AtomChase: Directed Search Towards Atomicity Violations

Mahdi Eslamimehr, Mohsen Lesani

This paper was presented at and awarded one of the two "Best Research Paper" awards at the 26th IEEE International Symposium on Software Reliability Engineering, November 2015, Gaithersburg, MD

VPRI Technical Report TR-2015-006

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

AtomChase: Directed Search towards Atomicity Violations

Mahdi Eslamimehr
Viewpoints Research Institute
Email: eslamimehr@ucla.edu

Mohsen Lesani
MIT
Email: lesani@mit.edu

***Abstract*—Atomicity violation is one of the main sources of concurrency bugs. Empirical studies show that the majority of atomicity violations are instances of the three-access pattern, where two accesses to a shared variable by a thread are interleaved by an access to the same variable by another thread. We present a novel approach to atomicity violation detection that directs the execution towards three-access candidates. The directed search technique comprises two parts: execution schedule synthesis and directed concurrent execution that are based on constraint solving and concolic execution. We have implemented this technique in a tool called AtomChase. In comparison to five previous tools on 22 benchmarks with 4.5 million lines of Java code, AtomChase increased the number of three-access violations found by 24%. To prevent reporting false alarms, we confirm the non-atomicity of the found execution traces. We present and prove sufficient conditions for non-atomicity of traces with the three-access pattern. The conditions could recognize the majority of 89% of the real atomicity violations found by AtomChase. Checking these conditions is two orders of magnitude faster than the exhaustive check.**

I. Introduction

The increasing dominance of hardware provides a multitude of threads that can concurrently access shared data. There are code blocks whose accesses to shared data should be executed without interference from other threads or the consistency of the data may be jeopardized. Programmers often use concurrency control mechanisms such as locks and transactional memory [23] to ensure *atomicity*. Programming atomicity is notoriously hard. A recent study [24] reported that 69% of concurrency bugs are atomicity violations. The same study found that 96% of concurrency bugs involve only two threads and 66% of (non-deadlock) concurrency bugs involve only one variable. Thus, researchers have focused on the *three-access* pattern [15], [25] as atomicity violation candidates. The pattern has the general form of two accesses to a shared variable by a thread that are interleaved by an access to the same variable by another thread.

Detection of atomicity violations has been a topic of recent attention. Previous research on atomicity violation detection can be divided into three categories: static detection, dynamic detection, and dynamic prediction. *Static* analysis approaches [20], [12] approximate the runtime behavior of the program at compile time to identify potential violations. Due to approximation, these approaches tend to report numerous false alarms. On the other hand, *dynamic* approaches [25], [35], [34], [13], [19], [28] detect atomicity violations in traces that are collected from program runs. These approaches do not report false alarms, however, considering the magnitude of production code-bases, an undirected run is unlikely to hit the violating paths. Therefore, dynamic *predicative* approaches [15], [17], [14], [5] try to find atomicity violations not only in the given trace of a program run but also in interleavings of the trace. Thus, predictive approaches increase the probability of catching bugs. Nevertheless, the bugs that occur in program paths that are not taken by the test run remain undiscovered.

The previous dynamic approaches can check whether three-access candidates appear in traces obtained from program runs or the interleavings of these traces. However, they cannot search towards three-access candidates. We present a novel approach to atomicity violation detection that *directs* the execution towards three-access candidates. Given a three-access candidate i.e. a triple of static program locations, we search for an input and an execution trace with the candidate violation. The search process comprises two modules that work in tandem: execution plan synthesis and directed concurrent execution. The plan synthesis module suggests execution plans that can lead to the violation candidate. It represents violating executions as constraints and applies SMT solvers to generate plans. The directed concurrent execution module tries to execute the suggested plan. It employs concolic execution [22], [3], [4] to iteratively follow the plan. The resulting execution trace is fed back to the plan synthesis module. Repeating plan synthesis and plan execution steers the search towards an executable plan that exhibits the violation candidate.

We have implemented the directed search technique in an atomicity violation detection tool for Java called AtomChase. We adopted 22 open-source benchmarks with a total of more than 4.5 million lines of code. We compared AtomChase with five existing atomicity violation detectors on these benchmarks. In comparison to these five tools combined, AtomChase found execution traces for 537 three-access candidates of which 423 were previously known and 114 are new three-access candidates, i.e. AtomChase found 91% of the previously found three-access violations and increased the found three-access violations by 24%. More than a quarter of the candidates were found after more than one million computation steps. AtomChase can replay the execution traces that it reports. AtomChase is fully automatic and the user does not need any expertise on atomicity violation detection.

A majority of atomicity violations are instances of the three-access pattern; however, not every three-access instance is an atomicity violation. We show an execution with a three-access pattern that is provably atomic. The usability of testing tools is severely affected by the frequency of their false alarms. Therefore, it is crucial to confirm the non-atomicity of an

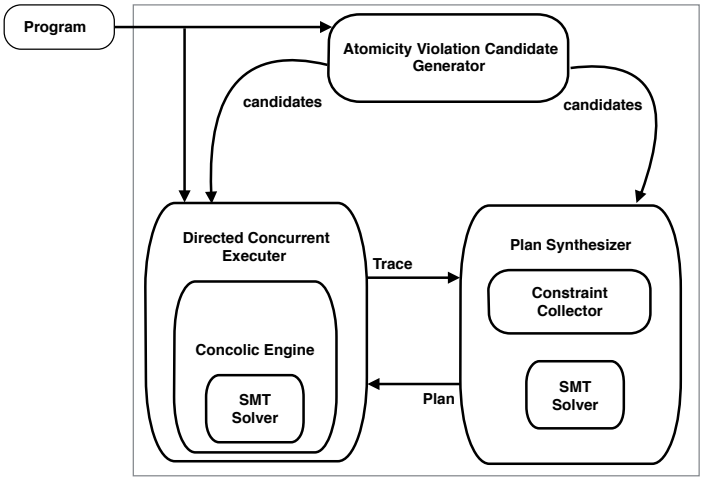

Fig. 1. Directed Search for Atomicity Three-access Candidates

Algorithm 1 Directed Search for Atomicity Three-access Candidates

```
1:  function SEARCHATOMICITYVIOLATIONS(prog)
2:      violations = ∅
3:      candidates = CandidateGenerator(prog)
4:      foreach c ∈ candidates do
5:          t = Executor(prog, c)
6:          while ¬TimeOut do
7:              p = PlanSynthesizer(t, c)
8:              t = Executor(prog, p)
9:              if IsViolation(t) then
10:                 violations = violations ∪ {t}
11:                 break
12:     return violations
```

execution that is an instance of a three-access pattern before reporting it to the user. We present and prove sufficient conditions for non-atomicity of traces that contain instances of three-access pattern. Our experiments show that these conditions could recognize 89% of the real atomicity violations found by the search technique. In addition, checking these conditions has linear time complexity on the size of the input trace. The experiments show that checking them is more than two orders of magnitude faster than the exhaustive checks.

