Coverage Guided Systematic Concurrency Testing

Chao Wang
NEC Laboratories America

Mahmoud Said
Western Michigan University

Aarti Gupta
NEC Laboratories America

ABSTRACT

Shared-memory multi-threaded programs are notoriously difficult to test, and because of the often astronomically large number of thread schedules, testing all possible interleavings is practically infeasible. In this paper we propose a coverage-guided systematic testing framework, where we use dynamically learned ordering constraints over shared object accesses to select only *high-risk* interleavings for test execution. An interleaving is of high-risk if it has not be covered by the ordering constraints, meaning that it has concurrency scenarios that have not been tested. Our method consists of two components. First, we utilize dynamic information collected from good test runs to learn ordering constraints over the memory-accessing and synchronization statements. These ordering constraints are treated as likely invariants since they are respected by all the tested runs. Second, during the process of systematic testing, we use the learned ordering constraints to guide the selection of interleavings for future test execution. Our experiments on public domain multithreaded C/C++ programs show that, by focusing on only the high-risk interleavings rather than enumerating all possible interleavings, our method can increase the coverage of important concurrency scenarios with a reasonable cost and detect most of the concurrency bugs in practice.

1. INTRODUCTION

Real-world concurrent programs are notoriously difficult to test because they often have an astronomically large number of thread interleavings. Furthermore, many concurrency related bugs arise only in rare situations, making it difficult for programmers to anticipate, and for testers to trigger, these error-manifesting thread interleavings. In reality, the common practice of running stress tests is not effective, since the outcome is highly dependent on the underlying operating system which controls the thread scheduling. Merely running the same test again and again does not guarantee that the erroneous interleaving would eventually show up. Typically, in each testing environment, the same interleavings, sometimes with minor variations, tend to be exercised since the scheduler performs context switches at roughly the same program locations.

Systematic concurrency testing techniques [7, 12, 20, 18] offer a more promising solution than stress tests. Although the application settings are different, these techniques use the same *stateless model checking* framework in order to systematically test all possible interleavings with respect to a program input. The model checking is *stateless* in that it directly searches over the space of feasible thread schedules, and in doing so, avoids storing the concrete program states (characterized as combinations of values of the program variables); this is in sharp contrast to classic model checkers (e.g. [9, 8, 3]), which search over the concrete state space—this is a well known cause of memory blowup.

In systematic concurrency testing, the model checker is often implemented by using a specialized *scheduler* process to monitor, as well as control, the execution order of statements of the program under test. A program state s is represented implicitly by the sequence of events that leads the program from the initial state to s. This is based on the assumption that, in a program where interleaving is the only source of nondeterminism, executing the same event sequence always leads to the same state. The state space exploration is conducted implicitly by running the program in its real execution environment again and again, but each time under a different thread schedule. Therefore, systematic concurrency testing can handle programs written in full-fledged programming languages such as C/C++ and Java.

Although systematic concurrency testing has advantages over the common practice of running stress tests (where we are at the mercy of the OS/thread library in triggering the *right* interleaving), it is based on a rather brute-force and exhaustive search. The search covers all possible interleavings (w.r.t. a given test input) in a somewhat pre-determined order, without favoring one interleaving over another and without considering the characteristics of the programs or properties to be tested. Systematic concurrency testing has been shown to be very effective in unit level testing (e.g. [12]). However, because of the often large number of interleavings, such exhaustive search is practically infeasible for realistic applications beyond unit testing.

Although there exist techniques to reduce the cost of exhaustive search in stateless model checking, such as dynamic partial order reduction (DPOR [5]) and preemptive context bounding (PCB [12]), they are not effective for large programs. For example, DPOR groups interleavings into equivalence classes and tests one representative from each equiva-

.

lence class. It is a sound reduction in that it will not miss any bug. However, in practice many equivalence classes themselves are redundant since they correspond to essentially the same concurrency scenarios. Therefore exhaustively testing them not only is expensive, but also rarely pays off.

We propose a coverage-guided *selective search*, where we continuously learn the ordering constraints over shared object accesses in the hope of capturing the already tested concurrency scenarios; then we use the learned information to guide the selection of interleavings to cover the untested scenarios. Since in practice, programmers often make, but sometimes fail to enforce, implicit assumptions regarding concurrency control, e.g. certain blocks are intended to be mutually exclusive, certain blocks are intended to be atomic, and certain operations are intended to be executed in a specific order. Concurrency related program failures are often the result of such implicit assumptions being broken, e.g. data races, atomicity violations, order violations, etc. We try to infer such assumptions dynamically from the already tested interleavings, and use them to identify *high-risk* interleavings for further testing, those interleavings that can break some of the learned assumptions.

Although the programmer's intent may come from many sources, e.g. formal design documents and source code annotation, they are often difficult to get in practice. For example, asking programmers to annotate code or write documents in a certain manner is often perceived as too much of a burden. The more viable approach seems to be to infer them automatically. Fortunately, the very fact that stress tests are less effective in triggering bug-manifesting interleavings also implies that it is viable to dynamically learn the ordering constraints. The reason is that, if no program failure occurs during stress tests, one can assume that the tested interleavings are good—they satisfy the programmer's implicit assumptions [10, 21]. In addition, if the program source code is available, the assumptions may also be mined from the code (e.g. [19]).

In our coverage-guided selective search framework, we use a metric called *History-aware Predecessor-Set* (HaPSet) to capture the ordering constraints over the frequently occurring (and non-erroneous) interleavings. HaPSets can capture common characteristics of a relatively large set of interleavings. During systematic testing, we use HaPSets as guidance to reduce the cost of systematic testing. Realizing that it is practically infeasible to cover all possible interleavings, we choose to execute only those interleavings that are not yet covered by HaPSets. During systematic testing, we also update the HaPSets by continuously learning from the good interleavings generated in this process, untill there are no more interleavings to explore or the desired bug coverage is achieved.

We have implemented the proposed techniques in a systematic testing tool called Fusion, which is designed for testing multithreaded C/C++ programs using Linux/POSIX threads (*PThreads*). Using some public domain concurrent applications as benchmarks, we show that by using HaPSets as guidance in systematic concurrency testing, we can significantly reduce the testing cost, while still maintaining the capability of detecting most of the concurrency bugs in practice. More specifically, in our preliminary experiments, the new *selective search* algorithm found all the bugs, and at the same time was often orders-of-magnitude faster than *exhaustive search*.

