HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis*

Qichang Chen[1], Liqiang Wang[1]**, Zijiang Yang[2], and Scott D. Stoller[3]

[1] Dept. of Computer Science, University of Wyoming, WY, USA.
{qchen2, wang}@cs.uwyo.edu,
[2] Dept. of Computer Science, Western Michigan University, MI, USA.
zijiang.yang@wmich.edu
[3] Computer Science Dept., Stony Brook University, NY, USA.
stoller@cs.stonybrook.edu

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 concurrently executing code unexpectedly violating the atomicity of a code segment, is one of the most common concurrency errors. However, atomicity violations are hard to find using traditional testing and debugging techniques.

This paper presents a hybrid approach that integrates static and dynamic analyses to attack this problem. We first perform static analysis to obtain summaries of synchronizations and accesses to shared variables. The static summaries are then instantiated 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 atomicity violations in unexecuted parts of the code. Compared to static analysis, the hybrid approach produces fewer false alarms. We implemented this hybrid analysis in a tool called `HAVE` that detects atomicity violations in multi-threaded Java programs. Experiments on several benchmarks and real-world applications demonstrate promising results.

1 Introduction

Today, multi-core hardware has become ubiquitous, which puts us at a fundamental 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 programs. However, writing correct multi-threaded programs is difficult, because concurrency can introduce subtle errors that do not exist in sequential programs, if concurrent accesses to shared data are not properly synchronized.

* The work was supported in part by Wyoming NASA Space Grant Consortium, NASA Grant NNG05G165H, Wyoming NASA EPSCoR, NASA Grant NCC5-578, NSF CCF-0811287, CNS-0831298, CNS-0627447, CCF-0613913, and CNS-0509230.

** The corresponding author.

Two of the most common concurrency errors are data races and atomicity violations. 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. 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. 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.

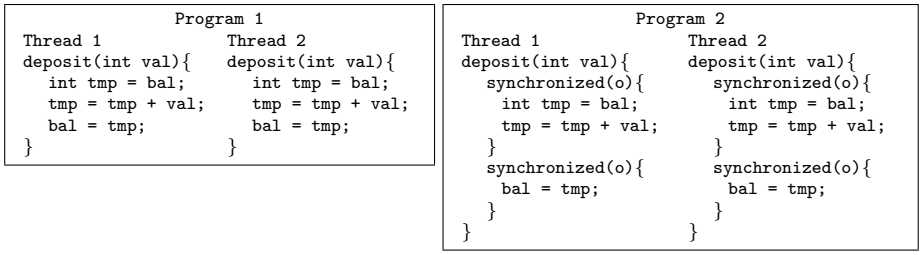

Fig. 1. Examples in Java demonstrating data races and atomicity violations.

Most existing approaches to detect atomicity violations are either purely dynamic (*e.g.* [6, 21, 20, 19]) or purely static (*e.g.* [9, 7]). 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. Moreover, many static analyses, such as the type system for atomicity in [9], require either manual annotation of the program or rewriting of the program into a special language. Dynamic analysis observes and analyzes the actual behaviors of a program by executing it. Generally, dynamic analysis is unsound compared to static analysis, because it does not analyze unobserved behaviors of programs. On the positive side, it generally produces much fewer false positives. Furthermore, dynamic analysis generally does not require manual annotation of code that is often required in static analysis; this is a significant practical advantage.

In order to exploit the complementary benefits of static and dynamic analyses, we propose a hybrid approach to detect atomicity violations. In our approach, we perform a conservative intraprocedural static analysis to generate a summary for each method in the program. Our runtime system tracks and records the values of reference variables during execution. When we observe an

unexecuted branch during dynamic analysis, the static summary of that unexplored branch is retrieved and instantiated using the recorded values. Thus, the instantiated summary speculatively approximates what would have happened if the branch had been executed.

We implemented the hybrid approach in a tool called *Hybrid Atomicity Violation Explorer (HAVE)* for detecting atomicity violations in multi-threaded Java programs and evaluated it on several benchmarks and real-world applications. The experiments show that the hybrid approach reports fewer false positives than the previous static approaches [9, 1], and fewer false negatives (*i.e.*, missed errors) than the previous dynamic approaches [6, 20, 19].

The rest of this paper is organized as follows. Section 2 formally defines atomicity violations. Section 3 presents the architecture of our tool HAVE. Section 4 introduces the conflict-edge algorithm. Section 5 describes some optimizations. Section 6 presents the experimental results. Section 7 reviews the related work. Section 8 discusses the conclusions and future work.

2 Atomicity Violations

An execution $\sigma = \langle s_1, \ldots, s_n \rangle$ is a sequence of accesses to shared variables, lock acquire, lock release, thread `start`, thread `join`, and `barrier` synchronization operations.