In summary, the main contributions of this paper are:

- An atomicity violation detection technique that can direct execution towards three-access candidates;
- Specification and proof of sufficient conditions for non-atomicity of the three-access pattern;
- An atomicity violation detection tool for Java and comparison with five existing tools.

In the rest of the paper, we first illustrate our search technique and its components. Then, we present our sufficient conditions for non-atomicity of the three-access pattern. Next, we present our implementation and experimental results. Finally, we discuss related work and conclusions.

II. Directed Search

In this section, we first present an overview of our search technique and its comprising modules. Then, we explain each module in more detail in the following subsections.

Atomicity blocks are code blocks containing accesses to shared data that should execute without interference from other threads. Many studies consider synchronized method bodies or statements as atomicity blocks. Atomicity can be violated if accesses in an atomicity block by a thread are interleaved by an access from another thread. The triple of events (e_1, e_2, e_3) in an execution trace π is an instance of the three-access pattern if (1) The events e_1, e_2 and e_3 access the same location. (2) The event e_1 is executed before e_2 and e_2 is executed before e_3 in π (while other events may be executed in between), (3) The events e_1 and e_3 are executed by the same thread and atomicity block, and (4) The thread of e_2 is not the same as the thread of e_1 and e_3. A three-access candidate is a triple of program statements (s_1, s_2, s_3) such that s_1 and s_3 are in the same atomicity block and s_2 is not in that atomicity block. Execution of the three statements can make an instance of the three-access pattern if the executions of s_1 and s_3 by the same thread are interleaved by the execution of s_2 (by another thread). Our search starts with a set of three-access candidates. For each of the candidates, we do a separate search.

Figure 1 shows an overview of the modules of the search process and their interconnections, and Algorithm 1 presents the search algorithm. The search is an iterative application of the directed concurrent executor and the plan synthesizer modules. First, the directed concurrent executor tries to execute all the three statements of the candidate regardless of their order. The executor module uses a concolic execution engine that in turn uses an SMT solver. The initial execution trace is fed to the plan synthesizer. Given a trace, the plan synthesis module mutates the trace and outputs an execution plan. A plan is a sequence of milestone events to guide the execution to the violation candidate. The plan synthesizer first generates a set of constraints that represent a consistent reordering of the input trace where the events of the candidate come in the candidate order. It then solves the constraints using an SMT solver to yield an execution plan. The plan is then fed to the executor module that attempts to execute it. If the plan is found to be executable, the search returns successfully with a concrete execution trace of the violation candidate. Otherwise, the executor module continues execution beyond the plan, typically until termination of the program. The plan synthesizer and the executor modules are repeated in turn. Each call to the executor module may produce a trace that is closer to the violation candidate, after which a call to the plan synthesizer module further improves the plan. The alternation of the executor and the plan synthesis modules is considerably more powerful than either one alone. The execution module alone finds only 36% (196 of 537) and the plan synthesis module alone finds only 11% (63 of 537) of the atomicity violations that they can find together.

Let us look at the simple example in Figure 2. Imagine that Alice and Bob have a joint checking account with a balance of $20. Alice uses an ATM with no fee, while Bob uses an ATM with a $5 fee. The function `input()` returns the amount of withdrawal input to the ATM.

```
1  class BankAccount {
2     int balance;
3     BankAccount(int b) { balance = b; }
4     synchronized int getBlance() {
5        return balance;
6     }
7     synchronized int setBalance(int b) {
8        balance = b;
9     }
10 }
```

```
1  BankAccount a = new BankAccount(20);
```

```
1  // Thread Alice:
2      int amount = input();
3      int serviceFee = 0;
4      int total = amount + serviceFee;
5  e1: int balance = a.getBalance();
6  e2: if (total <= balance)
7  e3:    a.setBalance(balance - total);
```

```
1  // Thread Bob:
2      int amount = input();
3      int serviceFee = 5;
4      int total = amount + serviceFee;
5  e4: int balance = a.getBalance();
6  e5: if (total <= balance)
7  e6:    a.setBalance(balance - total);
```

Fig. 2. Example of Directed Search for an Atomicity Candidate

Consider the three-access violation candidate (e_1, e_6, e_3). First, we run the execution module to find a trace having the three events of the candidate regardless of their order. We need to find a trace that executes e_1 and e_3 in the first thread and e_6 in the second thread. The execution can start with \$10 input value for the first thread and succeed passing the condition at e_2 and execute e_3. However, choosing \$10 input value for the second thread makes the condition at e_5 fail and e_6 is not executed. A second iteration for the second thread may choose \$5 as the input that successfully passes the condition at e_5 and executes e_6. The resulting trace is $\pi = e_1, e_2, e_3, e_4, e_5, e_6$. Let $\prec_\pi$ denote the total order of the trace π.

Next, the plan synthesis module tries to generate a plan that directs the execution towards the candidate. In particular, it should generate a reordering π' of the input trace π that has the candidate events in order. Thus, the event e_6 should come between e_1 and e_3 yielding the constraints $e_1 \prec_{\pi'} e_6 \wedge e_6 \prec_{\pi'} e_3$. Moreover, the new trace should preserve the intra-thread order of events. Thus, e_1 should come before e_2 that should come before e_3. Similarly, e_4 should come before e_5 that should come before e_6. These orders are captured by the following constraints: $e_1 \prec_{\pi'} e_2 \wedge e_2 \prec_{\pi'} e_3$ and $e_4 \prec_{\pi'} e_5 \wedge e_5 \prec_{\pi'} e_6$. In addition to the above constraints, the plan synthesis module generates constraints for the consistency of locations and synchronization objects that we present in section II-B. A solution to the above constraints can be $\pi' = e_1, e_4, e_2, e_5, e_6, e_3$.

The execution module tries to execute the constructed plan. Choosing values \$10 for both threads makes the plan executable. Each thread first reads the initial \$20 balance and passes the withdrawal condition. Then, each thread updates the balance. Note that the Alice's write at e_3 overwrites Bob's write at e_6. This execution is in fact an atomicity violation. Alice and Bob withdrew a total of \$25 together but only \$15 was deducted from their account. The resulting execution contains an instance of the violation candidate.

A. Directed Concurrent Execution

Given a concurrent program and a plan, the directed concurrent execution module tries to find a program execution that includes all the events of the plan in order. (The plan events should not necessarily execute one after another; other events can execute in between.) If it can successfully drive the program execution to follow the plan, it returns the execution trace (and the input values). Otherwise, it executes the longest executable prefix of the plan. Then, it continues execution and tries to execute the remaining events disregarding the plan order. The execution is based on the generalization of directed testing to execution plans [10], [11]. In contrast to the previous work that starts with a random initial run, the current work tries to start with an execution that contains the three program statements of the three-access candidate.

