Dynamic Generation of Likely Invariants for Multithreaded Programs

Markus Kusano, Arijit Chattopadhyay, Chao Wang
Department of ECE, Virginia Tech
Blacksburg, Virginia, USA

***Abstract*—We propose a new method for dynamically generating likely invariants from multithreaded programs. While existing invariant generation tools work well on sequential programs, they are ineffective at reasoning about multithreaded programs both in terms of the number of real invariants generated and in terms of their usefulness in helping programmers. We address this issue by developing a new dynamic invariant generator consisting of an LLVM based code instrumentation front end, a systematic thread interleaving explorer, and a customized invariant inference engine. We show that efficient interleaving exploration strategies can be used to generate a diversified set of executions with little runtime overhead. Furthermore, we show that focusing on a small subset of thread-local transition invariants is often sufficient for reasoning about the concurrency behavior of programs. We have evaluated our new method on a set of open-source multithreaded C/C++ benchmarks. Our experiments show that our method can generate invariants that are significantly higher in quality than the previous state-of-the-art.**

I. Introduction

Methods for dynamically generating likely invariants from sequential software have been used in many applications including program understanding, maintenance, testing/verification, and error diagnosis. However, effective tools for generating such invariants for concurrent software are still lacking. For example, *Daikon* [1], [2] is a highly successful invariant generation tool for sequential programs written in languages such as Java, C, C++, and Perl. However, as we will show in Section II, for multithreaded programs, *Daikon* often produces many confusing and incorrect results.

One problem of existing methods such as *Daikon* is that they depend heavily on the set of execution traces fed to the invariant inference engine. In general, increasing the amount of program behavior exercised in the set of execution traces increases the likelihood that the generated invariants are true. However, generating a sufficiently diversified set of execution traces is difficult for a multithreaded program since the program's behavior depends not only on the program's input but also on the thread's schedules. Thread scheduling, in a typical execution environment, is determined by the underlying operating system and the threading library; naively executing the program many times does not necessarily increase the diversity of the thread schedules.

Another problem of existing methods is that they tend to report too many invariants. Even if many of these reported invariants are real invariants, they are unlikely to be equally useful. It is impractical to assume that the user will have time to sift through all of them individually.

Multithreaded programs, due to the subtle interactions amongst threads and potentially large number of interleavings, pose challenges both in their design and analysis. Typically, when the focus of an analysis is on concurrent nondeterminism, one assumes the sequential part of the computation is correct: instead, the problem comes from rare and complex thread interactions. In such cases, we argue, that the focus should be put on a small subset of thread-local invariants, called the *transition invariants*, that capture the relations among shared variables directly related to concurrency control. For example, a certain code block should be kept atomic, or instances of certain operations should be made mutually exclusive.

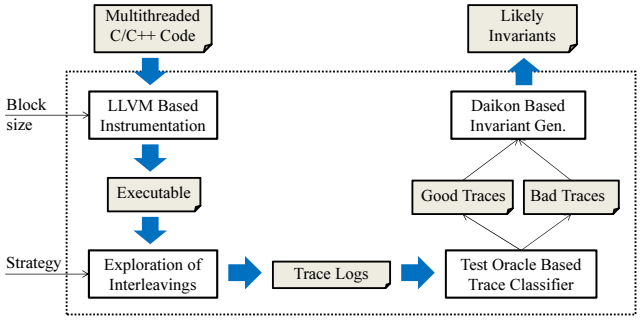

Figure 1. Our new dynamic invariant generation tool named *Udon*.

To this end, we propose a new dynamic invariant generation method for multithreaded programs. By leveraging systematic thread interleaving exploration algorithms to generate diversified execution traces, our method significantly improves the quality of generated invariants over existing methods.

The overall flow of our method is shown in Figure 1. Our method takes a multithreaded C/C++ program as input and returns likely invariants as output. First, we instrument the code using a new LLVM based front end to add monitoring capabilities for dynamic analysis. Then, we execute the program under the control of a systematic interleaving explorer. The generated traces are fed to a classifier which separates the passing traces from the failing traces. Finally, we feed a subset of the traces to a customized *Daikon* invariant inference engine which returns the likely invariants as output.

Another contribution of our work is investigating the impact of different interleaving exploration strategies on the precision and performance of invariant generation. In theory, all possible thread interleavings of a concurrent program should be given to the invariant inference engine to obtain the most precise invariants. However, due to the well-known interleaving explosion problem, the number of thread interleavings can be exponential with respect to the number of concurrent opera-

```
Thread 1                          Thread 2
1  p = &A;                        6  p = NULL;
2  if(p!=NULL){                   7  ...
3    ...                          8  ...
4    p->x += 10;                  9  ...
5    ...                          10 ...
6  }                              11 ...
7                                 12
```

Figure 2. Two threads sharing a pointer. In thread 1, the `if` statement and the subsequent write to `p` is intended to be atomic.

tions. Therefore, we propose the use of *selective exploration* strategies, as opposed to exhaustive exploration, to reduce runtime overhead. Our experimental evaluation shows that selective exploration often leads to invariants of similar quality to sound reduction methods, such as dynamic partial order reduction [3], but with an order-of-magnitude faster run time.

A third contribution of our work is a focus on generating invariants most relevant to concurrency related program behaviors. In general, the invariants generated by an inference engine fall into two categories: state invariants and transition invariants. For example, consider the program in Figure 2, where `A.x` is initialized to zero. The predicate `A.x == 10` at Line 4 is a state invariant—it holds at a thread-local program location and is expressed in terms of variables visible at that location. A transition invariant, in contrast, is a predicate that may hold at the entry and exit points of a code block and is expressed in terms of two versions of a program variable at the entry and exit points. For example, in Figure 2, the predicate `A.x == orig(A.x) + 10` is a transition invariant over the code block from Line 1 to Line 4, where `orig(A.x)` is the value of `A.x` at Line 1 (the original value) and `A.x` is the value of `A.x` at Line 4.

For reasoning about concurrency behaviors, we argue that it is often sufficient to focus on these transition invariants. The reason is that they capture the *no-thread-interference* properties, i.e., whether the associated code block is *atomic* or whether it *should be made atomic*. By atomic, we mean that the execution of the code block is not affected by the execution of other instructions from concurrently running threads.

Consider Figure 2. A bug can occur if Thread 2 sets the value of `p` to `null` when Thread 1 is executing Lines 2–6. If we generate invariants only from the non-buggy runs, we will observe the transition invariant `p == orig(p)` for the code block from Line 2 to Line 6. Examining the buggy runs, we will see that this invariant no longer holds. The difference in the invariants generated between the passing and failing runs shows the root cause of the bug: `p` is not constant.

Throughout this paper, we will show how discrepancies between the invariants generated from passing and failing runs, like this example, can be leveraged to understand the software code and diagnose concurrency bugs.

We have implemented our new methods and conducted experiments on a set of open-source multithreaded C/C++ programs. Our results show that the invariants generated by our new method are often significantly higher in quality than the previous state-of-the-art *Daikon*. Overall, this paper makes the following contributions:

```
1  int getBalance() {
2    int bal;
3    Lock();
4    bal = balance;
5    Unlock();
6    return bal;
7  }
8  void setBalance(int bal) {
9    Lock();
10   balance = bal;
11   Unlock();
12 }
13 void withdraw() {
14   int bal = getBalance();
15   bal = bal -100;
16   setBalance(bal);
17 }
```

Figure 3. Concurrent bank account example.