2. PRELIMINARIES

2.1 Concurrent Programs

We consider a concurrent program with a finite number of threads as a state transition system. Threads may access local variables in their own stacks, as well as global variables in a shared heap. Program statements that read and/or write global variables are called *(shared) memory-accessing* statements. Program statements that access synchronization primitives are called *synchronization* statements. Program statements that read and/or write only local variables are called *local* statements. For ease of presentation we assume that there is only one statement per source code line. Let $Stmt$ be the set of all statements in the program. Then each $st \in Stmt$ corresponds to a unique pair of source code file name and line number.

A statement st may be executed multiple times, e.g., when it is inside a loop or a subroutine, or when st is executed in more than one thread. Each execution instance of st is called an *event*. Let e be an event and let $stmt(e)$ denote the statement generating e. An event is represented as a tuple $(tid, type, var)$, where tid is the thread index, $type$ is the event type, and var is a shared variable or synchronization object. An event is of one of the following forms.

1. $(tid, read, var)$ is a read from shared variable var;
2. $(tid, write, var)$ is a write to shared variable var;
3. $(tid, fork, var)$ creates the child thread var;
4. $(tid, join, var)$ joins back the child thread var;
5. $(tid, lock, var)$ acquires the lock variable var;
6. $(tid, unlock, var)$ releases the lock variable var;
7. $(tid, wait, var)$ waits on condition variable var;
8. $(tid, notify, var)$ wakes up an event waiting on var;
9. $(tid, notifyall, var)$ wakes up all events waiting on var.

In addition, we use the generic event $(tid, access, var)$ to capture all other shared resource accesses that cannot be classified as any of the above types, e.g. accesses to a socket. We do not monitor thread-local statements.

2.2 The State Space

We use $\mathcal{S}$ to denote the set of program states. A transition $\mathcal{S} \xrightarrow{e} \mathcal{S}$ advances the program from one state to a successor state by executing an event e. An event is *enabled* in state s if it is allowed to execute according to the program semantics. We use $s \xrightarrow{e} s'$ to denote that event e is enabled in s, and state s' is the next state. Two events e_1, e_2 *may be co-enabled* if there exists a state s in which both of them are enabled. An execution ρ (interleaving) is a sequence $s_0, \ldots, s_n$ of states such that for all $1 \leq i \leq n$, there exists a transition $s_{i-1} \xrightarrow{e_i} s_i$.

During systematic concurrency testing, ρ is stored in a search stack S. We call $s \in S$ an abstract state, because unlike a concrete program state, s does not store the actual valuation of all program variables. (Storing concrete program states is less practical for large concurrent applications.) Instead, each s is implicitly represented by the sequence of executed events leading the program from the initial state s_0 to s. This is based on the assumption that executing the same event sequence leads to the same state.

Two concurrent transitions are *independent* if and only if the two events can neither disable nor enable each other, and swapping their order of execution does not change the combined effect. For example, two events are dependent if they access the same the object and at least one is a write (modification); and a lock acquire is dependent with another lock acquire over the same lock variable. In the literature (e.g. [6]), two interleavings are considered as equivalent iff they can be transformed into each other by repeatedly swapping the adjacent and independent transitions.

2.3 Predecessor Sets

An execution $\rho = s_0 \ldots s_n$ defines a total order over the set of memory-accessing and synchronization events. The predecessor set (PSet [21]) was designed to efficiently capture the event ordering constraints common to a potentially large set of executions. To this end, PSet was defined over memory-accessing statements, i.e. between read/write operations that may be executed adjacent to each other. Synchronization statements are ignored (this does not suit our purpose; we will extend in our definition of HaPSet).

Definition 1 (PSet). *Given a set $\{\rho_1, \ldots, \rho_n\}$ of interleavings and a memory-accessing statement $st \in Stmt$. The predecessor set, denoted $\mathsf{PSet}[st]$, is a set $\{st_1, \ldots, st_k\}$ of statements such that, for all $i : 1 \leq i \leq k$, an event produced by statement st is immediately dependent upon an event produced by statement st_i in some interleaving ρ_j, where $1 \leq j \leq n$.*

Event e is *immediately dependent* upon e_i if and only if e_i is the last event in its thread accessing the same object as t, and it is also dependent with e. Both st and st_i are statements rather than events, which are distinct execution instances of these statements. This is meant to keep PSets general enough so that the PSets learned from one correct interleaving remain valid for another correct interleaving, possibly under a different program input.

PSets are powerful enough to characterize the common bug patterns such as data races, atomicity violations, and order violation. Since each $\mathsf{PSet}[st]$ has at most $|Stmt|$ elements, the space to store all PSets is $O(|Stmt|^2)$ in the worst case. On average, however, the PSets learned from real-world programs are usually very small. The study in [21] showed that 95% of the program statements have an empty PSet, and for well-designed test suites, typically it takes less than 100 interleavings for PSet learning to converge, i.e. few new updates are possible afterward.

3. ORDERING CONSTRAINTS

Although PSets are efficient in capturing ordering constraints common to a large set of thread interleavings, as a coverage metric it does not suit our purpose well. This is because the applications are different. In [21], PSets are collected from good runs during testing and then treated as program invariants during production runs. A special-purpose microprocessor is designed to ensure that the PSets are always obeyed (with checkpoints and rollbacks upon PSet violations). The rationale is that, if PSets capture the concurrency scenarios of the tested interleavings, then by allowing only PSet-obeying interleavings in production runs, one can steer away from program failures even if the programs are still buggy.

In this paper, our goal is not runtime failure avoidance as in [21] but to improve the coverage during testing. The main difference is that, for failure avoidance, it is acceptable if some already tested interleavings are not captured by the PSets (as long as they are rare, disallowing them in production runs will not hurt performance much). However, for testing, it is crucial to capture what has already been tested, since the purpose is to prevent the same concurrency scenario from being tested again.

3.1 History-aware Predecessor Sets

We extend the idea of PSet to define a new coverage metric called HaPSet[1]. There are two main differences between HaPSets and PSets. First, we consider both synchronization statements (e.g. lock acquires) as well as memory-accessing statements in the definition of HaPSet. Second, for each $st \in Stmt$, in addition to the fields $file$ and $line$, we include thr and ctx, where thr is the thread that executes st and ctx is the call stack at the time st is executed. The reason is as follows: With $(file, line)$, there remains some degree of ambiguity regarding the statement which produces an event at run time. For example, the same statement may be executed in multiple function/method call contexts, or from multiple threads. In many cases, especially in object-oriented programs, such information is useful and should be included in order to capture any meaningful ordering constraint.