A *transactional unit* (or *transaction*) is an execution of a code block expected to be atomic. A *non-transactional unit* is an execution of a code block not expected to be atomic. For an event or transaction x, let $th(x)$ be the thread that performed x. As in [19], we assume that transaction boundaries are chosen so that thread start and join operations and barrier operations occur at transaction boundaries, not in the middle of transactions. Thus, thread and barrier operations induce a partial order on transactions: given an execution σ, and two transactional or non-transactional units u_1 and u_2, u_1 *happens-before* u_2, denoted $u_1 <_H u_2$, if (1) $th(u1) = th(u_2)$ and u_1 is executed before u_2, or (2) $th(u_1) \neq th(u_2)$ and (2a) $th(u_1)$ starts thread $th(u_2)$ after executing u_1 or (2b) $th(u_2)$ joins on $th(u_1)$ before executing u_2, or (3) $\exists u_i : (u_1 <_H u_i) \wedge (u_i <_H u_2)$. From monitoring an execution, we extract a set T of transactions, a set A of non-transactional units, and a happens-before relation $<_H$.

Given $\langle T, A, <_H \rangle$, *a trace of* $\langle T, A, <_H \rangle$ is an interleaving of events from units in $T \cup A$ that is consistent with the happens-before relation $<_H$ (*i.e.*, if $u_1 <_H u_2$, then all events in u_1 precede all events in u_2) and respects locking (*i.e.*, for every matching pair of acquire and release operations that belong to the same thread, no acquire or release of the same lock by other threads happens between them).

Traces π_1 and π_2 for $\langle T, A, <_H \rangle$ are *equivalent* if (1) they contain the same events, and (2) for each pair of conflicting accesses, the two accesses appear in the same order in both traces.

A trace of $\langle T, A, <_H \rangle$ is *serial* if the events of each transaction in T form a contiguous subsequence of the trace. $\langle T, A, <_H \rangle$ is *atomic* if every trace of $\langle T, A, <_H \rangle$ has an equivalent serial trace of $\langle T, A, <_H \rangle$.

For example, consider Program 2 in Figure 1. Suppose the method `deposit` is expected to be atomic. The program has only two serial executions, $[t_1.R(bal)\ t_1.W(bal)\ t_2.R(bal)\ t_2.W(bal)]$ and $[t_2.R(bal)\ t_2.W(bal)\ t_1.R(bal)\ t_1.W(bal)]$, where $t.A(x)$ denotes that thread t performs action A on variable x. The interleaved execution $[t_1.R(bal)\ t_2.R(bal)\ t_2.W(bal)\ t_1.W(bal)]$ is not equivalent to any serial trace, hence, the execution of method `deposit` is not atomic.

This notion of atomicity is also called *conflict atomicity* [19]. In [19], we also explored another notion of atomicity, called *view atomicity*. We do not consider view atomicity in this paper because checking it is more expensive and gives the same results as checking conflict atomicity in our experiments [19].

In this paper, we assume that the program does not have *potential for deadlock* (*i.e.*, some trace of the program may end in deadlock). This assumption is needed because a trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace. Potential for deadlock can be checked using our approach in [2].

3 Integrating Dynamic and Static Analyses

This section gives an overview of our hybrid approach to check atomicity violations. Figure 2 shows the architecture of our tool, `HAVE`, which consists of five components.

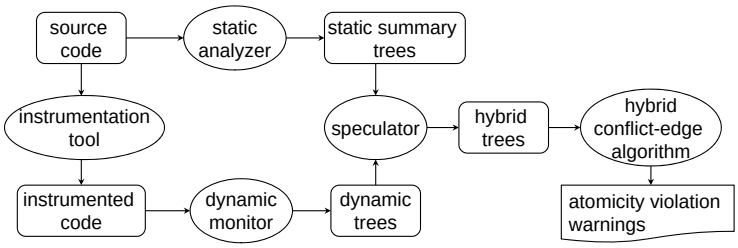

Fig. 2. The architecture of the tool `HAVE`.

1. A static analyzer, which parses the source code to generate *static summary trees (SSTs)*.
2. An instrumentation tool, which inserts event interception code.
3. A dynamic monitor, which intercepts events and builds *dynamic trees* during execution.
4. A speculator, which generates speculations for the unexecuted branches from SSTs and combines them with dynamic trees to form *hybrid trees*.
5. A detector, which analyzes the hybrid trees for atomicity violations using the hybrid conflict-edge algorithm described in Section 4.