- We show, through experiments, that existing dynamic invariant generation tools such as *Daikon* do not work well on multithreaded programs, both in terms of the number of true invariants generated and in terms of the usefulness of these invariants.
- We propose a new method for improving the quality of dynamic invariant generation for multithreaded programs by leveraging selective interleaving exploration strategies.
- We show that transition invariants are the most relevant invariants to help in reasoning about concurrency related behaviors. They are useful in program comprehension and diagnosing concurrency errors.
- We implement the proposed method and demonstrate its efficiency and effectiveness through experiments on a set of multithreaded C/C++ benchmarks.

The remainder of this paper is organized as follows. We present examples to illustrate our new methods in Section II. We establish notation in Section III and then present our new invariant generation algorithm in Section IV. We present both runtime optimizations and methods to clarify output to the user in Section V. This is followed by our experimental evaluation in Section VI. We review related work in Section VII and finally give our conclusions in Section VIII.

II. Motivating Examples

In this section, we present examples to illustrate the problems in existing methods, highlight our main contributions, and demonstrate some potential use cases for our new method.

First, consider the program in Figure 3, which has a global variable named `balance` being accessed by functions `getBalance()` and `setBalance()`. The third function, `withdraw()`, invokes the previous two functions to deduct a certain amount from `balance`. Since the global variable `balance` is protected by a `Lock()` and `Unlock()` pair every time it is accessed there is no data-race. However, there can be atomicity violations. The function `withdraw()` is meant to be executed atomically—without other threads interleaved between the calls to `getBalance()` and `setBalance()`—but the atomicity is not enforced. For example, starting with `balance=400`, if two concurrent threads run `withdraw()` at the same time, the result may be either `300` or `200`.

Existing invariant generation tools do not work well in this case. For example, if we run the program with *Daikon's* C

front end, most likely we will get a false invariant. The reason is that *Daikon* relies on the native execution environment to determine the program's thread schedule at run time, and in this example, since the code in each thread is small—significantly smaller than what can be executed in the Linux kernel's time slice—all threads will have ample time to run to completion before encountering a context switch.

If the erroneous interleaving never occurs during its analysis, *Daikon* would report the following false transition invariant for the `withdraw()` function: `balance = orig(balance) - 100`, where `orig(balance)` denotes the original value of `balance` at the entry point of `withdraw()` and `balance` denotes the value of `balance` at the exit point. This is not a true invariant, and reporting it to the user may do more harm than good. That is, it can make the developer believe that `withdraw()` is atomic, thereby masking the concurrency bug.

Our new method, in contrast, controls the thread scheduling of the program in order to create a diverse and representative set of execution traces. Consequently, the invariant inference engine would produce the following correct invariant for the `withdraw()` function: `balance < orig(balance)`. This is the correct result and is the best one can infer from the executions of this program (two threads running `withdraw()` concurrently). That is, the `balance` always decreases but not necessarily by `100`.

Another problem with existing tools is that they often report too many invariants. For example, running *Daikon* on the benchmark `FibBench` [4], which has 55 lines of code, would generate 24 likely invariants. Among them, 15 are true invariants (the rest are false), but only three of them are related to the concurrency behaviors of the threads. The others are either specific to the particular program input used in the test runs or the sequential part of the computation. Since our goal is to reason about concurrency related behaviors, our new method allows the user to focus only on the concurrency related invariants, known as *transition invariants*.

There can be many applications for transition invariants such as `balance = orig(balance) - 100` and `balance < orig(balance)`. Here, we give two examples: to help diagnose concurrency bugs and to infer atomic code regions.

During software testing, it is reasonable to expect the user to provide a test oracle which separates failing test runs from passing test runs. We allow users of our new tool to specify correctness conditions using `R_assert()` which, from the user's perspective, is identical to the standard C `assert()` function. For our running example, assume the user asserts that `balance==200` must hold at the end of the execution. This is illustrated in Figure 4. For the buggy program in Figure 3, if the function `withdraw()` is executed atomically by both threads, the assertion would pass; but, if the function is not executed atomically, the assertion would fail.

If we run our new invariant generation method on the passing traces only, it would report the transition invariant `balance = orig(balance) - 100`. In contrast, if we run our new invariant generation method on the failing traces only, it would report the transition invariant `balance < orig(balance)`. The discrepancy between these two sets of results (from passing and failing runs) will help the user diagnose the root cause of the concurrency failure.

```
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  R_assert(balance==200);  // test oracle
}
```

Figure 4. The `main()` function for the example in Figure 3.

```
void withdraw() {
  Lock();
  int bal = getBalance();
  bal = bal - 100;
  setBalance(bal);
  Unlock();
}
```

Figure 5. Enforcing atomicity in the `withdraw()` function in Figure 3.

Regarding atomic region inference, consider the same example in Figure 3. The transition invariant generated from the passing runs for the code block spanning Lines 13–17 is `balance = orig(balance) - 100`, which is consistent with the thread-local transition relation of this code block when it is executed without interference from other threads. In other words, the programmer's design intent, as revealed by all the passing test runs, is that `withdraw()` should be executed atomically. This suggests that to fix the bug in `withdraw()`, we need to enforce atomicity around the calls to `getBalance()` and `setBalance()` as illustrated in Figure 5.

```
1  typedef struct { int a, b, c;} Data;
2  Data *A[128] ;
3  Data *p = A[0];
4  void thr1() {
5    Data *tmp = p;
6    if (tmp != NULL) {
7      p->a = 100;
8      p->b = 200;
9      p->c = 300;
10     R_assert(tmp->a == 100 && tmp->b == 200
11         && tmp->c == 300 );
12   }
13 }
14 void thr2() {
15   p =  A[rand() % 127];
16 }
```

Figure 6. Concurrent program where one thread updates the fields of a structure while the other modifies a global pointer.

Another limitation with *Daikon*, due to the location of code instrumentation, is that invariants are only generated at function entry and exit points. Although this design choice is suitable for sequential code, it is not suitable for concurrent programs because concurrency constructs, such as atomic code regions, rarely coincide with the procedural boundaries. Our method, in contrast, has the capability of generating invariants at the boundary of code blocks of arbitrary size—the user can set the block size as input to our tool as shown in Figure 1. We illustrate this feature using the following example.

```
=======================================
..main.c_3_9():::ENTER
::p != NULL
=======================================
..main.c_3_9():::EXIT
::p == orig(::p)
=======================================
..main.c_5_9():::ENTER
::p != NULL
=======================================
..main.c_5_9():::EXIT
::p == orig(::p)
=======================================
```

Figure 7. Portion of the output of our tool for the example in Figure 6.

Consider the program in Figure 6. Within `thr1()`, the three fields are intended to be updated atomically; similar to the previous example, the programmer asserts the correctness condition using `R_assert()`. To infer the *intended* atomic region that spans from Line 6 to Line 12, the capability of generating invariants for arbitrary code blocks is crucial.