Since at run time, both the number of threads and the number of distinct calling contexts can be large, to avoid memory blowup, ctx only stores the most recent k (some small number—5 in our experiments) entries in the call stack, and thr only takes two values: 0 means it is the *local* thread, and 1 means it is the *remote* thread. Let e and e' be two events in an interleaving such that $stmt(e) = st$ and $stmt(e') = st'$, we have $st.thr = 0$ and $st'.thr = 1$ when $tid(e) < tid(e')$, and $st.thr = 1$ and $st'.thr = 0$ when $tid(e) > tid(e')$. We do not consider $tid(e) = tid(e')$, since it never triggers the HaPSet update. Formally, statement st is now defined as a tuple $(file, line, thr, ctx)$, where $file$ is the file name, $line$ is the line number, $thr \in \{0, 1\}$ is the thread, and ctx is the truncated calling context.

Definition 2 (HaPSet). *Given a set $\{\rho_1, \ldots, \rho_n\}$ of interleavings and a memory-accessing or synchronization statement $st \in Stmt$. The History-aware Predecessor Set, or $\mathsf{HaPSet}[st]$, is a 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 ρ_j, where $1 \leq j \leq n$.*

Note that this metric includes both syntactic and semantic elements. Data conflicts are at the heart of most concurrency errors (data races, atomicity violations, etc.)–these are tracked to make this metric relevant for the purpose of finding bugs. However, a generalization is achieved by associating it syntactically with statements, rather than with states. The thread index is again designed to distinguish between two threads for catching bugs, but abstracts over specific thread ids, thereby ensuring that it is scalable over many threads. Finally, by including a bounded functional context, we provide some measure of context-sensitivity–this is especially useful for object-oriented programs.

[1]pronounced as "Happy Set."

```
Thread T1                 Thread T2

                          { //alloc
                          e1 : p := &a;
                          }
{ //use
e2 : if (p ≠ 0)
e3 :    *(p) := 10
}
                          { //free
                          e4 : p := 0;
                          }
```

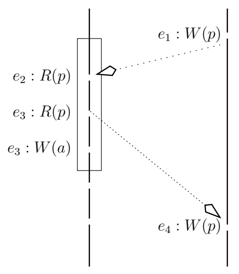

Figure 1: A serial execution of the intended atomic block (e_2e_3).

Example. Consider Figure 1, which has two threads T_1, T_2 sharing the pointer p. Assume that $p = 0$ initially. In the given execution, p is first initialized in e_1, then used in e_2, e_3, and finally freed in e_4. (We assume $e_1 - e_4$ are statements in the form $(file, line, thr, ctx)$.) Since e_1 is the last statement before e_2 and they have a data conflict, we add e_1 to $\mathsf{HaPSet}[e_2]$. For e_3 we do not add any statement into $\mathsf{HaPSet}[e_3]$ because e_2 is the last statement accessing p but it is from the same thread (hence no conflict). We add e_3 to $\mathsf{HaPSet}[e_4]$ since e_3 precedes e_4 in the given execution, and they have a data conflict. To sum up, the HaPSets learned from this execution are as follows,

$$\mathsf{HaPSet}[e_1] = \{\ \}, \quad \mathsf{HaPSet}[e_2] = \{e_1\},$$
$$\mathsf{HaPSet}[e_3] = \{\ \}, \quad \mathsf{HaPSet}[e_4] = \{e_3\}.$$

3.2 Why HaPSets are Useful?

We show that the seemingly simple HaPSets are capable of capture subtle concurrency control patterns.

3.2.1 Atomicity Violation

Consider Figure 1 again. In this example, the block containing e_2, e_3 is meant to be executed atomically—it first checks whether the pointer p is null, and if it is not null, assign 10 to the memory location pointed by p. Therefore, whether e_2 and e_3 are two consecutive reads of an interleaving is the key in deciding whether the interleaving is buggy or not. Let us see how HaPSets capture this atomicity constraint. First, note that in all good runs (where the atomicity is not violated), $\mathsf{HaPSet}[e_3]$ is always empty. This is because, although e_1, e_4 can be executed either before e_2 or after e_3, event e_3 is always preceded by e_2. Therefore, neither e_1 nor e_4 can appear in $\mathsf{HaPSet}[e_3]$. Second, $e_2 \notin \mathsf{HaPSet}[e_4]$ because e_3 (instead of e_2) always precedes e_4. Therefore the HaPSets leaned from all the good runs are as follows,

$$\mathsf{HaPSet}[e_1] = \{e_2\}, \quad \mathsf{HaPSet}[e_2] = \{e_1, e_4\},$$
$$\mathsf{HaPSet}[e_3] = \{\ \}, \quad \mathsf{HaPSet}[e_4] = \{e_3\}.$$

When using HaPSets as guidance during the systematic testing, it would be more fruitful to test interleavings that have not been *covered* by the above HaPSets. One such interleaving is $\rho' = e_1e_2e_4e_3$, which violates the atomicity and leads to the dereference of a null pointer. Note that ρ' corresponds to $\mathsf{HaPSet}[e_3] = \{e_4\}$ and $\mathsf{HaPSet}[e_4] = \{e_2\}$.

3.2.2 Busy Waiting

```
Thread T1                Thread T2             execution

a  do {                  f  lock(A)            c^1: R(x)
b     lock(A)            g  x = 1;                ...
c     tmp = x;           h  unlock(A)          c^2: R(x)
d     unlock(A)                                       ...
e  } while(tmp);                                   g: W(x)
                                               c^n: R(x)
```

Figure 2: The busy waiting example. Without HaPSet, systematic testing would result in excessive backtracking.

HaPSets can be used to avoid excessive testing of redundant interleavings—those that do not offer any new concurrency scenario. Consider Figure 2 as an example. There are two threads T_1, T_2 communicating via variable x. Assume that $x = 0$ initially. In the given execution $\{abcde\}^k fghabcde$, the loop in T_1 is executed k times before g in thread T_2 is executed.

Without the HaPSet learning and guidance, systematic testing would have to test a potentially large set of interleavings, each with a different number of loop iterations. This is because, strictly speaking, none of these interleavings are equivalent to others; therefore, based on the theory of partial order reduction, one needs to test all of them. However, such tests are often wasteful since they rarely lead to additional bugs. The HaPSets computed on the given execution are

$$\mathsf{HaPSet}[g] = \{c\}, \quad \mathsf{HaPSet}[c] = \{g\},$$
$$\mathsf{HaPSet}[b] = \{f\}, \quad \mathsf{HaPSet}[f] = \{b\}.$$

since some instances of statement c (or f) are immediately dependent on instances of g (or b), and vice versa. (Except for recursive locks, we ignore *unlock* statements in the computation of HaPSets.) When using HaPSets as guidance, we can avoid the aforementioned excessive backtracking because none of these interleavings can offer a concurrency scenario that has not been covered by the HaPSets.