The classical directed testing [22], [3], [4] tries to iteratively move toward a target statement in the program. The idea is to use information from one execution to generate a better input for the next one. The execution method called *concolic* execution not only executes the program *concretely* but also records *symbolic* constraints along the execution. In particular, it records constraints based on a assignments and branch conditions . If a branch leads the execution off the path to the target statement, the negation of the branch condition is recorded. A solution to the collected constraints yields the next input for the program. The new input keeps the execution on the path to the target statement at the last failing branch. If the target statement is deeply nested, many concolic executions may be needed before the target statement can be reached.

Directed testing can be generalized to execution plans. Given a concurrent program and a plan, the goal is to find inputs that lead to a program execution including all the events of the plan in order. We start from the first event of the plan and iteratively extend the execution to hit all the events. On each iteration, the target is the next event of the plan. Each iteration involves concolic execution in the thread that contains the target event. We control the thread scheduler to keep the execution in the thread with the target event. Similar to classical directed testing, the target event is reached though sub-iterations. The concolic execution executes the program and records constraints. The input for the next sub-iteration is obtained by solving the constraints from the previous sub-iteration. If we can hit all the events of the plan, we return success together with the constructed execution trace. Otherwise, we continue the execution in an attempt to execute as many of the events of the plan as possible even if they are executed out of the plan order. The next call to the plan synthesis module will benefit from the presence of these trace events.

B. Execution Plan Synthesis

The goal of the execution plan synthesis module is to generate an execution plan that leads to the violation candidate.

$$
\begin{aligned}
\phi_1 &= \textstyle\bigwedge_{e\in\pi}(e \preceq_{\pi'} e) \wedge \\
&\quad \textstyle\bigwedge_{e_1,e_2\in\pi}(e_1 \preceq_{\pi'} e_2 \wedge e_2 \preceq_{\pi'} e_1) \Rightarrow (e_1 = e_2) \wedge \\
&\quad \textstyle\bigwedge_{e_1,e_2\in\pi}(e_1 \preceq_{\pi'} e_2 \wedge e_2 \preceq_{\pi'} e_3) \Rightarrow (e_1 \preceq_{\pi'} e_3) \wedge \\
&\quad \textstyle\bigwedge_{e_1,e_2\in\pi}(e_1 \preceq_{\pi'} e_2 \vee e_2 \preceq_{\pi'} e_1) \\
\phi_2 &= \textstyle\bigwedge_{e_1,e_2\in\pi}((e_1 \preceq_{\pi} e_2 \wedge thr(e_1) = thr(e_2)) \Rightarrow e_1 \preceq_{\pi'} e_2) \\
\phi_3 &= \textstyle\bigwedge_{e_r\in\pi}(obj(e_r) \in Loc \wedge op(e_r) = read) \Rightarrow \\
&\qquad \textstyle\bigvee_{e_w\in\pi}(obj(e_w) = obj(e_r) \wedge op(e_w) = write \wedge arg(e_w) = ret(e_r) \wedge \\
&\qquad\quad e_w \prec_{\pi'} e_r \wedge \\
&\qquad\quad \textstyle\bigwedge_{e_{w'}\in\pi}(obj(e_{w'}) = obj(e_r) \wedge op(e_{w'}) = write) \Rightarrow \\
&\qquad\qquad (e_{w'} \preceq_{\pi'} e_w \vee e_r \prec_{\pi'} e_{w'})) \\
\phi_4 &= \textstyle\bigwedge_{e_u\in\pi}(obj(e_u) \in Lock \wedge op(e_u) = unlock) \Rightarrow \\
&\qquad \textstyle\bigvee_{e_l\in\pi}(obj(e_l) = obj(e_u) \wedge op(e_l) = lock \wedge thr(e_l) = thr(e_u) \wedge \\
&\qquad\quad e_l \prec_{\pi'} e_u \wedge \\
&\qquad\quad \textstyle\bigwedge_{e_{u'}\in\pi}(obj(e_{u'}) = obj(e_u) \wedge op(e_{u'}) = unlock) \Rightarrow \\
&\qquad\qquad (e_{u'} \prec_{\pi} e_l \vee e_u \preceq_{\pi} e_{u'})) \\
\phi_5 &= (e_1^* \preceq_{\pi'} e_2^*) \wedge (e_2^* \preceq_{\pi'} e_3^*) \\
\phi &= \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5
\end{aligned}
$$

Fig. 3. Constraints for the new trace π' given the trace π and the violation candidate (e_1^*, e_2^*, e_3^*)

The inputs to the module are the last execution trace and the violation candidate. We formulate the plan synthesis problem as a satisfiability problem. We construct and solve constraints that represent a consistent reordering of the events of the input trace that contains the input violation candidate.

Let t denote a thread identifier, a denote an atomicity block identifier, x denote an object, m denote an operation on shared objects, and v denote a value. An event e is a tuple (t, a, x, m, v, v') where t is the identifier of the executing thread, a is the identifier of the executing atomicity block, x is the shared object, m is the operation applied to x, v is the argument value (or a dummy value if m does not have a parameter) and v' is the returned value (or a dummy value if m does not have a return value). Each event that is not executed inside an atomicity block has a unique atomicity block identifier. In other words, events that are executed outside atomicity blocks are regarded as singleton atomicity blocks. Let the functions thr, $block$, obj, op, arg and ret map an event to its six above components respectively. We call an event with a $read$ operation a read event, and an event with a $write$ operation a write event.

A trace π is a sequence of events. The membership of an event e in a trace π is denoted as $e \in \pi$. Let the relation $\preceq_\pi$ represent the total order of events of π. For a trace π and a thread t, let $\pi|_t$ is the sequence of events in π for a particular thread t.

Let π be the input trace, (e_1^*, e_2^*, e_3^*) be the candidate events, and T be the set of threads of π. The trace π is a possible interleaving of the set of sub-traces $\{\pi|_t \mid t \in T\}$. The search state space is the set of interleavings of these sub-traces. We want to find an interleaving that preserves the consistency of locations, complies with the semantics of synchronization objects, and matches the order of events of the candidate.

We construct a quantifier-free formula ϕ that is depicted in Figure 3. Given the trace π and its order of events $\preceq_\pi$, we encode the following conditions on the order of events $\preceq_{\pi'}$ of the new trace π':

- Totality of the execution order (ϕ_1): The order of events $\preceq_{\pi'}$ of the trace π' is a total order. It is a reflexive, anti-symmetric, transitive and total relation.
- Intra-thread execution order preservation (ϕ_2): The order of events of a thread is preserved. If an event e_1 is before another event e_2 in π and they are in the same thread, then e_1 is before e_2 in π' as well. However, events of different threads can be reordered.
- Consistency of location objects (ϕ_3): The value read from a location is the value written by the most recent write to the same location, i.e. for every $read$ event e_r from a location l, there is a $write$ event e_w on l that writes the same value that e_r reads and e_w is the last $write$ event on l that comes before e_r in π'.
- Semantics of synchronization objects (ϕ_4): An unlocked operation of a lock l is executed only if a lock operation is executed on l by the same thread before it such that no other unlock operation on l interleaves these two operations. (We assume that there is an initial unlock operation.) We can similarly model other synchronization object types.
- Order of three violating events (ϕ_5): The three events of the three-access candidate come in order in π'.