Figure 7 shows a section of our tool's output regarding the transition invariant generated from passing runs. It first starts with a code block size of two and then iteratively expands the block size. A block size of two means that our tool will attempt to generate invariants over any code region containing two consecutive accesses to a shared object. The partitioning of the source code into code regions was performed by our LLVM based instrumentation front end as shown in Figure 1. In Figure 7, `..main.c_6_12()` means that we consider the block from Line 6 to Line 12 in Figure 6, whereas `..main.c_8_12()` means that we consider the block from Line 8 to Line 12. With a block spanning Lines 6–12 we cover all the shared memory accesses in `thr1()`. In both cases, when analyzing the passing runs, we can generate the invariant `p==orig(p)`, which indicates that the value of `p` is never changed. Amongst all the failing runs, this invariant does not hold. Again, the discrepancies in the invariants generated by the passing and failing runs correctly suggests that in order for the test runs to pass, the code block from Line 6 to Line 12 must be kept atomic.

III. Preliminaries

In this section, we present a formal model for concurrent programs, and introduce the basics of the dynamic invariant generation process.

A. Concurrent Programs

A multithreaded program consists of a set of *shared* variables, and a set of threads $\{ T_1, \ldots, T_n \}$ where n is the number of threads in the program. Each thread is a sequential program with a set of *thread-local* variables. Let st be an instruction. An execution instance of st is called an *event*, denoted $e = \langle tid, l, st, l' \rangle$, where tid is the thread ID, and l and l' are the thread program locations before and after executing st. An event is said to be *visible* if it accesses a shared variable or a thread synchronization object (mutex lock or condition variable). Otherwise, the event accesses only thread-local variables and it is said to be *invisible*. During systematic interleaving exploration and execution trace logging, invisible events will be ignored.

We model each thread T_i as a state transition system M_i. The entire transition system, denoted $M = M_1 \times M_2 \times \ldots \times M_n$, is constructed via interleaving composition. Let $M = \langle S, R, s_0 \rangle$, where S is the set of global states, R is the set of transitions, and s_0 is the initial state. Each state $s \in S$ is a n-tuple of thread states. Each transition $e \in R$ is an event from one of the n threads. An execution trace of M is a sequence $\rho = s_0 \xrightarrow{e_0} s_1 \ldots \xrightarrow{e_{n-1}} s_n$, where $s_0 \xrightarrow{e_0} s_1$ corresponds to executing event e_1 in state s_0 leading to state s_1.

We use the special event **halt** to denote normal program termination, and **abort** to denote faulty program termination, which corresponds to a failed `R_assert()` statement. An event from thread T_i may have the following types:

- **halt**, which denotes the normal program termination;
- **abort**, which denotes the faulty program termination;
- *fork(j)* for creating child thread T_j, where $j \neq i$;
- *join(j)* for joining back thread T_j, where $j \neq i$.
- *lock(lk)* for acquiring lock lk;
- *unlock(lk)* for releasing lock lk;
- *signal(cv)* for setting signal on condition variable cv;
- *wait(cv)* for receiving signal on condition variable cv;
- *read(v)* for reading of shared variable v;
- *write(v)* for writing to shared variable v;
- *mEntry(m)* for entering a function call;
- *mExit(m)* for returning from a function call;
- *bEntry(blk)* for starting a code block;
- *bExit(blk)* for ending a code block.

Here, *mEntry()* and *mExit()* are also supported by existing invariant generation tools such as *Daikon*, whereas *bEntry()* and *bExit()* are the new additions in our method.

We model thread synchronization events in our method in order to control the execution order during thread interleaving exploration. Using this model, at each moment during a program's execution, we know which threads are blocked (*disabled*) and which threads are executing (*enabled*)

An enabled thread becomes disabled if (i) it attempts to execute *lock(lk)* while lk is held by another thread; (ii) it attempts to execute *wait(cv)* while the signal on cv has not yet been set; or (iii) it attempts to execute *join(j)* while the child thread T_j is still running. Similarly, a disabled thread becomes enabled if (i) another thread releases the lock lk by executing *unlock(lk)*, (ii) another thread sets the condition variable cv by executing *signal(cv)*, or (iii) the child thread T_j terminates. An accurate model of the sets of enabled and disabled threads at runtime is required since attempts to schedule disabled threads while postponing the execution of enabled threads may lead to artificial deadlocks.

At runtime, our scheduler selects a given event from the set of enabled events. Which event to select is determined by the exploration strategy used by the scheduler. Similarly, the scheduler repeatedly executes the program, systematically exploring new interleavings, until the search strategy's interleaving coverage criteria is satisfied. We defer discussions of exploration strategies until Section IV. For an in-depth discussion on systematic concurrent program testing refer to [5], [6], [7].

B. Dynamic Invariant Generation

Dynamically generated invariants are predicates that hold over the execution traces produced by test runs. As such, they are not guaranteed to hold for all possible executions of the program. Furthermore, the invariant inference engine often uses a statistical analysis and, in theory, is neither sound nor complete. However, in practice, dynamic invariant generation tools such as Daikon have shown to be useful in a wide range of applications. In general, the number of invariants that can be generated, as well as the likelihood of them being true invariants, depends on the test suite.

Daikon [1], [2] is a highly successful invariant discovery tool that supports programming languages such as C, C++, C#, Eiffel, F#, Java, Lisp, and Visual Basic. For each of these languages, *Daikon* provides a front end tool for code instrumentation to add logging capability to the target program. The front end tools prepare the program to generate trace logs in a common format, which are then fed to the same back end invariant inference engine.

We have developed a new instrumentation tool based on the popular LLVM compiler platform to replace *Daikon's* previous front end. The main advantage of our new instrumentation tool is to leverage the large number of static program analysis procedures implemented in LLVM as well as to reduce the runtime overhead caused by instrumentation. We will show through experiments (Section VI) that our LLVM based instrumentation tool can indeed lead to faster dynamic analysis compared to the default front end in *Daikon* due to its significantly lower instrumentation overhead.

Since *Daikon* cannot diversify the thread schedules, it may generate many bogus invariants for a multithreaded program. Furthermore, *Daikon* is effective in generating linear invariants of the form ($ax + by < c$), but weak in generating more complex invariants such as polynomial invariants ($c_0 + c_1x^1 + c_2x^2 + ... < 0$) and array invariants. For the latest development along this line, please refer to the recent work by Nguyen *et al.* [8], [9]. However, our focus is not on improving the expressiveness of the generated invariants, but on improving their quality with respect to concurrency. The vast majority of invariants generated by existing tools such as *Daikon* capture the sequential program behavior. Our new method, in contrast, focuses on invariants that capture the concurrency behaviors.

IV. Udon: Our New Dynamic Invariant Generation Tool

In this section, we present the three components of our new method: an LLVM based code instrumentation tool, a systematic interleaving exploration tool, and a customized inference engine for *Daikon*. The overall flow of our tool, called *Udon*, is illustrated in Algorithm 1, which takes the source code of a multithreaded C/C++ program as input and returns a set of likely invariants as output.

A. LLVM Based Code Instrumentation

We developed an LLVM based front end for instrumenting multithreaded C/C++ code. As shown in Algorithm 1, it consists of three code transformation passes.

