observed events, even the errors do not show up in the current execution paths [9, 6, 12, 11, 13, 8]. Static analysis makes predictions about the runtime behavior of a program by analyzing its source code [7, 10]. The strength of static analysis is that it can consider all possible behaviors of a program. However, it may produce false positives (*i.e.*, false alarms), because some aspects of a program's behavior, such as alias relationships, values of array indices, and happens-before relationships, are very difficult to analyze statically. Model checking is a formal method for proving that a finite-state model satisfies a temporal logic property, which can be used to check concurrency errors [5]. Although known as the most rigorous automatic method to verify software, model checking still faces a combinatorial blowup of the state space, commonly known as the state explosion problem.

Static and dynamic analyses can be combined in various ways. Static analysis can be used to reduce the overhead of dynamic analysis. For example, static analysis can show that some statements are not involved in any data races or atomicity violations and hence do not need to be instrumented; this can significantly reduce the overhead of dynamic analysis by up to a factor of 20 [1]. Conversely, dynamic analysis can help static analysis by providing more accurate runtime information.

In order to exploit the complementary benefits of different program analyses, we are designing a hybrid approach that integrates static, dynamic analysis, and model checking. Generally, we perform static analysis for program source code to generate a summary of the program. When an instrumented program runs, we collect the observed events together with the static summary to build abstracted tree structures, which are used for checking concurrency-related programming errors. To help programmer distinguish real bugs from other benign or false warnings, we use symbolic analysis together with a constraint solver to confirm the reported warnings.

2 Our Current Work

2.1 Integrated Dynamic and Static Analysis for Atomicity Violation Detection

We have implemented the hybrid approach in a tool called *Hybrid Atomicity Violation Explorer (HAVE)* for detecting atomicity violations in multi-threaded Java programs [3]. In HAVE, we first perform a conservative intraprocedural static analysis to generate a static summary tree SST for each method in the program. When the instrumented program runs, our runtime system tracks and records accesses to shared variables and reference variables, and builds tree structures. When we observe an unexecuted branch during dynamic analysis, the static summary of that unexplored branch is retrieved and instantiated using the runtime values. Thus, the instantiated summary speculatively approximates what would have happened if the branch had been executed. Finally, we check atomicity violations based on the hybrid tree structures, which contains both information from static analysis and dynamic analysis.

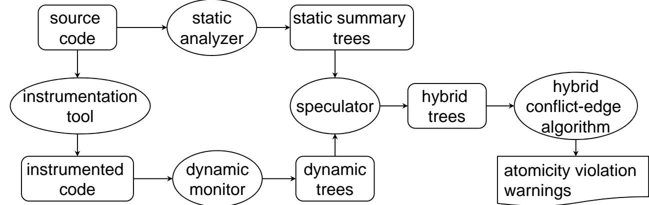

Figure 2. The architecture of the tool `HAVE`.

We have evaluated the tool HAVE on 9 benchmarks totaling 284 thousand lines of code which include large-scale web servers (Apache Tomcat and Jigsaw). We have discovered 13 bugs (non-atomic transaction) involving 145 locations in source code with HAVE in contrast with 11 bugs involving 90 locations in source code using our previous purely dynamic approach. The average slowdown is 16.5x, which is about 4 times as our previous purely dynamic approach [11]. Hence, the hybrid approach reports fewer false positives than the previous static approaches [7, 1], and fewer false negatives (*i.e.*, missed errors) than the previous dynamic approaches [6, 12, 11], at the sacrifice of performance.

Figure 2 shows the architecture of our tool, `HAVE`, which consists of five components.

2.2 Limitations of HAVE

The analysis of HAVE is incomplete and unsound.

First of all, no interprocedural analysis was applied during the stage of generating summaries. The summary itself is a condensed abstraction which is not a semantically equivalent to its source. Secondly, when the speculation is instantiated with the runtime information, some information could be missed since some symbolic names in that static summary might not be resolved, which leads to the incompleteness during post-stage analysis.

It is also not sound because our approaches could report false positives due to ignoring value information. For example, in the statements of "`if (x>0) then S1 else`

`S2; if (x<0) then S3 else S4;`", `S1` and `S3` are mutual exclusive (assuming that `x` is not modified in `S1` and `S2`), but our current approach can not eliminate it. In addition, HAVE is unable to verify that a warning is a real bug.

2.3 Thread Escape Analysis

In order to reduce the runtime overhead, we developed an integrated thread escape analysis that extends a dynamic escape analysis by incorporating static analysis. Thread escape analysis, a program analysis technique that determines which objects escape from their creating threads (*i.e.*, can be accessed by multiple threads), is important for subsequent program analyses.

Our hybrid approach works in two phases: in the first phase, it performs static analysis on program source code to obtain the concise static summaries; the second phase is a dynamic analysis: we monitor the actual field accesses during execution and perform an interprocedural synthesis on the runtime information and the static summaries to determine the escaped fields. Thus, the tool HAVE introduced in Section 2.1 will focus on the thread-shared fields only.