The solution to the above constraints represents the order of events in the trace π' that satisfies the semantics of location and synchronization objects and contains the three-access candidate. There is a trade-off between the executability of the plan and the speed of synthesizing it i.e. more precise constraints for synchronization objects can result in more executable plans. However, the complexity of these constraints can hinder the performance of the plan synthesis module. Instead of increasing the complexity of the constraints to enforce the executability of the resulting plan, we verify the executability of the plan in the next call to the execution module.

III. Non-atomicity of Three-Access Pattern

In this section, we present sufficient conditions for non-atomicity of traces with instances of the three-access pattern. For the sake of simplicity of the presentation, in this section we focus on traces that are composed of events on memory locations.

A trace is *sequential* if its atomicity blocks execute sequentially. More precisely, a trace π is sequential, $sequential(\pi)$, if and only if for every triple of events $e_1, e_2, e_3 \in \pi$, if $e_1 \preceq_\pi e_2 \preceq_\pi e_3$ and $block(e_1) = block(e_3)$, then $block(e_2) = block(e_1)$. This notion of atomicity aligns with the classical definition of serializability. [26]

Two traces are *equivalent* if one is a permutation of the other one and they have the same intra-thread order of events. More precisely, two traces π_1 and π_2 are equivalent, $\pi_1 \sim \pi_2$, if and only if for every event e, $e \in \pi_1$ if and only if $e \in \pi_2$ and for every thread t, $\pi_1|_t = \pi_2|_t$.

A trace is *consistent* if every read from a memory location returns the value that the last write to the location before it writes. More precisely, a trace π is consistent, $consistent(\pi)$, if and only if for every $read$ event e_r in π, if e_w is the last $write$ event to location $obj(e_r)$ before e_r in π, then $ret(e_r) = arg(e_w)$. We say that the event e_w justifies the event e_r.

A trace is *atomic* if it has a consistent equivalent sequential trace. More precisely, a trace π is atomic, $atomic(\pi)$, if and only if there exists a trace π' such that $sequential(\pi')$, $\pi' \sim \pi$, and $consistent(\pi')$.

Not every instance of the three-access pattern is an atomicity violation. For example, consider Figure 4. Figure 4(a) depicts a trace π with two threads. The two columns show the operations of the two threads separately. A $read$ event on a location x that returns the value v is denoted as $read(x)\colon v$ and a $write$ event on a location x that writes the value v is denoted as $write(x, v)$. The left thread executes an atomicity block containing events e_1 and e_3. The right thread executes two operations e_0 and e_2 outside atomicity blocks. The triple of events (e_1, e_2, e_3) is an instance of the write-read-write three-access pattern. The event e_1 writes a value to x that e_2 reads and e_3 later overwrites. Nonetheless, the atomicity of π is justified by the consistent equivalent sequential trace π' depicted in Figure 4(b). In π, the event e_2 reads the value v_1 written by e_1 while in π', the event e_2 is moved before e_1. However, in π', the value v_1 that e_2 reads is justified by the value v_1 that e_0 writes.

The definition of atomic trace provides an exhaustive method to check non-atomicity of a trace. A trace is not atomic if every trace that is equivalent to it and sequential is not consistent. The exhaustive method constructs all the possible equivalent and sequential traces for the input trace and checks that *none* of them are consistent. Although straightforward, the exhaustive method has exponential time complexity on the size of the input trace.

In Figure 5, we present sufficient conditions for non-atomicity of traces with the three-access pattern. The conditions are labeled according to the operation types of the three events. For example, C_{WRW} is the condition for the write-read-write three-access pattern where the first event is a write, the second one is a read and the third one is a write. The theorem below states that each of these conditions is sufficient for the non-atomicity of the trace. It is notable that checking these conditions has a linear time complexity on the size of the input trace. We will benefit from these conditions in AtomChase to quickly filter out a large fraction of false alarms. Let us consider the condition C_{WRW} as an example. The condition requires that (1) The events e_1, e_2 and e_3 are on the same variable, they are write, read and write operations respectively, the events e_1 and e_3 are from the same atomicity block, and e_2 is by a different thread, (2) The return value of e_2 is not equal to the argument value of e_3, and (3) There is no write in the trace before e_1 and by the same thread as e_1, before e_2 and by the same thread as e_2, or after e_3 and by the same thread as e_3, that writes the returned value of e_2.

Let us see how these conditions imply the non-atomicity of a trace π. We assume that π is atomic and arrive at a contradiction. The three above conditions are represented in Figure 6(a) as follows: (1) The two events e_1 and e_3 that are executed by the same atomicity block are shown in the left column. The event e_2 that is executed by a different thread and interleaves e_1 and e_3 is shown in the right column and is vertically located between e_1 and e_2. (2) The inequality of the return value v_2 of e_2 and the argument value v_3 of e_3 is written below the trace. (3) The crossed events depict the write events that cannot exist in the trace.

By the definition of atomicity, we have that there exists a consistent equivalent sequential trace π'. As π' is sequential, the event e_2 is either before the atomicity block containing the events e_1 and e_3 or after it in π'. These two cases are shown in Figure 6(b) and Figure 6(c). As π' is consistent, and contains the event e_2 that reads the value v_2, there should be a justifying write event before e_2 that writes v_2. We consider each case in turn.

- In Figure 6(b), e_2 comes before the atomicity block. By the equivalence of π and π' and the third condition above, we have that there is no write event that writes v_2 before e_1 or before e_2 in π'. Thus, the read value v_2 of e_2 cannot be justified. Hence, π' is not consistent that is a contradiction.
- In Figure 6(c), e_2 comes after the atomicity block. By the equivalence of π and π' and the third condition above, we have that there is no write event that writes v_2 after e_3 or before e_2. Thus, there is no write event between e_3 and e_2 that can justify the read value of e_2. In addition, from the second condition above, we have $v_2 \neq v_3$. The event e_3 writes v_3 and the event e_2 reads v_2. Thus, the event e_3 cannot justify e_2. Therefore, the value that the last write before e_2 writes is not the value that e_2 reads. Hence, π' is not consistent that is a contradiction.

Similarly, we can show that the stated conditions for the triple of other operation types are sufficient for non-atomicity of traces that contain them. We state these results in the following theorem.