3.1 The Static Analyzer

The static analyzer parses source code to construct static summary trees (SSTs). Each SST corresponds to a method in a certain class. Specifically, a static tree may contain nodes representing: (1) read/write to non-`final` and non-`volatile` fields; or (2) entrance and exit of synchronized blocks, including synchronized methods (which represent lock acquire and release operations); or (3) control-flow structures, namely, `if`, `for/while`, and `switch/case`; or (4) assignments to reference variables, which are used to speculate reference changes for the unexecuted code blocks. SSTs do not contain interprocedural information, *i.e.*, method calls are ignored. Unlike the dynamic monitor, the static analyzer ignores thread `start`, `join` and barrier synchronizations. Accesses to array elements are ignored in this paper due to the difficulty of statically resolving the indices of array elements. Figure 3 shows an example of a code block and its SST.

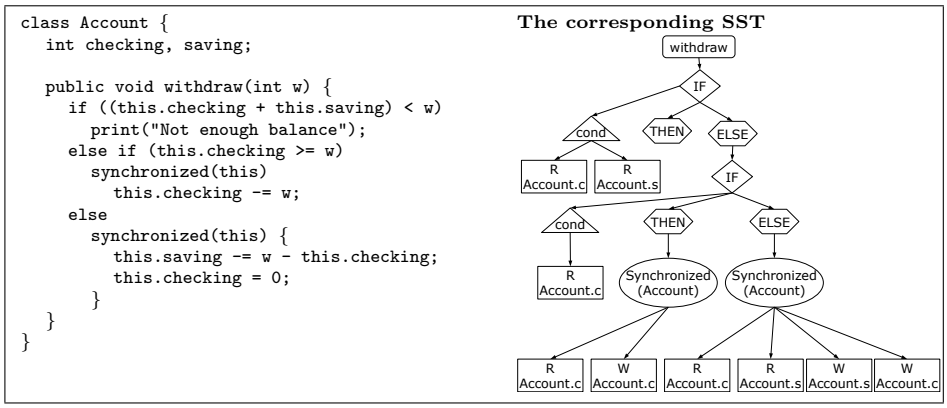

Fig. 3. An example of a static summary tree (SST), where `Account.c` and `Account.s` denote `Account.checking` and `Account.saving`, and "`R`" or "`W`" denotes that the node is a read or write, respectively.

3.2 Instrumentation

The instrumentation component instruments source code in order to intercept specific events during execution. The intercepted events include program control-flow structures, reads and writes to non-`final` and non-`volatile` fields, synchronization (including lock `acquire` and `release`, barrier operation, thread `start` and `join`), assignments to reference variables, and transaction boundaries.

Similar to [19], executions of the following code fragments are considered as transactions by default because their executions are often expected to be atomic

by programmers: non-private methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods. With exceptions, the executions of the `main()` method in which the program starts and the executions of `run()` methods of classes that implement `Runnable` are not considered as transactions, because these executions represent the entire executions of threads and are often not expected to be atomic. Moreover, `start, join` and barrier operations are treated as boundaries, *i.e.*, they separate the preceding events and following events into different units, and are not contained in any unit. We adopt this heuristic because execution fragments containing these operations are typically not atomic and hence are not expected to be transactions. The events not in transactions form non-transactional units. Note that for nested transactions, we check atomicity only for the outmost transactions, since they contain the inner transactions. The defaults can be overridden using a configuration file.

3.3 The Dynamic Monitor and Speculator

When an instrumented program runs, the dynamic monitor receives events issued by the instrumented code. The events of each unit (including transactional and non-transactional units) are stored in a structure called a *hybrid tree*, which consists of events observed in the execution and speculations based on static summary trees. Each leaf node represents a read or write to a shared variable and contains the runtime identifier for the shared variable. For example, `R(320.checking)` denotes a read to the field "`checking`" of an object identified by its hashcode `320`. Each non-leaf node except for the root represents a lock-based synchronization block or control-flow structure (*e.g.* `if/then/else`, `for/while` loop, `switch/case`). Each synchronization node contains the runtime identifier for the current lock (*i.e.*, synchronization object). The root node simply identifies the unit.

For each unexecuted branch in the unit, we instantiate the corresponding part of the method's SST by simulating its execution using the runtime context at the associated branch point, and add the resulting concrete events (*e.g.* synchronization nodes, reads and writes) to the hybrid trees. We instantiate symbolic names in the SST by querying binding tables. A binding table is maintained for each object; it stores the mappings between symbolic names and runtime values of all reference fields and local reference variables under the context of the object. A binding table is maintained for each class with static reference fields. Binding tables are updated when assignments to reference variables are executed. During speculative execution, assignments to reference variables in SSTs trigger updates on temporary copies of binding tables, instead of the original ones. Since there might be unresolved symbolic names left during speculation, the speculation may be not as accurate as its runtime equivalent observed in the dynamic analysis if it can be executed. This speculative technique may lead to false positives. Our experiments show that such kind of false positives are very rare in practice.