The first pass, `inspect_pass()`, takes C/C++ code as input and returns an instrumented version of the code as output.

Algorithm 1 High-level algorithm for *Udon*

$inst\_output \leftarrow$ inspect_pass(src_code)
$inst\_output \leftarrow$ daikon_pass($inst\_output$)
$inst\_output \leftarrow$ spacer_pass($inst\_output, spacer\_size$)
$trace\_file \leftarrow$ gen_traces($inst\_output$)
$thrd\_mod\_traces \leftarrow$ trace_classifier($trace\_file$)
$invariants \leftarrow$ inv_inference($thrd\_mod\_traces$)

Inside this pass, we first identify all the program points where a thread's schedule needs to be controlled. These program points include the calls to thread synchronization routines, and the read and write operations on shared memory locations discussed in Section III. At each program point, we inject new code before these *visible* operations to allow the control of the thread at run time by the scheduler. We leverage the conservative static analysis techniques implemented in LLVM to identify these visible operations.

The second pass, `daikon_pass()`, takes the previously instrumented code as input and returns another instrumented version of the code as output. Inside this pass, we inject new code to add event trace generation capabilities to the program. The event trace generated by the program conforms to the common file format as required by the back end invariant inference engine in *Daikon* [2]. By default, this pass instruments the code only at the function entry and exit points, which is comparable to the original C/++ front end for *Daikon*.

The third pass, `spacer_pass()`, takes the previously instrumented code as input and returns the final version of the code as output. Inside this pass, we also inject new code to add logging capabilities not just at the procedural boundaries, but also at the boundary of arbitrary code blocks. This is accomplished by inserting hook function calls to the entry and exit points of these code blocks, which in turn take care of the trace generation at run time.

Note that the events logged as a result of the second and third passes are kept in the same format. From the standpoint of the back end invariant inference engine, there is no distinction between a pair of entry and exit points for a function, and a pair of entry and exit points for an arbitrary code block. Therefore, the back end inference engine does not have to be drastically altered in order to infer invariants at the boundary of arbitrary code blocks. By varying the size of the instrumented code blocks, we can dynamically change the locations where state and transition invariants are generated. This will help us to detect likely atomic regions in the code.

B. Systematic Interleaving Exploration

The `gen_traces()` function in Algorithm 1 involves an exploration of the concurrent state space of the program. Due to the well-known interleaving explosion, in general, we cannot afford to naively enumerate all possible thread schedules while diversifying the execution traces for the back end inference engine. In this work, we build off a set of interleaving exploration strategies to produce a representative subset of thread interleavings.

The baseline search strategy relies on the theory of partial order reduction (POR) [5]. It groups the possible interleavings of a concurrent program into equivalence classes, and then selects one representative interleaving from each equivalence class to

explore. Equivalence classes are defined using Mazurkiewicz traces [10]. Two sequences of events are said to be in the same equivalence class if we can create one sequence from the other by successively permuting adjacent and *independent* events. Two events are *dependent* if they are from two different threads, access the same memory location, and at least one of them is a write or modify operation; otherwise, the two events are *independent*. It has been shown [11] that in the context of verifying concurrent systems, exploring one representative interleaving from each equivalence class is sufficient to catch all deadlocks and assertion violations.

One benefit of POR methods is that they are *sound* in that the reduced set of interleavings still capture all possible program behaviors. Therefore, using the explored interleavings as input for the subsequent invariant inference will lead to the best possible result; the explored interleavings form a *maximally diversified set* of execution traces.

The current state-of-the-art POR based algorithm is dynamic partial order reduction (DPOR) [3]. DPOR computes the dependency relation among events dynamically at run, as opposed to statically at compile time. As a result, DPOR proved to be a practical step forward for POR algorithms. It allowed for realistic programs written in full fledged programming languages such as C/C++ to be verified.

However, due to its exhaustive exploration of the search space, even DPOR may incur a large runtime overhead. As a result, there is a large body of work dedicated to the development of more efficient, yet unsound, exploration strategies. Two methods along these lines are preemptive context bounding (PCB) [6], and history-aware predecessor sets (HaPSet) [7]. In practice, even though they are unsound (i.e., they do not guarantee to find all concurrency bugs), empirical studies have shown that they still provide decent bug coverage in programs far too complex for DPOR to handle.

With respect to invariant generation, we will address performance concerns in Section V. We will show that replacing the ideal, yet slow, exhaustive search of DPOR with faster selective exploration strategies such as PCB and HaPSet, we can still maintain the quality of generated invariants while significantly reducing the execution time.

C. Customized Invariant Generation

The final two steps in Algorithm 1, denoted by `trace_classifier()` and `inv_inference()`, respectively, solve two issues in previous invariant inference engines.

First, the inference engine may produce confusing results when the event trace from both passing and failing executions are simultaneously used as input. To solve this problem, we developed a trace classification module that separates the execution traces into two groups based on a test oracle. Specifically, *Udon* provides a function called `R_assert()` through which the programmers can assert correctness conditions for the program. During dynamic analysis, if the assertion fails the corresponding execution trace will be classified as failing. Otherwise, the trace is passing. This behavior, from the user perspective, is identical to the standard C `assert()` function. While inferring likely invariants, the engine may choose to consider only the bad traces, only the good traces, or all traces. As a result, we can compare and contrast the invariants generated in these three scenarios.

Second, we customized the invariant generation engine in *Daikon* to focus on two types of invariants in a multithreaded program: the state invariants and the transition invariants. Both transition and state invariants are expressed in terms of *shared* variables – variables accessed by multiple threads in the execution traces. A state invariant is a predicate expressed in terms of the value of variables at a single program location. A transition invariant is a predicate expressed in terms of variable values at two different program locations of the same thread. In essence, a transition invariant is capable of capturing the *non-interference* impact of executing an arbitrary code block.

More formally, we consider a program P as a state transition system $P = \langle S, R, I \rangle$, where S is the set of states, $I \subseteq S$ is the set of possible initial states, and $R \subseteq S \times S$ is the transition relation. In general, a transition invariant [12], denoted T, is an over-approximation R^+ of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$. Intuitively, a transition invariant summarizes the relation between the pre- and post-conditions of a consecutive set of instructions (transitions) executed by a thread.

Transition invariants are particularly useful in software verification. To verify the concurrent behavior of a program, one typically assumes the sequential computation is correct but the thread interaction is potentially buggy. In this case, transition invariants would conform to the transition relation of a sequential code block in the absence of unexpected thread interference, but would deviate from the transition relation in the presence of thread interference. Therefore, observing that a sequential transition invariant differs from the concurrent transition invariant for the same code block is often indicative of a bug caused by thread interference.

For example, consider a shared counter that is incremented by multiple threads. The sequential (or correct) invariant for the increment operation would be that the counter increases its value by one at a time (`counter = orig(counter) + 1`). However, if the programmer fails to enforce atomicity in the increment operation, this invariant would no longer hold for all possible executions of the program. Clearly, the discrepancy between the sequential (or correct) and concurrent (or incorrect) transition invariants hints at the root cause of the aforementioned bug.