3.3 Learning from Good Runs

For our guided search to be effective, we need to learn HaPSets from a diversified set of interleavings. The quality of the learned HaPSets will be affected by both the test cases and the thread schedules. To diversify the thread schedules, we add randomized delays. In this testing environment, the program is executed under the control of a *scheduler* process, which is capable of controlling the order of operations from different threads. These control points are inserted into the program source code automatically via an instrumentation phase, before the source code is compiled into an executable.

For HaPSet learning, we maintain the following data structures: a set $\mathsf{HaPSet}[st]$ for each statement $st \in Stmt$; and a search stack S of abstract states $s_0 \dots s_n$, where s_0 is the initial state and s_n is the final state of the interleaving. Recall that each $s \in S$ is an abstract state because s does not store the actual valuations of program variables. Let $s_i.sel$ be the event executed at s_i in the given interleaving in order to reach s_{i+1}.

The pseudo code of our HaPSet learning is presented in Algorithm 1. The procedure RANDCTEST takes the initial state s_0 as input and generates the first interleaving with a randomized thread schedule. Each state $s \in S$ is associated with a set $s.enabled$ of events. Recall, for example, that a lock *acquire* would be considered as disabled at s, if the lock is held by another thread. Similarly, a *wait* would be considered as disabled at s, if the notification has not been sent. At each execution step, we randomly pick an event $e \in s.enabled$, execute it from s, which leads to state s'.

Note that the thread schedules ultimately are still determined by the underlying operating system. This ensures that all the generated interleavings are real. If any of them can trigger a program failure, then it is a real bug. Otherwise, all of them are assumed to be good runs, in that they expose the desired program behavior.

Algorithm 1 Learning from good test runs

```
1: Initially: For all statements $st$, HaPSet[$st$] is empty;
2:            $S$ is an empty stack; randCTest($s_0$)

3: randCTest($s$) {
4:   $S$.push($s$);
5:   LearnHaPSets($s$);                  // learning HaPSets
6:   while ($s.enabled$ is not empty) {
7:     Let $e$ be a randomly chosen item from $s.enabled$;
8:     Delay thread $tid(e)$ for a randomly chosen period;
9:     Let $s.sel = e$;
10:    Let $s'$ be the new state after executing $s \xrightarrow{e} s'$;
11:    randCTest($s'$);
12:  }
13:  $S$.pop($s$);
14: }

15: LearnHaPSets($s$) {
16:   if ($s \neq s_0$) ){
17:     Let $s_p \in S$ be the state preceding $s$;
18:     Traverse stack $S$, for each thread, find the last state
          $s_d$. where $s_d.sel$ and $s_p.sel$ access the same object;
19:     if ($s_d.sel$ and $s_p.sel$ have a data conflict) {
20:       Let $st_p = stmt(s_p.sel)$;
21:       Let $st_d = stmt(s_d.sel)$;
22:       HaPSet[$st_p$] ← HaPSet[$st_p$] ∪ {$st_d$}
23:     }
24:   }
25: }
```

During each run, we invoke learnHaPSets at every execution step. The input to this procedure is the newly reached state s. Let s_p be the state prior to reaching the current state s, and $s_p.sel$ be the event executed between s_p and s. For each thread, we find the last executed event $s_d.sel$ such that (1) $s_d.sel$ and $s_p.sel$ access the same object, (2) they are executed by different threads, and (3) there is a data conflict (read-write, write-write, lock-lock, or wait-notify). If such an $s_d.sel$ exists, we add the statement $stmt(s_d.sel)$ into the HaPSet of $stmt(s_p.sel)$.

4. SYSTEMATIC TESTING

As we mentioned earlier, systematically testing all possible interleavings can be achieved using stateless model checking. It can be viewed as a natural extension of randCTest in Algorithm 1. However, unlike randomized testing, here the scheduler has total control in deciding and enforcing the actual thread schedule.

4.1 Overall Algorithm

The overall algorithm is illustrated in Algorithm 2 by procedure sysCTest. It checks all possible thread schedules of the program for a given test input.

Algorithm 2 Systematic concurrency testing framework

```
1: Initially: $S$ is an empty stack; sysCTest($s_0$)

2: sysCTest($s$) {
3:   $S$.push($s$);
4:   UpdateBacktrack($s$);
5:   let $\tau \in Tid$ such that $\exists t \in s.enabled : tid(t) = \tau$;
6:   $s.backtrack \leftarrow \{\tau\}$;
7:   $s.done \leftarrow \varnothing$;
8:   while ($\exists t$: $tid(t) \in s.backtrack$ and $t \notin s.done$) {
9:     $s.done \leftarrow s.done \cup \{t\}$;
10:    let $s.sel = t$;
11:    let $s'$ be the new state after executing $s \xrightarrow{t} s'$;
12:    sysCTest($s'$);
13:  }
14:  $S$.pop( );
15: }

16: UpdateBacktrack($s$) {
17:   for each $t \in s.enabled$ {
18:     let $s_d \in S$ and $s_d.sel$ be the latest event such that
          $s_d.sel$ is dependent and may be co-enabled with $t$,
19:     if (such $s_d$ exists){
20:       $s_d.backtrack \leftarrow s_d.backtrack \cup$ BtSet($s_d, t$)
21:     }
22:   }
23: }
```

In addition to $s.enabled$, each state s is also associated with a set $s.done \subseteq s.enabled$ of already executed events; it records the scheduling choices made at s in some previous test runs. Furthermore, each state s is associated with a set $s.backtrack$, consisting of a subset of the enabled threads at s. Each $\tau \in s.backtrack$ represents a future scheduling choice at s, i.e. thread τ will be executed at s in some future test run.