Our speculative execution also constructs subtrees corresponding to speculative iterations of loop bodies. According to Theorem 3 in Section 4.3, if all

iterations of a loop perform the same accesses, then at most two iterations are needed to detect atomicity violations. We use this as a heuristic, without attempting to verify the hypothesis of the theorem. Specifically, when the control flow reaches a loop, if the execution contains no iterations of the loop at that point, we add two speculative iterations; if the execution contains only one iteration of the loop at that point, we add one speculative iteration.

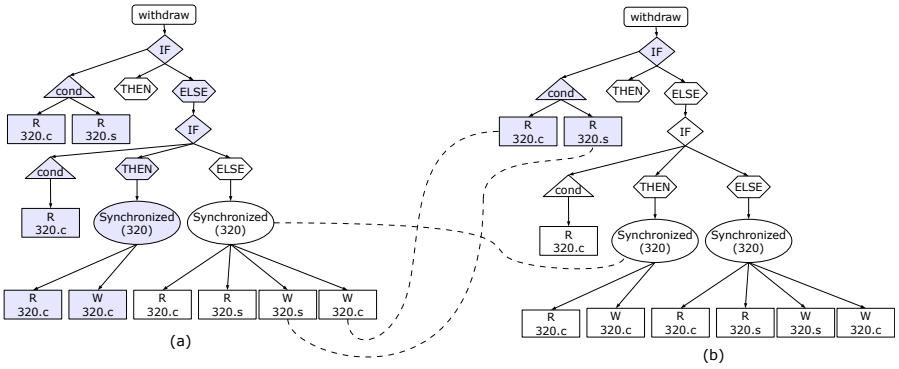

Fig. 4. An example of hybrid trees. Tree (a) corresponds to an execution of `withdraw` in Figure 3 when the `else if` is executed; tree (b) corresponds to the scenario when the first `then` is executed. The grey nodes are generated from the real executions; all the other nodes are speculated. The dotted lines denote conflict-edges introduced in Section 4. Only partial conflict-edges are marked out.

Two examples of hybrid trees are shown in Figure 4. Tree (a) and Tree (b) are generated by two threads of an execution that call the method `withdraw()` in Figure 3. The hashcode of the instance of `Account` is assumed to be `320`.

4 The Conflict-Edge Algorithm

This section presents the conflict-edge algorithm that detects atomicity violations based on hybrid trees. The algorithm adds edges, called *conflict-edges*, between hybrid trees, to connect two conflicting nodes (which is discussed in details in Section 4.1). The algorithm then generates all valid pairs of conflict-edges; informally, "valid" means that all nodes involved in the pair can coexist in some execution. The algorithm detects and reports atomicity violations by analyzing each valid conflict-edge pair. Note that the conflict-edge algorithm does not merely look for violations of atomicity in the observed execution, but also determines whether atomicity violations exist in feasible permutations of the observed execution.

4.1 Building Conflict-Edges Between Hybrid Trees

Two nodes n_1 and n_2 *conflict* if (1) they are in different hybrid trees, and (2) they represent accesses to the same variable and at least one of them is a write, and (3) thread `start`, `join`, and barrier operations do not induce a happens-before relation on them (*i.e.*, do not prevent them from occurring simultaneously). Let $held(n_i)$ denote the set of locks held when n_i is executed, which is determined by the synchronization nodes that are ancestors of n_i in the tree.

For each pair (n_1, n_2) of conflicting nodes, if $held(n_1) \cap held(n_2) = \emptyset$, we add a conflict-edge between n_1 and n_2; otherwise, we add a conflict-edge between the highest ancestors of n_1 and n_2 that are synchronization nodes for the same lock. The highest synchronization nodes represent the outmost common lock held during the executions of n_1 and n_2. The conflict-edge reflects the granularity at which the code blocks containing conflicting accesses can be interleaved. For example, Figure 4 shows partial conflict-edges between the two hybrid trees.

4.2 Detecting Atomicity violations

A hybrid tree t represents a set $[\![t]\!]$ of possible (transactional or non-transactional) execution units, corresponding to different choices of the branches of the `if`, `switch`, and loop statements that appear in it. For simplicity, our speculative analysis assumes that each branch could be taken, independently of other choices; in other words, the conditions guarding the branches are ignored. Given a set $T = \{t_1, \ldots, t_n\}$ of hybrid trees representing transactions, a set $A = \{a_1, \ldots, a_m\}$ of hybrid trees representing non-transactional units, and a happens-before relation $<_H$ on these trees, $\langle T, A, <_H \rangle$ is *atomic* if, for all $t'_1 \in [\![t_1]\!], \ldots, t'_n \in [\![t_n]\!], a'_1 \in [\![a_1]\!], \ldots, a'_m \in [\![a_m]\!]$, $\langle \{t'_1, \ldots, t'_n\}, \{a'_1, \ldots, a'_m\}, <'_H \rangle$ is atomic, where two execution units are related by $<'_H$ iff the hybrid trees they were generated from are related by $<_H$.