Theorem 1 (Non-atomicity of Three-access Pattern). $\forall \pi, e_1, e_2, e_3\colon C(\pi, e_1, e_2, e_3) \Rightarrow \neg atomic(\pi)$

Please see the appendix [9] for the proofs.

	e_0: $write(x, v_1)$
e_1: $write(x, v_1)$	
...	
	e_2: $read(x)$: v_1
e_3: $write(x, v_3)$	

(a)

	e_0: $write(x, v_1)$
	e_2: $read(x)$: v_1
e_1: $write(x, v_1)$	
...	
e_3: $write(x, v_3)$	

(b)

Fig. 4. Atomicity of a trace with the write-read-write three-access pattern

$$
\begin{aligned}
&IsPattern(e_1, e_2, e_3, o_1, o_2, o_3) = \\
&\quad obj(e_1) = obj(e_2) = obj(e_3) \wedge op(e_1) = o_1 \wedge op(e_2) = o_2 \wedge op(e_3) = o_3 \ \wedge \\
&\quad block(e_1) = block(e_3) \wedge thr(e_2) \neq thr(e_1) \\
&NoWPred(\pi, e, v) = \\
&\quad \textstyle\bigwedge_{e' \in \pi}(e' \preceq_\pi e \wedge thr(e') = thr(e) \wedge obj(e') = obj(e) \wedge op(e') = write) \Rightarrow \\
&\qquad (arg(e') \neq v) \\
&NoWSucc(\pi, e, v) = \\
&\quad \textstyle\bigwedge_{e' \in \pi}(e \preceq_\pi e' \wedge thr(e') = thr(e) \wedge obj(e') = obj(e) \wedge op(e') = write) \Rightarrow \\
&\qquad (arg(e') \neq v) \\
&NoWBet(\pi, e_1, e_2, v) = \\
&\quad \textstyle\bigwedge_{e' \in \pi}(e_1 \preceq_\pi e' \preceq_\pi e_2 \wedge thr(e') = thr(e_1) \wedge obj(e') = obj(e_1) \wedge op(e') = write) \Rightarrow \\
&\qquad (arg(e') \neq v) \\
&C_{WRW}(\pi, e_1, e_2, e_3) = \\
&\quad IsPattern(e_1, e_2, e_3, write, read, write) \wedge \\
&\quad ret(e_2) \neq arg(e_3) \wedge NoWPred(\pi, e_1, ret(e_2)) \wedge NoWPred(\pi, e_2, ret(e_2)) \ \wedge \\
&\quad NoWSucc(\pi, e_3, ret(e_2)) \\
&C_{RWR}(\pi, e_1, e_2, e_3) = \\
&\quad IsPattern(e_1, e_2, e_3, read, write, read) \wedge \\
&\quad ret(e_1) \neq ret(e_3) \wedge NoWBet(\pi, e_1, e_3, ret(e_3)) \\
&C_{WWR}(\pi, e_1, e_2, e_3) = \\
&\quad IsPattern(e_1, e_2, e_3, write, write, read) \wedge \\
&\quad arg(e_1) \neq ret(e_3) \wedge NoWBet(\pi, e_1, e_3, ret(e_3)) \\
&C_{RWW}(\pi, e_1, e_2, e_3) = \\
&\quad IsPattern(e_1, e_2, e_3, read, write, write) \wedge \\
&\quad ret(e_1) \neq arg(e_2) \wedge NoWPred(\pi, e_1, ret(e_1)) \wedge NoWPred(\pi, e_2, ret(e_1)) \ \wedge \\
&\quad NoWSucc(\pi, e_2, ret(e_1)) \\
&C(\pi, e_1, e_2, e_3) = \\
&\quad C_{WRW}(\pi, e_1, e_2, e_3) \vee C_{RWR}(\pi, e_1, e_2, e_3) \vee C_{WWR}(\pi, e_1, e_2, e_3) \ \vee \\
&\quad C_{RWW}(\pi, e_1, e_2, e_3)
\end{aligned}
$$

Fig. 5. Sufficient conditions for non-atomicity of traces with the three-access pattern.

IV. Implementation

We have implemented our directed search technique in a tool called AtomChase. It is written in and works on Java 6 programs. We implemented the algorithm HAVE [7] to quickly produce three-access candidates. We work with events at the Java bytecode level and use Soot [32] version 2.5.0 to instrument bytecode. We implemented our directed concurrent execution module on top of the Lime[1] concolic execution engine version 2.2.0. We used Yices[2] [8] SMT solver version 2.2.0 to solve the constraints of the plan synthesis module. AtomChase uses the same memory space as the benchmarks.

Time optimization by parallel constraint solving. Constraint solving is a time-consuming procedure, particularly when a solver being asked to solve billions of constraints. We applied a heuristic to parallelize the constraint solving step. We treat event total order constraints (ϕ_1) separate from the other constraints. The plan synthesis module first generates execution plans that satisfy ϕ_1 constraints, and then we use the constraint solver to verify if each plan satisfies other constraints too. Since generating and verifying a plan are independent procedures we can perform them in parallel. Therefore the constraint solving procedure can be split into many tasks and assigned to numerous threads each handling a candidate plan.

Handling dynamic class loading. Many of the benchmarks use reflection. Unfortunately, among the atomicity violation detectors that we compare against, only DoubleChecker [1] handles reflection well. We enable the other atomicity violation detectors to handle reflection with the help of the

[1] http://www.tcs.hut.fi/Software/lime
[2] http://yices.csl.sri.com

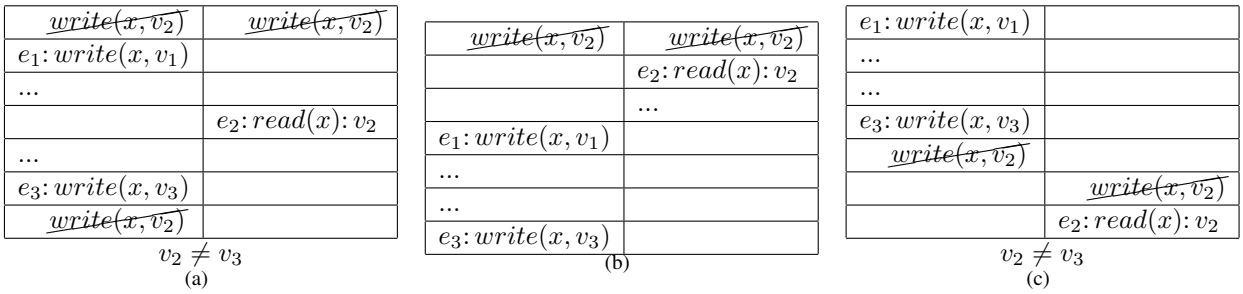

Fig. 6. The sufficient conditions for non-atomicity of the write-read-write three-access pattern

toolchain TamiFlex [2]. The core of the problem is that reflection is at odds with static analysis and bytecode instrumentation: reflection may make static analysis unsound and may load uninstrumented classes. We have combined each of AtomFuzzer [27], Penelope [31], Atomizer [17] and HAVE with TamiFlex and we have run all our experiments without warnings. We believe that, all the atomicity violation detectors handle reflection correctly.