The procedure sysCTest takes state s as input, where s_0 is used for the initial call. At each execution step, it first invokes subroutine updateBacktrack to update backtracking points at some previous state $s' \in S$. (Backtracking will be explained in the next paragraph.) Then from $s.backtrack$ it picks an enabled thread τ to execute, leading to a distinct thread interleaving down the line. The recursive call at Line 11 returns only after the interleaving ends and we have backtracked to state s. At this point, $s.backtrack$ must have been updated by some previous call to sysCTest; it may contain some threads other than τ, meaning that executing them (as opposed to τ) from state s may lead to different interleavings. The entire procedure terminates when we backtrack from state s_0 eventually. Since we do not store the concrete program states in S, backtracking to a state s' is implemented by re-starting the test run and then applying the same thread schedule till state s' is reached again.

In the naive approach, at every state $s \in S$, $s.backtrack$ consists of all the enabled threads. The set of interleavings generated by this naive algorithm is the same as the set of possible interleavings generated by the actual program execution. However, the naive approach may end up testing many redundant interleavings. UPDATEBACKTRACK(s) is designed to remove some of the redundant interleavings. It takes the current state as input and iterates through all the enabled event $t \in s.enabled$ to find the latest event $s_d.sel$ that is dependent and may be co-enabled with t. If such an s_d exists, it means that if we flip the execution order from $s_d.sel \ldots t$ to $t \ldots s_d.sel$, the new interleaving will not be equivalent to the current one. In practice, the various systematic concurrency testing tools differ mainly in their ways of computing the backtrack set.

4.2 Backtracking: Baseline and Variations

The baseline algorithm is only slightly different from the naive algorithm. That is,

$$BTSet \leftarrow \{tid(q) \mid q \in s_d.enabled\}$$

It is still more efficient than the naive algorithm, since it adds BTSET only at state s_d (as opposed to every state). For example, consider the case where s_d does not exist in Line 18. In this case, t is independent with all the previously executed events ($s_d.sel$ for all $s_d \in S$), and swapping the execution order of t and $s_d.sel$ would not lead to a new equivalence class. The baseline algorithm would not add any backtrack point for such cases.

4.2.1 Preemptive Context-Bounding (PCB)

Traditionally, a *context switch* is defined as the computing process of storing and restoring the CPU state (context) when executing a concurrent program, such that multiple processes or threads can share the same CPU resource. The idea of using context bounding to reduce complexity of software verification was first introduced by Qadeer and Wu [14] for static program analysis and later extended to testing [12]. It has since become an influential techniques since in practice many concurrency bugs can be exposed by interleavings with few context switches. In this setting,

$$BTSet \leftarrow \{tid(q) \mid q \in s_d.enabled, \text{ and } cb(s_d, q) \le mcb\}$$

where $cb(s_d, q)$ is the number of context switches after executing q at s_d, and mcb is the maximal number of preemptive context switches allowed in an interleaving. From state s_d, one can execute event q only if the number of context switches will not exceed the bound.

Although PCB allows us to skip many interleavings, for those with $\le mcb$ context switches, it still needs to test them using brute-force exhaustive search. For large programs, even with small context bound (e.g. 4 or 5), the number of thread interleavings can still be extremely large.

4.2.2 Dynamic Partial Order Reduction (DPOR)

Partial order reduction is based grouping interleavings into equivalence classes and then testing only one representative from each equivalence class. It is a well studied topic in model checking. For concurrency testing, the most advanced technique is the DPOR algorithm by Flanagan and Godefroid [5]. BTSET is computed by Algorithm 3. First, we search for an event $q \in s_d.enabled$ such that there exists a happens-before relation between q and the currently enabled event t. Intuitively, q happens before t in an interleaving if either (a) we cannot execute t before q due to program semantics, or (b) swapping the execution order of q and t would lead to a different equivalence class. Obviously q *happens before* t if they are from the same thread. Other examples include (1) q and t are from different threads but have data conflict over a shared object; and (2) there exist events r, s in the interleaving such that, q happens before r, r happens before s, and s happens before t. The happens-before relation is transitive (cf. [5]).

Algorithm 3 Computing the backtrack set in DPOR.

```
1: let $q \in s_d.enabled$ such that either $tid(q) = tid(t)$, or
   there is a happens-before relation between $q$ and $t$ }
2: if (such $q$ exists)
3:     BTSET $\leftarrow \{tid(q)\}$;
4: else
5:     BTSET $\leftarrow \{tid(q) \mid q \in s_d.enabled\}$;
```

If such q exists, then we have a reduction—we only need to add $tid(q)$ to $s_d.backtrack$, since executing thread $tid(q)$ is necessary for the purpose of swapping t and $s_d.sel$. (In POR theory, this backtrack set is called a *persistent set*.) Otherwise, we do not have reduction and have to resort to the baseline to add all enabled threads to $s_d.backtrack$. Although partial order reduction is sound in that it never misses real bugs, in practice, the number of interleavings after DPOR can still be very large.

5. GUIDING SYSTEMATIC TESTING

Our coverage-guided search algorithm builds on the systematic testing framework in Algorithm 2. In contrast to the exhaustive search strategies used by DPOR and PCB, we use HaPSets learned from the already tested (good) runs to select high-risk interleavings for future testing. In this section, we first explain how to use HaPSets to guide the interleaving selection, and then explain how to continuously update the HaPSets.

5.1 Guiding Interleaving Selection

We achieve this by modifying the implementation of subroutine UPDATEBACKTRACK. Recall that in Algorithm 2, Line 18 of UPDATEBACKTRACK searches through the stack S to find the last event $s_d.sel$ that is dependedent and may be co-enabled with t. If such an $s_d.sel$ exists, it means that swapping the execution order from $s_d.sel \ldots t$ to $t \ldots s_d.sel$ would produce a different interleaving. In the modified version, we insist that in addition to the condition in Line 18, the following HaPSet related condition must hold: $stmt(t) \notin \mathsf{HaPSet}[stmt(s_d.sel)]$.

Note that if $stmt(t)$ is not in the HaPSet of $stmt(s_d.sel)$, it means that in all tested runs, the statement that generates $s_d.sel$ has never been immediately dependent upon the statement that generates t. In this case, the new execution order $t \ldots s_d.sel$ represents a concurrency scenario that has never been covered by the previous test runs. On the other hand, if $stmt(t)$ is already in the HaPSet of $stmt(s_d.sel)$, the new interleaving would have a lower risk because this concurrency scenario has been covered previously.