Conflict-edges e and e' are *incompatible* if one end node of e and one end node of e' appear in mutually exclusive branches of a hybrid tree, such as `then` and `else` branches of the same `if` statement, or different cases of a `switch` statement; otherwise, conflict-edges e and e' are *compatible*.

Conflict-edge e is an *ancestor* of conflict-edge e' in hybrid tree t if an endpoint of e is an ancestor of an endpoint of e' in t. A pair (e, e') of conflict-edges is *valid* for hybrid tree t, if (1) e and e' are compatible, (2) e is not an ancestor of e' in t, and vice versa, and (3) e and e' are incident on different nodes of t. In the rest of the paper, all pairs mentioned are valid by default if without explicit indication.

We have the following theorem to determine atomicity for a transactional hybrid tree. Let F_σ be a hybrid forest generated from an execution σ. Given a transactional hybrid tree t contained in F_σ, let $F_\sigma \setminus \{t\}$ be the set of all the other units.

Theorem 1. *Suppose a hybrid forest F_σ has no potential for deadlock. If t has no valid pair in F_π, $\langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$ is atomic.*

```
CheckAtomicityViolations() {
  AVScenarios := ∅;
  for each transactional hybrid tree t do
    for each valid conflict-edge pair (e, e′) of t do
      if only two hybrid trees including t are connected by e and e′ then
        /* find an atomicity violation scenario */
        AVScenarios := AVScenarios ∪{(e, e′)};
      else
        if ∃ a valid cycle c of conflict-edges involving (e, e′) then
          AVScenario := AVScenario ∪{c};
}
```

Fig. 5. The conflict-edge algorithm to detect atomicity violations

Proof Sketch: If t does not have valid pairs in F_π, for every trace of $\langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$, there are only two possible cases: (1) t has at most one node with an incident conflict-edge; or (2) t has a set S of nodes with incident conflict-edges, and for all $n_1, n_2 \in S$, n_1 is an ancestor or descendant of n_2. In the first case, for each non-serial trace, we can construct an equivalent serial trace by commuting all events in t to the position of that node. In the second case, we can construct an equivalent serial trace by commuting all events in t to the position of the lowest node in S. Hence, $\langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$ is atomic. □

Given a valid pair of conflict-edges (e, e'), a sequence of conflict-edges involving e and e' may form into a cycle, where two conflict-edges connected to the same tree are considered linked. A cycle involving (e, e') is *valid* if (1) neither e nor e' is an ancestor of the other; and (2) all conflict-edges on the cycle are compatible; and (3) for each node n on the cycle, $held(n) \cap held(n_e^t) = \emptyset \wedge held(n) \cap held(n_{e'}^t) = \emptyset$, where n_e^t and $n_{e'}^t$ are the end nodes of e and e' in t, respectively; and (4) all involved transactional and non-transactional units are concurrent (*i.e.*, the happens-before relation enforces no order on them). We have the following theorems to check atomicity violations.

Theorem 2. *Suppose a hybrid forest F_σ has no potential for deadlock. If a valid pair of conflict-edges (e, e') on a transactional hybrid tree t is involved in a valid cycle of F_σ, then t has an atomicity violation with the scenario indicated by the cycle.*

Proof Sketch: Suppose the valid cycle consists of conflict-edges $e_0, e_1, \ldots, e_n$, where $e_0 = e$ and $e_n = e'$. Let u_i and u_{i+1} be the execution units containing the endpoints of e_i, for $i = 0..n$. Note that $u_0 = t$ and $u_{n+1} = t$. The conditions in the definition of valid cycle imply that $u_0, \ldots, u_n$ are distinct and there exist $u'_0 \in [\![u_0]\!], \ldots, u'_n \in [\![u_n]\!]$ and an interleaving σ' for $\langle T', A', <'_H \rangle$ that contains events in the order $\langle endpoint(e_0, t), endpoint(e_0, u_1), endpoint(e_1, u_1), endpoint(e_1, u_2), \ldots, endpoint(e_n, u_n), endpoint(e_n, t) \rangle$ (*i.e.*, all of the other nodes on the cycle occur between the two nodes of t on the cycle), where T' and A' contain the transactional and non-transactional units, respectively, in $\{u'_0, \ldots, u'_n\}$, and $u'_i <'_H u'_j$ iff $u_i <_H u_j$. Because the two endpoints of a conflict-edge represent conflict-

ing accesses to a shared variable, all executions equivalent to σ must preserve the order of $t, u_1, \ldots, u_n, t$. Thus, there is no serial execution (in particular, no execution in which t occurs serially) equivalent to σ, so the cycle indicates an atomicity violation. □