Due to the help of both systematic interleaving exploration techniques and customized invariant inference engines, *Udon* can more robustly generate high-quality invariants for multithreaded applications than existing methods.

V. Optimizations

The method presented in the previous section addresses the problem of dynamically generating high-quality invariants from multithreaded programs. However, using DPOR may cause a performance bottleneck due to the exponential growth in the number of interleavings with respect to program size. Another problem of the new method is that the number of reported invariants can still be very large. Despite that many of them are indeed invariants, reporting all of them without filtering can overwhelm the user. In this section, we present our solutions to these problems.

A. Exploring Interleavings Selectively

We address the performance problem by replacing the exhaustive DPOR exploration strategy with efficient, but unsound, selective search strategies. In this context, our goal is to drastically reduce the runtime overhead while maintaining the diversity of the generated interleavings. DPOR is a sound reduction in that it can prune away redundant interleavings without missing any concurrency related program behavior. To this end, it groups all possible thread interleavings into equivalence classes and then tries to explore only one representative interleaving from each equivalence class. However, due to the limited amount of program information available at run time, DPOR still may create many redundant equivalence classes for the purpose of generating invariants.

Consider the busy-waiting example in Figure 8, which has two threads T_1, T_2 communicating via the variable x ($x = 1$ initially). Under DPOR, the systematic exploration would generate infinitely many interleavings. Each interleaving corresponds to a different execution of the loop by the first thread. Each of these interleavings belongs to a separate equivalence class (since dependent memory locations are being updated) so each must be tested. Notice that, except for the first two interleavings, denoted $c(ab)$ and $(ab)c(ab)$, respectively, none of the other interleavings of the form $(ab)^k c(ab)$, where $k = 2, 3, ...$, can offer new concurrency scenarios.

In this paper, we propose to avoid generating an excessive number of execution traces during interleaving exploration by using a *selective search*, as opposed to an *exhaustive search*. The aim of a selective search is to cover a small subset of *high-risk* concurrency scenarios, while avoiding the redundant ones as shown in the example in Figure 8. The rationale is that, in practice, programmers often make implicit assumptions regarding the concurrency control of threads, e.g., certain code blocks should be mutually exclusive, certain code blocks should be atomic, and a certain operation order should be obeyed. Concurrency bugs are frequently the result of these implicit assumptions being broken, leading to data races, atomicity violations, and order violations. The goal of a selective search is to maximize the coverage of such scenarios while reducing the runtime overhead.

One popular selective search strategy proposed in the context of software testing is preemptive context bounding (PCB) [6]. PCB explores interleavings with only a bounded number of involuntary context switches. The strategy can be effective in concurrency testing because, in practice, many concurrency bugs can be exposed using a small number of context switches. We note that although PCB is effective in practice, the number of explored interleavings remains exponential with respect to the number of concurrent threads. Furthermore, it is not effective on the example in Figure 8, where all interleavings have exactly one context switch.

Another popular, and more scalable, selective search strategy is the history-aware predecessor set (HaPSet) [7] based reduction. It can be viewed as an improvement over PCB since the number of explored interleavings is no longer exponential with respect to the number of concurrent operations or threads. This is accomplished by focusing on covering only the ordering combinations of read/write instructions in the program, as opposed to the many *instances* of these instructions.

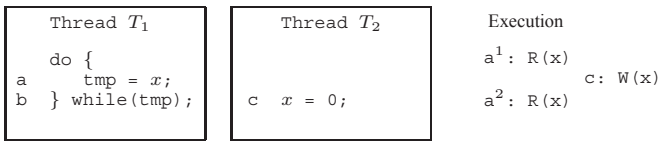

Figure 8. Using HaPSet reduction to prune interleavings ($x = 1$ initially).

Formally, a program statement st is defined as a tuple $(file, line, thr, ctx)$, where $file$ is the file name, $line$ is the line number, thr is the thread ID, and ctx is the bounded calling context. Given a set $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$ of interleavings and a statement $st \in Stmt$ that accesses a shared object, the History-aware Predecessor Set, or $\mathsf{HaPSet}[st]$, is defined as the set $\{st_1, \ldots, st_k\}$ of statements such that, for all $i : 1 \leq i \leq k$, an event e produced by st is immediately dependent upon an event e_i produced by st_i in some interleaving $\rho \in \mathcal{T}$.

Consider again the example in Figure 8. After exploring the first two interleavings, denoted $c(ab)$ and $(ab)c(ab)$, respectively, the HaPSets computed over these interleavings are as follows: $\mathsf{HaPSet}[a] = \{c\}$ and $\mathsf{HaPSet}[c] = \{a\}$. Since a and c are the only two conflicting program statements in the program, all possible HaPSet combinations have already been covered. Therefore, the interleaving exploration stops since no other interleaving can lead to new HaPSet scenarios.

We shall show through experiments in Section VI that faster interleaving exploration algorithms such as PCB and HaPSet are often as good as DPOR in terms of generating high-quality invariants. At the same time, these algorithms can be orders-of-magnitude faster, which makes *Udon* practically useful.

B. Focusing on Transition Invariants

By default, the number of invariants generated by our method—as well as other similar tools such as *Daikon*—can be large. However, not all of these invariants are useful for reasoning about the concurrency related program behaviors. For example, among the 22 likely invariants generated for *FibBench* [4] by *Daikon*, only three are related to concurrency, whereas the others are specific to the sequential part of the computation. Therefore, in this work, we propose to focus on only the transition invariants over shared variables. In the remainder of this section, we show why transition invariants are useful in helping the user understand the software code and diagnose concurrency bugs.

Let us assume that a multithreaded program has some assertions that would be satisfied in most cases, but may fail in some rare interleavings. Furthermore, regarding the concurrency control, there is no formal documentation other than the source code that describes the programmer's intent. In this case, we classify the execution traces of the program into two groups: the passing traces and the failing traces. To help the user understand the root cause of the concurrency error manifested in the failing traces, we leverage the two sets of invariants generated by *Udon* from the set of passing and failing traces and identify the discrepancy between them.

Formally, let I_p and I_f be the likely invariants generated from the passing and failing traces, respectively. As a result,

```
1 void thread() {
2   sum = sum + 1;
3 }
4 int main(){
5   // create, run, and join (NUM) threads...
6   R_assert (sum==NUM);
7 }
```

Figure 9. Concurrent program with multiple threads updating a counter.

$I_d = I_p \setminus I_f$ consists of all the invariants satisfied by the passing but not the failing traces. Our conjecture is that the discrepancy often provides information to help understand why the error occurs. In the following example, we show that I_d can indeed help a programmer understand the root cause.

Consider Figure 9, where a parameterized number of threads share a global counter `sum` initialized to zero. `NUM` is the number of threads that execute the function `thread()` concurrently. These threads are created, run, and joined inside `main()`, before it checks the value of `sum`. The test oracle provided by the programmer is shown on Line 6, which states that the expected result is `sum==NUM`. The assertion passes in runs where each thread executes the function `thread()` atomically, but fails in runs where the threads interfere with each other. Specifically, depending on how they interfere with each other, the value for `sum` ranges from `1` to `NUM`.