Algorithm 4 illustrates our new procedure UPDATEBACKTRACK for HaPSet guided selective search. One of the main advantages of our HaPSet guided search is that, it fits naturally into the existing flow of systematic testing. The addition of HaPSet guided search requires only small changes to the software architecture. The guidance from HaPSets affect only our selection of state s_d (Line 4). Once s_d is selected, the backtrack set can be computed independently. This

Algorithm 4 Guiding the systematic testing (with DPOR)

```
1: UPDATEBACKTRACK($s$) {
2:   for each $t \in s.enabled$ {
3:     let $s_d \in S$ and $s_d.sel$ be the latest event such that
         (1) $s_d.sel$ is dependent and may be co-enabled with $t$,
         (2) $stmt(t) \notin$ HaPSet$[stmt(s_d.sel)]$;      // guiding
4:     if (such $s_d$ exists){
5:       $s_d.backtrack \leftarrow s_d.backtrack \cup$ BTSET($s_d, t$)
6:     }
7:   }
8: }
```

means we can choose to use the various existing methods to compute BTSET. In practice, we have found that both PCB and DPOR work well under the guidance of HaPSets, although combining HaPSet with DPOR often performs slightly better. Note that HaPSet guidance effectively prunes away large subspaces in the search. Unlike DPOR, this pruning is not safe, i.e. it may miss errors. This is the basic tradeoff we make to gain scalability and improved performance.

At this point, one may contemplate the possibility of using HaPSets with both DPOR and PCB. We caution that there is a theoretical difficulty in soundly combining PCB with DPOR (or any persistent-set based POR) in the first place. PCB and DPOR fundamentally are not compatible, because if you use both, and also set the context bound is k, some equivalence classes may be missed completely even if they actually contain some interleavings with $CB \leq k$. In [11], Musuvathi and Qadeer designed a method to combine a sleep-set based POR with context bounding (and method is quite involved), but to our knowledge, there has been no method for soundly combining PCB with persistent-set based POR (such as DPOR).

5.2 Continuous Learning

In our guided search framework, the quality of HaPSets is very important. Although we try to diversify the thread schedules via randomization, the training runs may still miss crucial concurrency scenarios. The interleaving encountered during the guided search may contain these missing concurrency scenarios, and therefore are complementary to the initial learning. Therefore, we propose to update the initial HaPSets during systematic testing by *continuously learning* from the tested (good) interleavings. Continuous learning is made possible by the fact that, unless a bug is detected, the interleaving checked by systematic testing is always a good run.

Algorithm 5 illustrates the overall selective search algorithm, wherein the call to LEARNHAPSETS at Line 4 allows for continuous learning of HaPSets. The learning subroutine is the same as the one used during the initial learning in Algorithm 1.

The nice thing about continuous learning is that, the good interleavings produced by systematic testing are freely available, since they are byproducts of the search. The more concurrency scenarios we capture using the HaPSets, the less number of interleavings would need to be tested in the future. This ensures progress with respect to the HaPSet coverage metric. Therefore, on-the-fly updating HaPSets allows the guided search to become a self-improving process, making the whole process converge much faster.

Algorithm 5 Continuous learning within systematic testing

```
1: Initially: $S$ is an empty stack; GUIDEDCTEST($s_0$)

2: GUIDEDCTEST($s$) {
3:   $S$.push($s$);
4:   LEARNHAPSETS($s$);          // continuous learning
5:   UPDATEBACKTRACK($s$);
6:   let $\tau \in Tid$ such that $\exists t \in s.enabled : tid(t) = \tau$;
7:   $s.backtrack \leftarrow \{\tau\}$;
8:   $s.done \leftarrow \varnothing$;
9:   while ($\exists t$: $tid(t) \in s.backtrack$ and $t \notin s.done$) {
10:    $s.done \leftarrow s.done \cup \{t\}$;
11:    let $s'$ be the new state after executing $s \xrightarrow{t} s'$;
12:    GUIDEDCTEST($s'$);
13:  }
14:  $S$.pop( );
15: }
```

Example. Consider Figure 2 again. Assume that the first interleaving is $\rho_1 = s_0 \xrightarrow{a} s_1 \xrightarrow{f} s_2 \xrightarrow{g} \ldots s_5 \xrightarrow{b} s_6 \xrightarrow{c} \ldots$. The HaPSets computed from ρ_1 via continuous learning are HaPSet$[c] = \{g\}$, HaPSet$[b] = \{f\}$. Furthermore, the DPOR backtrack sets will be $s_1.backtrack = \{1, 2\}$ and $s_2.backtrack = \{2\}$, since thread 1 is disabled at state s_2. According to our guided search algorithm, the next interleaving to be executed is $\rho_2 = s_0 \xrightarrow{a} s_1 \xrightarrow{b} \ldots$. The new HaPSets computed from ρ_2 are HaPSet$[g] = \{c\}$, HaPSet$[f] = \{b\}$. After that, however, our guided search algorithm will allow no other interleavings. The key point here is that, in many cases the pruning actually happens at states like s_1, where the locking statements are executed, not at the states where the memory-accessing statements (c, g) are executed. This is why we need to include synchronizations in the definition of HaPSet. In fact, if we use only memory-accessing statements (as in the definition of PSet [21]), there will be no pruning possible for Figure 2.

6. EXPERIMENTS

We have implemented the proposed method in a tool called FUSION. The tool is capable of testing multithreaded C/C++ programs written using the POSIX thread library. We use source code instrumentation to add the monitoring and control points to the program, in order to control the memory-accessing and synchronization statements at run time. Our implementation is based on the C/C++ front-end from Edison Design Group. The instrumentation consists of two steps: (1) before each shared memory access, it inserts a request to the scheduler asking for permission to execute; (2) before each *PThreads* library routine, it inserts a request to the scheduler. Since identifying *a priori* the set of memory locations that may be shared by more than one thread is difficult for realistic C/C++ programs (due to the widespread use of pointers and heap allocated data structures), we use a light-weight intra-procedural escape analysis to conservatively decide whether a statement may access the shared memory. This is a sound approximation because treating a local statement as if it is shared poses no threat to the correctness of our testing tool—it merely increases the monitoring/control overhead at runtime.

In addition, any system/library function calls that may block the calling thread also need to be monitored. This includes, for example, system calls for socket communication (e.g. `select, send, recv`). This also includes system calls using realtime information (e.g. `usleep(), sched_yield(), pthread_cond_timedwait()`). Such system calls need to be properly modeled by the scheduler of the testing tool.

We conducted experiments on some real-world C/C++ applications written for the Linux/PThreads platform. All benchmarks are from the public domain, accompanied by test cases to facilitate concrete execution. Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory. We have compared the runtime performance as well as the bug-detecting capability of the following methods: HaPSet, DPOR, and PCB. Here HaPSet is our guided search algorithm. DPOR is the original DPOR algorithm [5]. For PCB [12], we have set the context switch bounds from 0 to 4. For a fair comparison of the three testing methods, we skipped *a priori* HaPSet learning sessions, while relying solely on continuous learning to infer the HaPSets.