We implement our analysis for Java programs in a tool called HEAT (Hybrid Escape Analysis for Thread) [2] and evaluate it on several benchmarks and real-world applications. The experiment shows that the hybrid approach improves accuracy of escape analysis compared to existing approaches and significantly reduces overhead of subsequent program analyses on several benchmarks (in our experiment, specifically, a hybrid approach for checking atomicity violations). For example, many memory-intensive programs would take many hours to finish under previous dynamic or hybrid analysis because of overwhelming number of events generated from monitoring the field accesses. Our tool identifies more unshared fields, which in turn considerably trims down the number of monitored events and allows the dynamic or hybrid analysis to finish in a reasonable time.

3 Ongoing and Future Work

3.1 Interprocedural Speculation

We plan to extend our existing tool HAVE with interprocedural speculation. We perform an iterative context-sensitive interprocedural analysis on the static summary trees SST during speculation for different calling contexts.

When we encounter a method call during speculation, we expand it based on its definition in SST and substitute the formal parameters with actual arguments. This context-sensitive approach enables us to resolve object references inside the method body for invocations at different sites.

3.2 Reduce Runtime Overhead

Since the runtime overhead is usually huge in our hybrid approach as well as other dynamic analyses, reducing monitoring overhead will be critical to make the tool to be practical. We will develop heuristics to selectively turn off the monitoring after relevant paths and interleavings have been tested. An instrumented program may run slowly at the start because of monitoring and checking. As the program runs, our analyzer caches the monitored events. For a code block, if relevant paths and events have been observed, the monitoring on the code block can be turned off to speed up the execution. Thus, the monitoring on frequently-executed code will be disabled soon, and the monitoring on infrequently-executed code remains so long lurking bug may be detected. Because most overhead resides in frequently-executed portions, the runtime overhead of the instrumented program will remain at very low level after initial executions.

Specifically, we will explore the criterions for program state equivalence to identify redundant program states for monitoring thus reduces the overall runtime overhead. When our dynamic monitor observes a method call, it records the current program's state into a succinct summary with regard to key program state conditions (*e.g.*, the locks holding by the current thread, the calling context, the states of shared objects). So when a method call is invoked again under the same or a similar context, it is executed without monitoring. We will adopt some heuristics to help us set up the differentiation standards.

3.3 Fault Localization and Confirmation

To accurately locate and confirm programming faults, we need to check massive thread interleavings and feasible execution paths. The percentage of paths covered by dynamic analysis is usually small. For example, a program with 3 threads and 50 lines of code per thread may have more than 10^{69} different interleavings. We will use symbolic analysis techniques that implicitly analyze all possible thread interleavings under an execution. This may seem like an intractable task considering the fact that the number of interleavings is exponential. Advances in modern constraint solvers, however, suggest that this is quite feasible.

In general, accurately locating the faulty code requires a complete specification of the system behavior. Unfortunately, such specifications are often missing in realistic software development settings. Without a complete specification, it is not possible to determine whether a particular line in the code is faulty or not. We will model thread executions using suitable constraints and reduce the fault localization and confirmation problem to solve a set of constraints. Our symbolic analysis will be based on satisfiabil-

ity modulo theories (SMT), which benefit from recent significant advances in Boolean satisfiability (SAT) solvers and SMT solvers. Specifically, we resort to a state-of-the-art constraint solver Yices [4] to resolve the constraints.

In this approach, one needs to add a quadratic number of constraints (with respect to the total number of transitions). These constraints may pose a significant performance overhead for a SMT solver. We plan to use partial order reduction, which is based on a static analysis technique to find statically independent transitions, to reduce the number of constraints. In principle, we need only to add constraints to statements that are not statically independent.

4 Conclusions

This paper presents our existing, ongoing, and future work on a hybrid approach that integrates static, dynamic, and symbolic analyses to attack concurrency-related error detection.

Combining static and dynamic analyses is an active research area in program analysis. We are exploring different approaches to integrate them. In our existing hybrid approach, the summaries from static analysis are instantiate with runtime values during dynamic executions to speculatively approximate the behaviors of branches that are not taken. Compared to dynamic analysis, the hybrid approach is able to detect concurrency error in unexecuted parts of the code. Compared to static analysis, the hybrid approach produces fewer false alarms.

The future work includes but is not limited to developing a comprehensive approach to optimize performance, improve accuracy, locate and confirm concurrency errors.

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] Q. Chen, L. Wang, and Z. Yang. HEAT: A Combined Static and Dynamic Approach for Escape Analysis. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC2009)*, Seattle, USA, July 2009. IEEE Press.

[3] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Integrated dynamic and static analysis for atomicity violations. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 425–439. Springer, 2009.

[4] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[5] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2008.

[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 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.

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

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

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

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

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

[13] 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.