When given the passing traces, *Udon* will generate the transition invariant `sum == orig(sum) + 1` for the `inc()` function. However, when given the bad traces *Udon* will generate the transition invariant `sum > orig(sum)`, which covers cases in which `sum` is increased by $2, 3, ...,$ NUM. By comparing the two sets of transition invariants, we can see the difference in behavior of the passing and failing runs. In the passing runs, `sum` is always incremented, whereas in the failing runs, it is not.

Another possible application of transition invariants is to help the user identify atomic regions. When the transition relation of a code region is consistent with the transition invariant generated for the same code region, we say that section has been executed atomically. Furthermore, if the transition invariant is generated from passing traces only—and they are not satisfied by the bad traces—we can assume that the code region is *intended to be atomic*. For example, the function `thread()` in Figure 9 and the function `withdraw()` in Figure 3 are intended to be atomic, whereas only the fixed version of `withdraw()` in Figure 5 is atomic. The atomic code regions inferred in this way can help the user comprehend the software code and diagnose failed execution traces.

VI. Experiments

We have implemented the proposed method in a software tool called *Udon*. *Udon* can handle unmodified C/C++ code written using POSIX threads to automatically generate dynamic invariants. We used LLVM to create a new front end for *Daikon* [2] and a modified version of *Inspect* [13], [14] for systematic concurrent program exploration.

We evaluated *Udon* on 19 open source programs. The first set of programs come from the 2014 Software Verification Competition [4]. These programs, while small in terms of lines of code, implement complex low-level synchronization algorithms such as Peterson's [15] and Dekker's [16] solutions to the mutual exclusion problem. The second set of programs are real-world applications: `pfscan` is a parallel directory scanner and `nbds` [17] is a C implementation of several non-blocking concurrent data structures. All tests were run on a machine with a 2.60 GHz Intel Core i5-3230M CPU and 8 GB of RAM running a 64-bit Linux OS.

Our experimental evaluation was designed to answer the following research questions:

- Can the previous state-of-the-art method, *Daikon* [2], generate correct invariants for concurrent programs?
- Can our new method, *Udon*, robustly generate high-quality invariants for concurrent programs?
- Can *Udon* scale to programs of realistic size and complexity?

A. Results

Table I shows the results of an experiment comparing the performance of *Udon* against *Daikon* [2]. For each test program, both *Daikon* and *Udon* were used to generate invariants. By design, *Udon* needs to re-run the program multiple times in order to explore the concurrent behavior of a program, whereas *Daikon* runs the program only once. In order to create a fair comparison, we also allowed *Daikon* to run each test program the same number of times as *Udon*. We refer to the multi-run *Daikon* strategy as *Daikon** and the single-run strategy as *Daikon* in Table I.

Columns 1–4 of Table I show the program name, lines of code (LoC), number of program program points, and number of monitored shared variables, for each test. Columns 5–7 and 8–10 show the total number and the number of incorrect invariants generated by *Daikon*, *Daikon**, and *Udon* respectively. For experiment purposes only, we manually inspected the results to verify if the invariants were true. Finally, Columns 11–13 and 14–16 show the number of runs and run time in seconds for each method.

First, the results show that *Daikon* generates incorrect invariants for every test program. The cause of this is clear: since *Daikon* only exercises a small portion of the concurrent behavior of a program — even if it runs the program multiple times — it fails to observe many different states of the program and, as a result, deduces incorrect invariants. Comparing Columns 8 and 9 of Table I shows that running *Daikon* multiple times on the same program has little to no effect at reducing the number of incorrect invariants. The likely cause of this is that simply re-running a concurrent program repeatedly explores only a small portion of the entire interleaving space.

Second, Columns 7 and 10 show that *Udon* is capable of generating a large number of correct invariants for each test program. On average, *Udon* produces only one incorrect invariant per test. Compared to *Daikon* and *Daikon**, *Udon* produces, on average, over an order of magnitude fewer incorrect invariants. The incorrect invariants generated by *Udon* are due to the fact that HaPSet [7], the default concurrent coverage metric used by *Udon*, can skip certain interleavings where new values of memory could have been explored. As a result, the invariants are generated based on an incomplete exploration of the program.

Table I
COMPARING INVARIANTS GENERATED BY *Daikon* (BASELINE), *Daikon** (MULTI-RUN), AND *Udon* (NEW) FOR THE SAME SET OF PROGRAM POINTS.

				Number of Invariants			Incorrect Invariants			Number of Runs			Run Time (s)		
Name	LoC	Prog. Points	Shared Vars.	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon
Sync01_Safe	64	3	1	15	15	15	1	1	0	1	4	4	1.6	2.8	4.3
FibBenchSafe	55	3	2	24	24	17	10	10	0	1	6	6	2.9	3.4	4.2
Lazy01Safe	52	4	1	20	22	22	7	4	0	1	9	9	2.6	4.3	4.8
Stateful01_Safe	55	3	2	21	21	21	6	6	3	1	4	4	1.4	2.7	3.9
DekkerSafe	68	3	4	39	44	52	29	24	8	1	53	53	1.7	19.5	4.8
LamportSafe	119	3	5	48	59	76	36	44	2	1	58	58	5.0	21.5	5.3
PetersonSafe	55	3	4	39	39	57	29	29	0	1	46	46	1.7	17.3	4.7
TimeVarMutex	55	3	3	27	27	24	9	9	0	1	3	3	1.6	2.2	3.7
Szymanski	106	3	3	30	32	35	25	24	0	1	111	111	1.6	39.4	5.2
IncTrue	55	2	2	15	19	30	3	3	0	1	19	19	2.2	7.7	3.9
IncCas	60	3	1	14	19	38	3	3	0	1	9	9	2.7	3.7	3.6
IncDec	68	5	6	95	122	205	57	53	0	1	39	39	3.4	15.7	6.3
IncDecCas	89	4	3	49	49	73	38	38	1	1	8	8	2.9	4.3	4.8
Reorder	63	3	4	44	54	95	8	6	0	1	29	29	2.5	10.6	7.4
AccountBad	61	4	6	74	78	121	22	15	2	1	9	9	3.8	5.4	6.9
Pfscan	932	24	15	670	798	840	9	9	0	1	20	20	3.0	13.8	8.9
nbds-hashtable	3278	77	14	1123	1194	2064	2	2	0	1	74	74	5.4	119.7	39.0
nbds-skiplist01	2362	54	24	1053	1055	1370	1	1	0	1	161	161	4.4	287.6	26.2
nbds-list_idx01	2386	54	22	773	773	1143	1	1	0	1	132	132	4.6	235.2	17.0
Average				220	234	332	16	15	1	1	42	42	2.8	42.9	8.6

Table II
BREAKDOWN OF THE INVARIANTS GENERATED BY *Udon*.