V. Experimental Results

A. Benchmarks and Platform

We adopt 22 open-source benchmarks of more than 4.5 million lines of Java code from different sources including Grande and DaCapo suites. Figure 7 lists our 22 benchmarks. Please see the appendix [9] for more description of each benchmark.

The sizes of the benchmarks vary widely: we have 2 huge (1M+ LOC), 10 large (20K–1M LOC), 8 medium (1K–8K LOC), and 2 small (less than 1K LOC) benchmarks. Figure 7 also lists the high watermark of how many threads each benchmark runs, and the input size in bytes for each benchmark. Our benchmarks are drawn from open source repositories and most of them come with a specific input. For each benchmark, we use the predetermined input that appears to exercise the code well. Benchmarks in the Grande suite are packaged in three different input sizes. We run the Grande benchmarks with size A. DaCapo suite has a harness to run each benchmark with preset thread numbers, environment variable and program inputs. DaCapo has three workload size: small, default, and large. We conducted all experiments with the default setting. Test harnesses for Colt, TSP, Hedc and open programs like ArrayList, TreeSet, HashSet, Vector were downloaded from the Penelope homepage.[3] We ran all our experiments on a Macbook pro 2.3 GHz Intel core i7, 16 GB 1600 Mhz DDR3, Mac OS X 10.9.4.

B. Atomicity Violation Detectors

We compare AtomChase with four dynamic atomicity violation detectors: DoubleChecker, AtomFuzzer, Penelope, Atomizer and a hybrid atomicity detector, namely HAVE. We present a summary of these tools below. Additionally, we compare with a combination of the results of all the tools above that we call DAPA[4].

[3] http://web.engr.illinois.edu/~sorrent1/penelope/experiments.html

[4] The name DAPA is composed of the first letter of the names of the four tools DoubleChecker, AtomFuzzer, Penelope, and Atomizer

Name	LOC	# threads	input size (bytes)
Sor	1270	5	404
TSP	713	10	58
Hedc	30K	10	220
Elevator	2840	5	60
ArrayList	5866	26	116
TreeSet	7532	21	64
HashSet	7086	21	288
Vector	709	10	128
RayTracer	1942	5	412
MolDyn	1351	5	240
MonteCarlo	3619	4	26
Derby	1.6M	64	564
Colt	110K	11	804
Avrora	140K	6	74
Tomcat	535K	16	88
Batic	354K	5	366
Eclipse	1.2M	16	206
FOP	21K	8	34
H2	20K	16	658
PMD	81K	4	116
Sunflow	108K	16	24
Xalan	355K	9	616
TOTAL	4587K		

Fig. 7. The benchmarks.

HAVE first runs a static analysis to collect information about shared and synchronization variables. It then uses the collected static data in a dynamic analysis that approximately predicts atomicity violations in unexplored part of the program. Because of the nature of static speculation, HAVE predictions may include false positives.

DoubleChecker performs two dynamic analyses: imprecise and precise. The imprecise analysis builds a graph that collects imprecise cross-thread dependencies among atomicity blocks. Cycles in the graph are considered as atomicity violation candidates. The precise analysis collects precise information about cross-thread dependencies and checks the candidates that were found by the imprecise analysis. DoubleChecker has two run modes: the single-run and multi-run. The single-run mode performs both analyses in a single run. The multi-run mode first performs the imprecise analysis in the first round of execution, then it performs both the precise and the imprecise analyses in the second run. The single-run mode is faster but the multi-run mode is more accurate.

AtomFuzzer is a dynamic tool that looks for the three-access atomicity violation pattern. It executes the program with a random scheduler that is biased towards executing events that could lead to a specific type of atomicity violation. Whenever a thread enters an atomic block and is about to acquire the same lock that it has previously acquired and released, it pauses the thread and continues execution until another thread is about to acquire the same lock. At this point, AtomFuzzer detects a three-access violation and reports it.

Penelope is a dynamic predictive atomicity violation detector. It first runs the program randomly and collects an execution trace. It represents the possible re-interleavings of the obtained trace that contain atomicity violations as constraints. It uses a constraint solver to yield a violating trace.

Atomizer is a dynamic atomicity checker that uses simple heuristics to locate atomic blocks. It gets a single test run and uses a lockset algorithm [29] combined with a reduction analysis to verify the atomicity of blocks labelled as atomic.

In all the above tools, we configured the atomicity blocks to be the synchronized methods and statements. This is the default assumption in Atomizer, AtomFuzzer and Penelope. We used Jikes RVM 3.1.3 to run DoubleChecker and used Oracle JVM 1.6 for the rest of the tools. We let the JVM adjust the heap size automatically.

C. Measurements

Figure 8 shows the number of three-access violations that we found in the 22 benchmarks by the 6 techniques. For AtomChase, the first column shows the total number of three-access violations that AtomChase found, the second column shows the number of three-access violations that AtomChase found but DAPA did not find and the third column shows the number of three-access violations that both AtomChase and DAPA found. The columns EV and CV show the number of three-access violations that were validated as non-atomic by the exhaustive check and the sufficient conditions check (presented in section III) respectively. We identify each bug by its three-access candidate i.e. the three program locations and their execution order. We count once all the bugs with the same three-access candidate. Similarly, we compare the bugs found by two different tools based on the three-access candidates to determine the set of disjoint and common bugs. We only count the errors and not warnings.

Figure 9 shows the run time of each tool on each benchmark and shows the geometric mean for each technique. We made no attempt to throttle the amount of time that the tools can use. The timings for AtomChase include the time to execute HAVE. The last two columns shows the run time of the exhaustive and sufficient condition checks on the three-access traces found by AtomChase. The numbers in every column other than the Exhaustive Validation (EV) column are in seconds. The numbers in the EV column are in hours.

The following table shows the lengths of the 537 atomicity violation traces that AtomChase found (including the 114 traces found only by AtomChase):

Schedule length	Total =	AtomChase +	DAPA
$10^2 - 10^3$	36	0	36
$10^3 - 10^4$	29	2	27
$10^4 - 10^5$	113	9	104
$10^5 - 10^6$	199	7	192
$10^6 - 10^7$	61	23	38
$10^7 - 10^8$	87	63	24
$10^8 - 10^9$	12	10	2
	537	114	423

D. Assessment

We now present our findings based on both the measurements listed above and our additional analysis of the atomicity violations that were found.

AtomChase versus other Tools. We can see in Figure 8 that AtomChase finds the most atomicity violations (537) of all the techniques. Among those 537 atomicity violations, 114 cases were found only by AtomChase and are entirely novel to this paper while 423 cases were found by DAPA as well. Dually, 39 atomicity violations were found only by DAPA. In summary, we have that the combination of DAPA and AtomChase found 576 atomicity violations in the 22 benchmarks.