6.1 The `Thrift` C++ Library

Our first set of benchmarks come from the Thrift C++ library. Thrift is a software framework used by Facebook for scalable cross-language services development. The library has 18.5K lines of C++ code. We used the version checked out directly from the main development trunk (as of the paper submission time). The test program is also from the main trunk, as part of the `make check` script. There is a deadlock error inside the concurrency package, which itself is a thin C++ layer wrapping up PThreads mutex and condition variable routines to support thread pool and task management.

The original test program was written for stress tests. It upfront creates hundreds of worker threads and tens of thousands of tasks to run in parallel. This is a typical way of creating a heavy workload, hoping to increase the odds of triggering some rare and bug-manifesting interleavings. With systematic testing, we do not need that many threads/tasks to expose bugs. Therefore, we set the number of threads from 2 to 5, and with on average 5 tasks per threads.

We compared the performance of the three methods. The results are shown in Table 1. The first four columns show the name, the lines of code, the number of threads, and the bug type. Here `thrift-lib-w2-5t`, for example, stands for the test case with 2 worker threads and 5 tasks per worker. The remaining columns show the performance of each method, including the number of interleavings tested and the run time in seconds. We set the time bound to 10 minutes per method, i.e. TO in the table means timed out in 600 seconds without finding a bug.

The results show that HaPSet found all the bugs and was also fast. Furthermore, it scaled well as we increased the number of concurrent threads. In comparison, DPOR found a bug for the 2-worker case, while timed out for the other cases. For PCB0 (with $mcb = 0$), it terminated in 247.2 seconds and missed the bug. (Our experience shows that in general PCB0 is not effective since it frequently misses real bugs.) With context bound set to 1, PCB found the bugs for the 2-worker and 3-work cases. However, PCB did not scale as well when we increased the number of threads or the context bound.

All three algorithms have significant runtime overhead in comparison to a native test execution. Depending on the types of target programs, i.e. CPU-bound or communication-bound, the slowdown ranges from 10X to 100X. This overhead comes from two sources. First, in order to control the nondeterminism in executing concurrent programs, the scheduler insists that at any time, only one thread is allowed to execute. This essentially serializes a concurrent execution. Second, the monitoring and control of memory-accessing events often have large overhead.

For `thrift-lib-w2-5t`, although HaPSet checked 14 runs, it actually spent more time than what DPOR spent on checking 23 runs. This is because not all these 14 runs are included in the 23 runs; and each run may execute a different set of statements and therefore may take a different amount of time. Furthermore, both HaPSet learning and guiding have some computational overhead.

6.2 The `aget/pbzip/pfscan` Benchmarks

Our second set of benchmarks are medium-size multi-threaded applications downloaded from the *sourceforge.com* website. They include `aget-0.4`, a ftp client capability of concurrently downloading different segments of a large file, `pbzip2-0.9.4`, a parallel implementation of bzip2 for file compression and decompression, and `pfscan-1.0`, a concurrent file scanner that combines the functionality of `find, xargs` and `fgrep`. First we compared the performance of the three methods. The results are shown in Table 2. For these exampels, HaPSet found all the bugs and was also the fastest, whereas both DPOR and PCB2 timed out on `pbz2-f` and `pfscan` .

Table 2: Comparison of HaPSet, DPOR, and PCB2 on the `aget/pbzip2/pfscan` examples

Test Program				HaPSet		DPOR		PCB2	
name	LoC	bug	thr	runs	time	runs	time	runs	time
aget	1.2k	race	6	14	36.9	96	173	94	172
pbzip2	1.9k	order	7	2	0.5	2	0.5	2	0.5
pbz2-f	1.9k	race	7	8	2.6	608	TO	631	TO
pfscan	960	deadlk	3	28	2.8	1541	TO	2867	TO

In `aget`, there is a data race over a variable called `bwritten` which is shared by the multiple downloading worker threads and a separate thread updating the progress bar. In `pbzip2`, there is an order violation between the main thread and the consumer threads, sometimes causing a segmentation fault as a result of null pointer dereferencing. After fixing this bug (`pbz2-f`), our tool found a previously unreported data race over variables `OutputBuff[i].buf` and `OutputBuff[i].bufSize` between the consumer threads and the fileWriter thread. This is a real bug that may cause corrupted file output. In `pfscan`, there is an injected order violation [21], where a variable called `aworkers`, if initialized too late in time, may cause the main thread to hang.

On `aget`, we also compared the various settings of PCB (with mcb from 0 to 4), to assess its scalability. The results in Table 3 show that, PCB0 timed out after 10 minutes without finding the bug. With all the other settings, PCB found the bug. Although PCB1 has the best performance for this example, we caution that in general one needs at least PCB2 since even the simplest atomicity violations need at least two

Table 1: Comparison of HaPSet, DPOR, and PCB with various bounds on the `thrift-lib-cpp` example.

Test Program				HaPSet		DPOR		PCB0		PCB1		PCB2		PCB3	
name	LoC	thrds	bug type	runs	time(s)	runs	time(s)	runs	time(s)	runs	time(s)	runs	time(s)	runs	time(s)
thrift-lib-w2-5t	18.5k	3	deadlk	14	27.8	23	18.6	512(no)	247.2	26	29.2	215	146.9	871	TO
thrift-lib-w3-5t	18.5k	4	deadlk	18	27.5	733	TO	1301	TO	399	229.7	876	TO	742	TO
thrift-lib-w4-5t	18.5k	5	deadlk	22	33.7	665	TO	1111	TO	980	TO	677	TO	639	TO
thrift-lib-w5-5t	18.5k	6	deadlk	25	38.1	572	TO	899	TO	670	TO	582	TO	573	TO

context switches to trigger—with PCB1, no program failure caused by atomicity violation can be detected.

Table 3: PCB with various context bounds on `aget`

PCB0		PCB1		PCB2		PCB3		PCB4	
runs	time	runs	time	runs	time	runs	time	runs	time
286	TO	12	24.9	94	172	151	277	95	174

On `aget`, we also assessed the scalability of HaPSet by gradually increasing the number of worker threads from 2 to 10. The results in Table 4 show that the number of interleavings tested by HaPSet (before the bug is detected) grows only modestly. This is mainly due to the abstraction over specific thread ids that we use in the definition of HaPSets. This is in contrast to both DPOR and PCB, where the number of interleavings typically grow exponentially as we increase the number of threads.