Corollary 1. *Suppose a hybrid forest has no potential for deadlock. If a valid pair of conflict-edges on a transactional hybrid tree t is incident to only two transactions (including t), then t has an atomicity violation with the scenario indicated by the pair.*

Proof Sketch: The valid pair forms into a cycle. Thus, the conclusion is simply implied by Theorem 2. □

For each hybrid tree t, we detect atomicity violations by checking valid pairs of conflict-edges as shown by `CheckAtomicityViolations()` in Figure 5. Given a valid pair (e, e') of t, if e and e' involve only two hybrid trees, this pair implies an atomicity violation by Corollary 1. If e and e' involve three hybrid trees (recall that e and e' are already incident to t), we check atomicity violations based on Theorem 2. Our current implementation does not use Theorem 1, because our system looks for potential atomicity violations; it does not try to verify atomicity. Corollary 1 is applied first because it is cheaper.

For example, in Figure 4, an atomicity violation is revealed by a valid pair ($<$ `a.W(320.c)`, `b.R(320.c)` $>, <$ `a.W(320.s)`, `b.R(320.s)` $>$). The atomicity violation cannot be discovered by a purely dynamic approach because the valid pair connects a speculative branch in tree `a` with an executed branch in tree `b`.

Let S (mnemonic for Size of trees) denote the maximum number of nodes in any hybrid tree. Note that the number of trees is $|T \cup A|$. The worst-case time complexity of constructing conflict-edges is $O((|T \cup A| \times S)^2)$. Let n_c denote the maximum number of conflict-edges incident on any hybrid tree. Usually n_c is much less than $|T \cup A| \times S^2$. Theorem 2 requires finding a valid cycle, which is $O((|T \cup A| \times n_c)^2)$. The total number of valid pairs is $O(T \times n_c^2)$. Hence, the worst-case time complexity of checking atomicity violations is $O(|T| \times |T \cup A|^2 \times n_c^4)$.

4.3 Unwrap Loops

The following theorem shows that executing a loop twice is sufficient to find atomicity violations, if all iterations perform the same accesses.

Consider a loop such that every iteration contains the same sequence of access events. Let σ_2 denote an execution in which, at some point, a thread performs exactly two iterations of the loop. Let t_2 be the corresponding transaction containing the two iterations. Let σ_m be an execution that differs from σ_2 only in that, at the same point, the thread performs more than two iterations. Let t_m be the corresponding transaction containing the m iterations. Suppose t_2 and t_m differ only on the number of iterations.

Theorem 3. *$\langle t_2, A, <_H \rangle$ is not atomic iff $\langle t_m, A, <_H \rangle$ is not atomic.*

Proof Sketch: "⇒": it is obvious.

"⇐": We prove the contrapositive, *i.e.*, if $\langle t_2, A, <_H \rangle$ is atomic, then $\langle t_m, A, <_H \rangle$ is atomic. Given any trace π_m of $\langle t_m, A, <_H \rangle$, it must have a corresponding trace π_2 of $\langle t_2, A, <_H \rangle$, where π_m and π_2 differ only on the number of iterations for the loop. Because there must exist a way to swap π_2 into an equivalent serial trace, the events in π_m can be swapped in the same way. Specifically, if there is an event in π_m to prevent from swapping, an event in π_2 with the same properties (*i.e.*, accesses the same variable, holds the same lock, and observes the same happens-before relation) must exist to prevent π_2 from being serializable. Hence π_m has an equivalent serial trace, *i.e.*, $\langle t_m, A, <_H \rangle$ is atomic. □

5 Optimization: Dynamic Sharing Analysis

To reduce the runtime overhead of monitoring, we restrict monitoring to shared variables. Before an object becomes shared (*i.e.*, escapes from the thread that created it), all events involving it can be ignored. We designed and implemented dynamic sharing analysis to accurately determine the sharing property of each variable. This analysis extends our previous dynamic escape analysis [20] and introduces an additional execution on the same input before the atomicity analysis.

The first execution is used to determine whether each field of every class ever becomes shared during the entire run. Note that we do not construct and analyze hybrid trees during this execution. Each field of a class is processed independently, since some fields might be always accessed by a single thread even if the owner object is shared by multiple threads. Specifically, for each field, if that field in some instance has ever been accessed by multiple threads, the field of the corresponding class is marked as `shared`; otherwise, it is considered `unshared`.