Found by both:	423
Found only by DAPA:	39
Found only by AtomChase:	114
Total:	576

Hence, AtomChase increases the number of found atomicity violations by 24% (114 of 462). Let us consider atomicity violations that AtomChase found but DAPA missed. One such atomicity violation is a bug in Eclipse, specifically in the class `Main` in the package `org.eclipse.equinox.launcher`, The two methods `setDescription()` and `getDescription()` are called concurrently while protected by different synchronization objects. The effect of this atomicity violation is cloning corrupted data. We have reported this bug to Eclipse Bugzilla and it is confirmed[5].

Let us now consider the 39 atomicity violations that DAPA found but AtomChase missed. Those atomicity violations were in TSP (1), ArrayList (1), Colt (1), Avrora (7), Tomcat (4), Batic (5), Eclipse (1), H2 (1), PMD (1), Sunflow (6), and Xalan (11). Penelope was the biggest contributor to that 39 atomicity violations, and found 26 of them. DoubleChecker found 10, and Atomizer revealed 3 of those atomicity violations. For example, DoubleChecker detected an atomicity violation in Eclipse that AtomChase missed.[6] The violation results in the following misbehavior: when multiple threads perform undo actions simultaneously, a user can get "error 404 (Not Found)".

Our experiments show that AtomChase finds the most atomicity violations, and that Penelope, DoubleChecker, and Atomizer remain worthwhile tools as each of them detect atomicity violations that the other tools cannot find.

[5] Bug ID 329143: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=329143`

[6] The atomicity violation happens in the class `CompositeArtifactRepository` in the package `org.eclipse.equinox.internal.p2.artifact.repository`.

	Hybrid	Dynamic									
benchmarks	HAVE	DoubleChecker	AtomFuzzer	Penelope	Atomizer	DAPA	AtomChase				
							total	= new	+ DAPA	EV	CV
Sor	3	0	0	0	0	0	0	0	0	0	0
TSP	28	3	1	1	0	4	6	3	3	5	4
Hedc	11	1	0	3	1	3	5	2	3	5	4
Elevator	18	1	0	0	0	1	3	2	1	3	3
ArrayList	9	7	1	1	0	7	6	0	6	5	4
TreeSet	4	0	1	1	0	1	1	0	1	1	1
HashSet	4	1	1	1	1	1	1	0	1	1	1
Vector	7	2	0	1	0	2	2	0	2	2	2
RayTracer	6	0	0	2	1	3	3	0	3	2	2
MolDyn	4	0	1	1	0	1	2	1	1	2	2
MonteCarlo	8	2	2	2	1	3	3	0	3	3	2
Derby	105	19	6	60	4	66	83	17	66	79	73
Colt	41	8	4	13	2	16	23	8	15	20	16
Avrora	50	10	7	38	9	43	41	5	36	38	34
Tomcat	116	49	53	64	43	66	77	15	62	73	65
Batic	44	39	10	34	16	40	36	1	35	33	29
Eclipse	131	73	41	49	66	81	121	41	80	116	112
FOP	76	20	8	24	12	26	28	2	26	27	25
H2	48	17	13	15	9	17	26	10	16	21	17
PMD	18	0	4	11	2	11	13	3	10	12	9
Sunflow	22	6	4	20	4	21	17	2	15	15	11
Xalan	51	24	22	49	20	49	40	2	38	36	31
TOTAL	804	282	179	390	191	462	537	114	423	499	447

Fig. 8. The number of three-access violations found in 22 benchmarks by 6 techniques. EV = Exhaustive Validation, CV = Condition Validation

Now we analyze the individual contributions of the tools with respect to each other. Our first observation is that: AtomFuzzer $\subseteq$ Penelope. In words, if AtomFuzzer finds an atomicity violation, then Penelope also detects that atomicity violation. Our second observation is that: Atomizer $\subseteq$ Penelope $\cup$ DoubleChecker. In other words, if Atomizer finds an atomicity violation, then either Penelope or DoubleChecker (or both) find that atomicity violation as well.

Timings. The geometric means of the execution times show that AtomFuzzer is the fastest while Penelope is the slowest, and AtomChase is almost twice as fast as Penelope. The longer execution time of AtomChase can be attributed to its iterative exploration. Our experiments show that by parallelizing the constraint solving process we can improve timing by **10-18%**.

Number of schedules. The number of calls to the directed execution module appears to be rather small: for every benchmark, it is less than four times the number of three-access violations. This result shows that the alternation of plan synthesis and execution can effectively steer the search towards the target trace in a modest number of iterations.

Number of steps of execution. Some schedules can be as long as 105 million events, which illustrates that the plan synthesis scales to long schedules. For each of the seven benchmarks (HashSet, Derby, Tomcat, Batic, Eclipse, Sunflow, Xalan), at least one atomicity violation happens in a trace that has more than a million events. Among the 160 atomicity violations found after at least a million steps, 96 were found only by AtomChase and among 12 atomicity violations that happen after 10 million steps 10 were found by AtomChase. AtomChase can replay the atomicity violations that it finds and hence, can report reproducible bugs. Finding atomicity violations after many steps of computation can be attributed to the ability of our iterative search to drive the execution towards the candidates.

Checking non-atomicity. The exhaustive checks show that 92% of the traces reported by AtomChase are real atomicity violations. This means that 7% of them are false positives i.e. match the three-access patterns but are in fact atomic. The non-atomicity sufficient conditions could recognize that 83% of the traces reported by AtomChase are real atomicity violations. Therefore, these conditions could recognize 89% of the real atomicity violations found by AtomChase. In addition, checking these conditions is more than three orders of magnitude faster than the exhaustive checks.

VI. Related Work

In section 1 we mentioned some of static, dynamic and predictive techniques for checking atomicity. In subsection V-B we discussed 5 techniques for atomicity violation detection. In this section, we discuss other techniques and tools in the area of atomicity violation detection.

Static analysis. Application of static techniques such as type systems [20], [18], model checking [16], [21], and petri nets [12] to atomicity has been investigated. In contrast to static analysis, our technique is an iterative dynamic analysis.

Dynamic Analysis. Avio [25] can extract atomicity invariants of the application from correct runs and detect the violation of those invariants in test runs. Velodrome [19] presents an efficient and scalable representation of the transactional happens-before relation and presents a dynamic atomicity violation detector that reports no false positives.