Name	Regular Invs	Transition Invs	Total Invs	% Trans. Invs
Sync01_Safe	12	3	15	20.0
FibBenchSafe	14	3	17	17.6
Lazy01Safe	17	5	22	22.7
Stateful01_Safe	15	6	21	28.6
DekkerSafe	30	22	52	42.3
LamportSafe	61	15	76	19.7
PetersonSafe	44	13	57	22.8
TimeVarMutex	15	9	24	37.5
Szymanski	25	10	35	28.6
IncTrue	22	8	30	26.7
IncCas	33	5	38	13.2
IncDec	167	38	205	18.5
IncDecCas	58	15	73	20.5
Reorder	83	12	95	12.6
AccountBad	97	24	121	19.8
Pfscan	619	221	840	26.3
nbds-hashtable	1419	645	2064	31.3
nbds-skiplist01	939	431	1370	31.5
nbds-list_idx01	812	331	1143	29.0

Finally, we examine the scalability of our method. Columns 15 and 16 of Table I show that, on average, using our new LLVM based front end for instrumentation results in a five times speedup over the previous, *Daikon*, front end. The reason is that *Daikon*'s front end for C/C++ (called Kvasir) uses Valgrind [18] to dynamically instrument the executable every time it runs the program. Whereas *Udon* instruments the program only once at the compile time. As a result, using our new front end should provide a speed up when analyzing both sequential and multithreaded C/C++ programs.

Table II shows a breakdown of the invariants generated by *Udon*. We classified each invariant into one of two categories: transition invariants over shared variables and all other (regular) invariants. Transition invariants were generated with respect to the entry and exit of each function. Table II shows that by considering only transition invariants we can present the user with a more manageable output compared to considering all invariants. As shown in the previous sections, these transition invariants present a concise summary of the concurrency behavior of a program.

Figure 10 compares the scalability of three interleaving exploration strategies implemented in *Udon*. HaPSet [7] is the default strategy, DPOR [3] is theoretically the ideal strategy (since it will lead to the most precise results), and PCB [6], which is a widely used strategy in the testing literature (we used a context bound of two). In this experiment, we ran *Udon* on the Indexer benchmark from SVCOMP'14 varying the number of threads in the program. Here, the x-axis denotes the number of threads, and the y-axis denotes the number of interleavings explored by each strategy. The result shows that the number of interleavings quickly explodes under DPOR. Under PCB, the increase in the number of interleavings is slower, since only the interleavings with a bounded number of preemptive context switches are explored; nevertheless, the growth is still (predictably) exponential with respect to the number of threads. In contrast, the increase in the number of interleavings is the smallest under HaPSet.

To quantify the effect of different interleaving exploration strategies on the quality of the generated invariants, we ran *Udon* on the benchmarks using HaPSet [7], PCB [6], and DPOR [3], respectively. For PCB, we used a context bound of two. The results are summarized in Table III. Here, we compare the number of runs, time, and number of incorrect invariants. An × in a column indicates the test took longer than two hours. Column 1 shows the name of the test program, Columns 2–4 and 5–7 show the number of runs and time of HaPSet, PCB, and DPOR, respectively. Finally, Columns 8–10 show the number of incorrect invariants found by each method.

First, since DPOR provides a sound guarantee to explore all relevant thread schedules, it produces no incorrect invariants. However, DPOR suffers from an exponential increase in run time relative to the length and number of threads in a program. As a result, DPOR failed to finish analyzing `nbds-hashtable` while both HaPSet and PCB were able to finish in a reasonable time. However, if a user desires high invariant accuracy at the cost of longer run time, *Udon* is capable of using DPOR instead of HaPSet.

Since both HaPSet and PCB skip interleavings where new memory values could be encountered, they both suffer from incorrect invariants being generated. However, on average, HaPSet performs significantly better than PCB in terms of the number of correct invariants created, the number of runs explored, and time. For these reasons, we selected HaPSet as

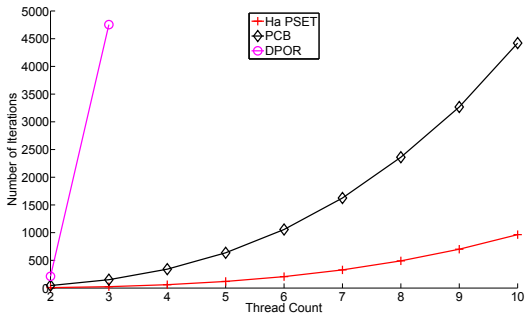

Figure 10. Comparing the thread interleaving exploration strategies in *Udon*.

Table III
COMPARISON OF THE INTERLEAVING EXPLORATION STRATEGIES IN *Udon*.

	Number of Runs			Run Time (s)			Incorrect Invariants		
Name	HaPSet	PCB	DPOR	HaPSet	PCB	DPOR	HaPSet	PCB	DPOR
Sync01_Safe	4	14	7	4.3	4.3	4.7	0	0	0
FibBenchSafe	6	33	17K	4.2	4.3	139.1	0	0	0
Lazy01Safe	9	49	40	4.8	5.3	5.5	0	1	0
Stateful01_Safe	4	13	12	3.9	4.2	4.2	3	1	0
DekkerSafe	53	13	3896	4.8	4.4	37.6	8	16	0
LamportSafe	58	19	392	5.3	4.6	9.4	2	9	0
PetersonSafe	46	13	730	4.7	4.4	12.4	0	0	0
TimeVarMutex	3	17	4	3.7	4.0	4.2	0	4	0
Szymanski	111	21	5980	5.2	4.2	50.3	0	11	0
IncTrue	19	17	212	3.9	3.8	6.1	0	2	0
IncCas	9	19	33	3.6	4.4	5.0	0	0	0
IncDec	39	52	484	6.3	6.7	12.0	0	0	0
IncDecCas	8	22	30	4.8	5.0	5.6	1	8	0
Reorder	29	506	19K	7.4	12.5	234.9	0	2	0
AccountBad	9	53	40	6.9	7.4	7.8	2	19	0
Pfscan	20	100	56K	8.9	11.8	2263	0	0	0
nbds-hashtable	74	879	×	39.0	86.1	×	0	0	×
nbds-skiplist01	161	249	10K	26.2	20.6	217.0	0	0	0
nbds-list_idx01	132	85	1498	17.0	16.0	44.5	0	0	0
Average:	42	115	6187	8.75	11.31	161.2	1	4	0

the default search strategy in *Udon*: it provides a good balance between precision and scalability.

VII. RELATED WORK

Static Techniques. There is a large body of work on using static analysis [19], [20] for invariant generation, whose main advantage is that the reported invariants are true for every reachable state of the program. Typically, invariants generated by these techniques are predicates expressed in some linear abstract domains, such as difference logic, octagonal, or polyhedral. There are also methods based on constraint solving [21], [22], [23], which can generate more complex invariants such as polynomial and non-linear invariants. Recent development along this line includes the work by Furia *et al.* [24] on generating loop invariants from post-conditions, and invariants related to integer arrays [25], [26], [27]. However, due to the inherent limitations of static analysis, these methods tend to lack either in precision or in scalability. Our method, in contrast, relies on dynamic analysis.

Dynamic Techniques. There is also a large body of work on dynamic invariant generation, including tools such as *Daikon* [1], [2], which have been highly successful in practice. The main advantage of dynamic invariant generation is scalability: they have been applied to realistic applications where static techniques fail to scale. Other dynamic invariant generation tools include DIDUCE [28], DySy [29], Agitator [30], and Iodine [31]. However, existing dynamic generation tools do not work well on multithreaded programs due to the nondeterminism in thread scheduling. Our contribution, *Udon*, fills the gap by solving the issue of nondeterminism with respect to dynamic invariant generation.