During the second execution with the same input, we keep track of when an object (instead of field) becomes shared while constructing hybrid trees and analyzing atomicity violations. Fields classified as unshared from the first execution are not monitored. When an object becomes shared, all its monitored fields are marked as shared. To indicate whether an object has escaped from its creating thread, we add a boolean instance field to every class with the initial value `false`. We use Java reflection mechanism to dynamically update the field. An object o becomes shared in the following scenarios: (1) o is stored in a static field or a field of a shared object; (2) o is an instance of a thread and the thread is started; (3) o is referenced by a field of another object o', and o' becomes shared (this leads to cascading sharing); (4) o is passed as an argument to a native method that may cause it to be shared.

The dynamic sharing analysis is based on an assumption that given the same input, the sharings of a variable are the same during different executions, which is true in our experiment of Section 6. The dynamic sharing analysis has improved performance significantly. For example, it reduces the overall runtime by 40%

on the benchmarks `Tsp` and `Jigsaw` compared to the executions without the dynamic sharing analysis.

Another optimization is that, for access nodes with the same parent node, we preserve only the first two accesses in the same type (read or write) to each shared variable, because the first two accesses can represent all discarded accesses for checking atomicity. This is justified by Theorem 7.1 in [19].

6 Experiments

Program	LOC	Threads	Base	Dummy	Purely Dynamic			Hybrid			Code Coverage
					Time	nAV	NA methods	Time	nAV	NA methods	
Elevator	339	3	0.1	0.2	0.5	18	0-2-0	1	18	0-2-0	89.2%
Tsp	519	3	0.4	3.5	14	28	2-0-0	66.9	54	2-0-0	79.7%
Sor	8253	3	0.8	1.2	1.6	0	0-0-0	3.1	0	0-0-0	74.9%
Hedc	4267	3	0.3	0.4	0.5	7	1-0-0	0.6	7	1-0-0	35.1%
Jigsaw	100846	68	1.4	1.7	2.7	3	1-0-0	118.3	24	2-0-0	8.1%
Tomcat	168297	3	3.3	4.1	7.7	10	0-2-0	38.5	18	1-2-0	13.7%
Vector1.4	383	2	0.1	0.2	0.6	10	4-4-0	0.8	10	4-4-0	69.2%
Stack1.4	418	2	0.1	0.2	0.7	10	3-4-0	0.8	10	3-4-0	85.7%
HashTable1.4	597	2	0.2	0.3	0.6	4	0-4-0	0.9	4	0-4-0	47.5%

Fig. 6. Comparison of the purely dynamic commit node algorithm and the hybrid conflict-edge algorithm in performance and accuracy. The column "nAV" denotes the number of atomicity violations, which are counted based on the places in source code where the events involved in atomicity violations appear. The column "NA-methods" denotes the number of non-atomic methods with the categories being `bug` - `benign` - `false positive`. All times are measured in seconds.

We tested our tool on the following programs: `Elevator`, `Tsp`, `Sor`, and `Hedc` from [15], `Jigsaw 2.2.6` from [11], `Apache tomcat 6.0.16`, and `Vector`, `Stack`, and `Hashtable` from JDK 1.4.2.

We performed the experiments on a machine with 1.8 GHz Intel dual-core CPU, 2GiB memory, Windows XP SP3, and Sun JDK 1.6.

Figure 6 compares the running time and results of our hybrid algorithm with the purely dynamic commit node algorithm for conflict-atomicity in [19]. "Base" is the original program's running time without instrumentation. "Dummy" is the instrumented program's running time without analyzing atomicity violations (*i.e.*, analysis is not performed after intercepting the events). "Purely Dynamic" is the instrumented program's running time using the purely dynamic commit node algorithm in [19]. "Hybrid" is the running time of our hybrid algorithm. "Code Coverage" is the coverage of statements in the current execution, which is obtained using an Eclipse plugin EclEmma.

For `Tsp`, `HAVE` discovers more potential atomicity violations because of speculation. For example, we found that an atomicity violation involves a read on the static field `TspSolver.MinTourLen` in the speculative branch in the method `split_tour` and two writes on the same field in the executed code of the method `set_best` called by the method `recursive_solve`.

For `Jigsaw`, `HAVE` also reveals more atomicity violations than the purely dynamic approach. HAVE reports that the non-atomic method `perform` in `httpd.java` has multiple atomicity violations regarding several fields such as the instance field `LRUNode.next` and the instance field `ResourceStoreImpl.resources`. The previous purely dynamic approach missed this because some field accesses occur in speculatively executed branches.