	Hybrid	Dynamic						
benchmarks	HAVE	DoubleChecker	AtomFuzzer	Penelope	Atomizer	AtomChase	EV	CV
Sor	9	6	3	1142	3	403	0	0
TSP	9	17	3	2674	2	497	0.5	8
Hedc	36	24	5	722	10	776	4.3	9
Elevator	43	22	2	3481	14	411	0.2	3
ArrayList	26	11	3	556	8	591	0.3	4
TreeSet	21	8	4	903	5	427	0.3	2
HashSet	22	8	4	119	5	478	0.2	2
Vector	10	9	2	202	4	485	0.1	1
RayTracer	33	14	6	5419	10	1563	0.5	3
MolDyn	28	24	5	2752	8	1971	0.7	5
MonteCarlo	47	19	11	2285	13	892	0.8	7
Derby	76	58	45	7803	57	1779	10.4	201
Colt	48	29	18	636	11	1066	3.3	26
Avrora	105	32	66	6787	73	2008	9.8	83
Tomcat	121	65	60	4687	55	2361	12.7	229
Batic	70	38	23	1322	28	851	4.7	90
Eclipse	129	57	52	7524	97	1490	15.9	418
FOP	54	21	19	1080	33	992	4.1	52
H2	36	27	14	1329	18	1347	5.6	49
PMD	68	43	28	4210	9	916	2.9	28
Sunflow	59	19	31	2648	41	1101	2.5	33
Xalan	67	60	25	3483	42	2042	6.2	95
geom. mean	39	22	10	1767	14	949	1.7	18

Fig. 9. The run time of the 6 techniques on the 22 benchmarks and the runtime of the exhaustive and sufficient condition checks on the three-access traces found by AtomChase. EV = Exhaustive Validation, CV = Condition Validation. Every column other than EV is in seconds and EV is in hours.

Predictive analysis. These techniques run the program under test and obtain a trace. Then they try to predict violations in alternative interleavings of the trace [30]. Similar to AtomChase, Penelope [31] runs the program and records execution constrains. However, Penelope does not collect synchronization constraints, while AtomChase does. More importantly, while Penelope explores only the interleavings of the given trace, AtomChase reruns the program to explore new paths towards the violation. Wang and Stoller [34], [33] present a runtime predictive analysis to predict atomicity violations in alternative schedules of a program run. They record the executed paths of the program as condition-guarded statements and incorporate the conditions in the constraints. Therefore, solutions to their constraints are executable schedules. However, the plan synthesis module of AtomChase does not necessarily generate executable plans and the executability of the plans is verified by the execution module. While their technique explores only the paths that were taken by the initial run, AtomChase can explore new paths towards atomicity violation. SideTrack [36] and JPredictor [6] are predictive atomicity violation detector tools. Similar to previous tools, they can predict violating executions from a given execution but does not report any false alarms.

VII. Limitations

Our approach has four main limitations. First, our approach relies on HAVE to produce three-access candidates. If HAVE misses an atomicity violation, so does AtomChase. Second, our approach relies on a constraint solver both in the plan synthesis and directed execution modules. The satisfiability of the form of constraints that we use in the plan synthesis module is a decidable problem, while the form of constraints that we use in the directed execution module are derived from expressions in the program text and may be undecidable. Thus, for constraint solving in directed execution, we are at the mercy of expressions in the program text and the power of the constraint solver. Third, AtomChase cannot filter benign atomicity violations. An atomicity violation in a program is benign if the correctness of the program is not affected by the atomicity violation. Fourth, our approach has no support for native code.

VIII. Conclusion

We presented an atomicity violation detection technique that directs execution towards three-access candidates by iterative application of directed concurrent execution and execution plan synthesis techniques. We implemented the technique in an automatic atomicity violation detector called AtomChase. Our experiments with a large benchmark suite of more than 4.5 million lines of code show that AtomChase increased the found three-access violations by 24% in comparison to the five previous atomicity violation detectors combined. We presented and proved sufficient conditions for non-atomicity of traces with the three-access pattern. The conditions could recognize the majority of 89% of the real atomicity violations found by AtomChasewhile they significantly expedite checking the non-atomicity of the found traces.

References

[1] S. Biswas, J. Huang, S. A., , and B. M. D. Doublechecker: Efficient sound and precise atomicity checking. In Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation, PLDI '14, 2014.

[2] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[5] F. Chen and G. Roşu. Parametric and sliced causality. In Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07, pages 240–253, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] F. Chen, T. F. Serbanuta, and G. Rosu. jpredictor: A predictive runtime analysis tool for java. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 221–230, New York, NY, USA, 2008. ACM.

[7] Q. Chen, L. Wang, Z. Yang, and S. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In M. Chechik and M. Wirsing, editors, Fundamental Approaches to Software Engineering, volume 5503 of Lecture Notes in Computer Science, pages 425–439. Springer Berlin Heidelberg, 2009.

[8] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, Computer-Aided Verification (CAV'2014), volume 8559 of Lecture Notes in Computer Science, pages 737–744. Springer, July 2014.

[9] Eslamimehr and Lesani. http://www.mah-d.com/issre15appendix.pdf, June 2015.

[10] M. Eslamimehr and J. Palsberg. Race directed scheduling of concurrent programs. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pages 301–314, New York, NY, USA, 2014. ACM.

[11] M. Eslamimehr and J. Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 353–365, New York, NY, USA, 2014. ACM.

[12] A. Farzan and P. Madhusudan. Causal atomicity. In T. Ball and R. Jones, editors, Computer Aided Verification, volume 4144 of Lecture Notes in Computer Science, pages 315–328. Springer Berlin Heidelberg, 2006.

[13] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In Proceedings of the 20th International Conference on Computer Aided Verification, CAV '08, pages 52–65, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,, TACAS '09, pages 155–169, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, pages 248–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] C. Flanagan. Verifying commit-atomicity using model-checking. In S. Graf and L. Mounier, editors, Model Checking Software, volume 2989 of Lecture Notes in Computer Science, pages 252–266. Springer Berlin Heidelberg, 2004.

[17] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[18] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. ACM Trans. Program. Lang. Syst., 30(4):20:1–20:53, Aug. 2008.

[19] 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 '08, pages 293–303, New York, NY, USA, 2008. ACM.

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.

[21] J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In B. Steffen and G. Levi, editors, Verification, Model Checking, and Abstract Interpretation, volume 2937 of Lecture Notes in Computer Science, pages 175–190. Springer Berlin Heidelberg, 2004.

[22] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer, 2(4):366–381, 2000.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[24] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[25] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[26] C. H. Papadimitriou. The serializability of concurrent database updates. J. ACM, 26(4):631–653, Oct. 1979.

[27] 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, SIGSOFT '08/FSE-16, pages 135–145, New York, NY, USA, 2008. ACM.

[28] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, ESOP '09, pages 394–409, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4):391–411, Nov. 1997.

[30] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on, pages 99–108, July 2011.

[31] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[32] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In D. Watt, editor, Compiler Construction, volume 1781 of Lecture Notes in Computer Science, pages 18–34. Springer Berlin Heidelberg, 2000.

[33] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pages 137–146, New York, NY, USA, 2006. ACM.

[34] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. IEEE Trans. Softw. Eng., 32(2):93–110, Feb. 2006.

[35] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 1–14, New York, NY, USA, 2005. ACM.

[36] J. Yi, C. Sadowski, and C. Flanagan. Sidetrack: Generalizing dynamic

atomicity analysis. In Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.