Hybrid Techniques. There are also hybrid techniques for invariant generation, which leverage both static analysis and dynamic analysis to improve performance. For example, Nguyen *et al.* [8], [9] proposed a method for generating invariants expressed as polynomials and linear relations over a limited number array variables. Such invariants have been difficult to generate by existing methods. There are also hybrid techniques based on random testing [32] and guess-and-check [33], which first generate a set of candidate invariants from concrete execution data and then verify them using SMT solvers.

Interleaving Exploration. There is a large body of work on using selective interleaving exploration techniques for testing concurrent programs, including ConTest [34], CHESS [6], [35], [36], [37], CTrigger [38], CalFuzzer [39], PENELOPE [40], and Maple [41], and property guided pruning techniques [42], [43] implemented in Inspect. Recent empirical evaluations of such techniques can be found in [44], [45], [46]. However, the focus of this paper is not on improving software testing, but on leveraging the related techniques for generating high-quality invariants. In this sense, our work is orthogonal to these existing methods.

Atomicity Inference. Various methods have been proposed for inferring atomicity and detecting concurrency bugs. They may rely on static analysis [47], dynamic analysis [48], [49], [50], [51], [40], [52], or symbolic analysis [53], [54], [55], [56], [57], [55] techniques. However, their focus is primarily on discovering the intended order of conflicting events from different threads. The thread-local transition invariants generated by our new method is similar to the likely deterministic specifications generated by the *Determin* tool [58], which has its own construct for specification of invariants. The main difference is that *Determin* relies on a given set of thread schedules, whereas in our work, different schedules are generated automatically.

VIII. CONCLUSIONS

We presented a new method for dynamically generating invariants from multithreaded programs. We used selective interleaving exploration to simultaneously improve invariant quality while keeping runtime overhead low. We also proposed the use of thread-local transition invariants to help the user understand the code and diagnose concurrency errors. We implemented our method and evaluated it on a set of multithreaded C/C++ programs. Our experiments show that, when compared to the state-of-the-art, such as *Daikon*, our new method produces better invariants while remaining scalable.

ACKNOWLEDGMENTS

This research was primarily supported by the ONR under grant N00014-13-1-0527. Partial support was provided by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *International Conference on Software Engineering*, 1999, pp. 213–224.

[2] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[3] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005, pp. 110–121.

[4] "2014 software verification competition. URL: http://sv-comp.sosy-lab.org/2014/."

[5] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., 1996.

[6] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 267–280.

[7] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *International Conference on Software Engineering*, 2011, pp. 221–230.

[8] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *International Conference on Software Engineering*, 2012, pp. 683–693.

[9] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *International Conference on Software Engineering*, 2014, pp. 608–619.

[10] A. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, 1987, pp. 279–324.

[11] P. Godefroid, "Model checking for programming languages using verisoft," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 174–186.

[12] A. Podelski and A. Rybalchenko, "Transition invariants," *International Symposium on Logic in Computer Science*, pp. 32–41, 2004.

[13] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," University of Utah, Tech. Rep. UUCS-08-004, 2008.

[14] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby, "Efficient stateful dynamic partial order reduction," in *International SPIN Workshop on Model Checking Software*, 2008, pp. 288–305.

[15] G. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[16] E. W. Dijkstra, "The origin of concurrent programming." New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Cooperating Sequential Processes, pp. 65–138.

[17] "Non-blocking data structures. URL: https://code.google.com/p/nbds/."

[18] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003.

[19] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[20] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1978, pp. 84–96.

[21] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *In Computer Aided Verification*, 2003, pp. 420–432.

[22] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using Gröbner bases," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004, pp. 318–329.

[23] E. Rodríguez-Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *J. Symb. Comput.*, vol. 42, no. 4, pp. 443–476, Apr. 2007.

[24] C. A. Furia and B. Meyer, "Fields of logic and computation," 2010, ch. Inferring Loop Invariants Using Postconditions, pp. 277–300.

[25] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar, "Automatic verification of integer array programs," in *International Conference on Computer Aided Verification*, 2009, pp. 157–172.

[26] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006, pp. 427–442.

[27] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov, "Invariant and type inference for matrices," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2010, pp. 163–179.

[28] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *International Conference on Software Engineering*, 2002, pp. 291–301.

[29] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *International Conference on Software Engineering*, 2008, pp. 281–290.

[30] M. Boshernitsan, R. Doong, and A. Savoia, "From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing," in *International Symposium on Software Testing and Analysis*, 2006, pp. 169–180.

[31] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "Iodine: a tool to automatically infer dynamic invariants for hardware designs," in *Design Automation Conference*, 2005, pp. 775–778.

[32] M. Li, "A practical loop invariant generation approach based on random testing, constraint solving and verification," in *International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, 2012, pp. 447–461.

[33] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *European Symposium on Programming*, 2013, pp. 574–592.

[34] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, Jan. 2002.

[35] K. E. Coons, M. Musuvathi, and K. S. McKinley, "Bounded partial-order reduction," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2013, pp. 833–848.

[36] S. Bindal, S. Bansal, and A. Lal, "Variable and thread bounding for systematic testing of multithreaded programs," in *International Symposium on Software Testing and Analysis*, 2013, pp. 145–155.

[37] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-bounded scheduling," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011, pp. 411–422.

[38] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 25–36.

[39] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *International Conference on Computer Aided Verification*, 2009, pp. 675–681.

[40] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: weaving threads to expose atomicity violations," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2010, pp. 37–46.

[41] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2012, pp. 485–502.

[42] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan, "Dynamic model checking with property driven pruning to detect race conditions," in *International Symposium on Automated Technology for Verification and Analysis*, 2008, pp. 126–140.

[43] M. Kusano and C. Wang, "Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction," in *IEEE/ACM International Conference On Automated Software Engineering*, 2014, pp. 175–186.

[44] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: An empirical study," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 15–28.

[45] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "The impact of concurrent coverage metrics on testing effectiveness," in *IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 232–241.

[46] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *International Symposium on Software Testing and Analysis*, 2012, pp. 210–220.

[47] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 1–14.

[48] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 37–48.

[49] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 93–110, 2006.

[50] F. Chen, T. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *International Conference on Software Engineering*, 2008, pp. 221–230.

[51] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang, "Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis," in *International SPIN workshop on Model Checking Software*, 2009, pp. 279–295.

[52] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation," in *International Conference on Formal Methods and Models for Codesign*, 2011, pp. 99–108.

[53] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2010, pp. 328–342.

[54] V. Kahlon and C. Wang, "Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs," in *International Conference on Computer Aided Verification*, 2010, pp. 434–449.

[55] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predicting serializability violations: SMT-based search vs. DPOR-based search," in *Haifa Verification Conference*, 2011, pp. 95–114.

[56] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *NASA Formal Methods*, 2011, pp. 313–327.

[57] N. Sinha and C. Wang, "On interference abstractions," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011, pp. 423–434.

[58] J. Burnim and K. Sen, "DETERMIN: inferring likely deterministic specifications of multithreaded programs," in *International Conference on Software Engineering*, 2010, pp. 415–424.