Table 4: HaPSet on various threads of `aget`

2 threads		3 threads		4 threads		5 threads		6 threads	
runs	time	runs	time	runs	time	runs	time	runs	time
3	3.1	7	10.0	12	21.4	14	36.9	16	54.2
7 threads		8 threads		9 threads		10 threads			
runs	time	runs	time	runs	time	runs	time		
18	80.9	20	116.4	22	159.9	24	256.9		

6.3 Extracted `Mozilla/MySQL` Bugs

Recall that DPOR is a sound reduction whereas both PCB and HaPSet are unsound and in theory may miss bugs. Our results in previous subsections show that the reduction by HaPSet can be significant. Therefore, a natural question is, would it reduce too much to miss many bugs? To answer this question, we conducted experiments on a set of extracted bug samples. Each sample is a small program showcasing a bug extracted from the real code of Mozilla and MySQL. These examples were kindly provided by the authors of [21]. For testing purposes, we randomly inserted some shared memory accesses and locking statements to make them nontrivial for our tool.

Since the programs are small, the emphasis here is not on comparing the runtime performance, since all three competing methods can finish quickly. Rather, we would like to compare their bug-finding capability.

Table 5 shows the experimental results. The first two columns show the names and the bugs targeted by the test case. Here *atom* means an atomicity violation, *order* means an order violation, and *deadlk* means a deadlock. Eventually, the program failures caused by these atomicity/order violations are either segmentation faults or corrupted data. The next six columns compare the number of interleavings and the run time (in seconds). HaPSet not only is fast but also finds all the bugs, despite that it skips most of the interleavings explored by DPOR and PCB2. This provides strong evidence supporting our claim that, in practice, the drastic interleaving reduction achieved by HaPSet does not cause systematic testing to miss many real bugs.

Table 5: Comparison on extracted (but real) bugs

Test Program		HaPSet		DPOR		PCB2	
name	bug	runs	time	runs	time	runs	time
MysqlLog	atom	5	0.3	22	1.2	12	0.7
NodeState	order	5	0.3	22	1.5	12	0.8
Loadscript	atom	5	1.0	79	6.0	27	2.1
SeekToItem	atom	3	0.2	127	9.8	22	1.6
UpdateTimer	atom	3	0.2	128	11.2	10	1.0
FileTransport	deadlk	5	0.2	22	1.2	12	0.9
CreateThread	order	5	0.2	22	1.1	12	0.6
ReadWriteProc	order	5	1.9	175	12.0	20	2.8
OpenInputStr	deadlk	5	0.3	1409	107.7	42	3.1
HttpConnect	order	5	3.3	37	28.5	16	12.0
TimerThread	deadlk	3	0.2	101	7.3	18	1.2

7. RELATED WORK

The notion of predecessor set was first introduced by Yu and Narayanasamy [21]. Their goal was runtime failure avoidance, for which the PSets learned during testing were encoded into the program's executable. Then they designed a special-purpose microprocessor for such executables to ensure that during the production runs, the PSet constraints are always obeyed, in the hope of steering away from untested concurrency scenarios. In this paper, our goal is not runtime failure avoidance but to improve the coverage of testing. We use systematic testing to try to trigger the previously untested concurrency scenarios. To this end we have extended their idea to define the new metric called HaPSet.

As we have already explained in the previous sections, our work is related to the various systematic testing techniques [7, 12, 20, 18] based on stateless model checking. These tools are all based on brute-force exhaustive search in that their interleaving selection is not guided heuristically by any coverage metric. Among classic model checkers, SPIN [9] and Java PathFinder [8] are closely related, and they can also handle real code written in C/C++ or Java. However, they are based on the manipulation of concrete program states rather than *stateless* model checking.

CTrigger [13] and CalFuzzer [16] are two testing tools that also use dynamically collected information. CTrigger is based on the notion of access invariants, i.e. the atomicity of two consecutive memory-accessing events, and CalFuzzer is based on detecting potential data races. In comparison, the HaPSets used in our method are more general since they can characterize concurrency patterns that subsume data races and three-access atomicity violations. More importantly, in our method, the interleaving selection is systematic and each test run is guaranteed to exercise a not-yet-tested interleaving; whereas in the other two methods, interleaving selection is achieved by inserting `sleep()` statements to certain program points, to increase the odds of triggering certain interleavings. Therefore they do not have a guarantee of progress.

Our method is also related to the various runtime error detection algorithms, e.g. [15] and [17, 2]. These methods focus on analyzing a given interleaving, to either detect bugs in that interleaving or predict bugs in some other related interleavings. These methods are orthogonal to ours, since our method can systematically generate new interleavings to feed to these methods. Our method is also different from the various testing techniques based on randomization, e.g. IBM's Contest [4] and [1], although randomization can be used to diversify the input to our HaPSet learning.

8. CONCLUSIONS

We have proposed a coverage-guided systematic concurrency testing algorithm, where ordering constraints learned from the good test runs are used to guide the selection of high-risk interleavings for future test execution. We propose HaPSets to capture these ordering constraints and use them as a metric to cover important concurrency scenarios. This selective search strategy, in comparison to exhaustively testing all possible interleavings, can significantly increase the coverage of important concurrency scenarios with a reasonable cost, while maintaining the capability of detecting subtle bugs manifested only by rare interleavings.

9. REFERENCES

[1] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[2] F. Chen, T. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for java. In *International Conference on Software Engineering*, pages 221–230. ACM, 2008.

[3] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[4] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium*, page 286, 2003.

[5] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of programming languages*, pages 110–121, 2005.

[6] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Springer, 1996. LNCS 1032.

[7] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[8] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[9] G. Holzmann, E. Najm, and A. Serhrouchni. SPIN model checking: An introduction. *STTT*, 2(4):321–327, 2000.

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[11] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, Dec. 2007.

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation*, pages 267–280, 2008.

[13] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems*, pages 25–36. ACM, 2009.

[14] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Programming Language Design and Implementation*, pages 14–24, 2004.

[15] 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, 1997.

[16] K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation*, pages 11–21. ACM, 2008.

[17] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226, 2005.

[18] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking mpi programs. In *Principles and Practice of Parallel Programming*, pages 285–286. ACM, 2008.

[19] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation*, pages 1–14, 2005.

[20] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[21] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, pages 325–336, 2009.