For `Tomcat`, the static field `StringCache.accessCount` in the method `toString (ByteChunk bc)` of `StringCache.java` has the potential for atomicity violation when at least two threads find `StringCache.bcCache != null` and speculate the `else` branch. The same risk exists for the static field `StringCache.hitCount` in the same method, if both threads fail the condition test before it. We classify this atomicity violation as a bug, because it may cause the statistics to be inaccurate, even though this inaccuracy does not cause other incorrect behavior.

7 Related Work

The most closely related work is our commit-node algorithm in [19], which is purely dynamic. The main contribution of this paper is to extend it to a hybrid algorithm that combines static and dynamic analyses. This paper also presents a new optimization to the algorithm.

Dynamic algorithms to detect atomicity violations can be classified into two categories, based on whether they aim to detect *potential* atomicity violations (*i.e.*, whether any feasible permutation of an observed trace is unserializable), or *actual* atomicity violations (*i.e.*, whether an observed trace is unserializable). The algorithms to detect potential atomicity violations include this paper, Wang and Stoller's reduction-based, block-based algorithms, commit-node algorithms, [17, 20, 19], and Flanagan and Freund's reduction-based algorithm [6], which is similar to Wang and Stoller's reduction-based algorithm. Xu *et al.* infer computation units (subcomputations that the programmer might expect to be atomic) based on data and control dependencies and report an atomicity violation when an unserializable write by another thread is interleaved in a computation unit [21]. Lu *et al.*'s AVIO system learns access interleaving invariants as indications of programmers' likely expectations about atomicity and reports an atomicity violation when an observed interleaving violates an access interleaving invariant [12]. Flanagan *et al.* developed a sound and complete atomicity violation detector based on analysis of exact dependencies between operations [8]. Farzan and Madhusudan developed a space-efficient algorithm for detecting atomicity violations [4]. Park and Sen propose a randomized dynamic analysis technique that greatly increases the probability that a special class of potential atomicity violations will manifest as actual atomicity violations [13].

Static analyses have been developed to infer or verify atomicity of code segments, *e.g.*, [16, 9, 7, 18]. Static analysis gives stronger guarantees, because it considers all possible behaviors of a program, but is typically more restrictive or reports more false alarms than dynamic analysis. Model checking can also be used to check atomicity [5, 10, 4]. Model checking also provides strong guarantees but is feasible only for programs with relatively small state spaces.

Static and dynamic analyses can be combined in various ways for atomicity checking. Agarwal, Sasturkar, Wang, and Stoller used static analysis to reduce the overhead of the reduction-based algorithm [14] and the block-based algorithm [1]. JPredictor uses static analysis to improve the accuracy of the dependency relation used in dynamic checking for potential concurrency errors, including atomicity violations [3]. Those techniques, in contrast to ours, do not use speculative execution.

8 Conclusions and Future Work

This paper describes a new approach to enhance dynamic analysis with results from static analysis to make the dynamic analysis more effective at finding subtle atomicity violations, by augmenting the dynamic analysis to consider some of the behavior of unexecuted branches in the program. This is significant because software testing rarely achieves full code coverage in practice.

In our experiments, our hybrid conflict-edge algorithm scales almost as well as our previous dynamic algorithm [19]. Comparing our results in Figure 6 with results for those benchmarks in other papers [19, 20, 9, 6, 4, 8], our system detects all the atomicity violations detected by the purely dynamic algorithms described in those other papers and, for some benchmarks, detects additional atomicity violations.

Directions for future work include extending the static analysis to be interprocedural, taking the predicates guarding branches into account, incorporating more sophisticated approaches to identify transaction boundaries, and using a testcase generator to generate inputs that lead to execution of speculative events involved in atomicity violations to verify that they are not false alarms.

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)*, 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. F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for Java. In *Proc. 30th International Conference on Software Engineering (ICSE)*, pages 221–230. ACM, 2008.

4. 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.
5. C. Flanagan. Verifying commit-atomicity using model-checking. volume 2989 of *LNCS*, pages 252–266. Springer-Verlag, 2004.
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, 2004.
7. C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. 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 *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 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)*, 2003.
10. J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. volume 2937 of *LNCS*. Springer-Verlag, Jan. 2004.
11. Jigsaw, version 2.2.4. Available from http://www.w3c.org.
12. 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)*, 2006.
13. C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 135–145. ACM, 2008.
14. A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2005.
15. C. von Praun and T. R. Gross. Object race detection. volume 36(11) of *SIGPLAN Notices*, pages 70–82, Oct. 2001.
16. C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology, vol.3, no. 6*, June 2004.
17. L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2), 2003.
18. 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)*, June 2005.
19. 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, 2006.
20. L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *Transactions on Software Engineering*, 32(2):93–110, Feb. 2006.
21. 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)*, 2005.