Efficient Concurrency-Bug Detection Across Inputs

Dongdong Deng Wei Zhang Shan Lu
University of Wisconsin, Madison
dongdong@cs.wisc.edu, wzh@cs.wisc.edu, shanlu@cs.wisc.edu

Abstract

In the multi-core era, it is critical to efficiently test multi-threaded software and expose concurrency bugs before software release. Previous work has made significant progress in detecting and validating concurrency bugs under a given input. Unfortunately, software testing always faces large sets of test inputs, and existing techniques are still too expensive to be applied to every test input in practice.

In this paper, we use open-source software to study how existing concurrency-bug detection tools work for a set of inputs. The study shows that an interleaving pattern, such as a data race or an atomicity violation, can often be exposed by many inputs. Consequently, existing bug detectors would inevitably waste their bug detection effort to generate duplicate bug reports, when applied to a set of inputs.

Guided by the above study, we propose a coverage metric, Concurrent Function Pairs (CFP), to efficiently approximate how interleavings overlap across inputs. Using CFP, we have designed a new approach to detecting data races and atomicity-violation bugs for a set of inputs.

Our evaluation on open-source C/C++ applications shows that our CFP-guided approach can effectively accelerate concurrency-bug detection for a set of inputs by reducing redundant detection effort across inputs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords multi-threaded software, software testing, concurrency bugs, bug detection

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

1.1 Motivation

The rise of the multi-core era dictates the prevalence of multi-threaded software. Unfortunately, concurrency bugs widely exist in multi-threaded software [20] and have caused severe damages in the real world [28, 45, 52]. Therefore, effective bug-detection and testing techniques are needed to expose concurrency bugs before software release.

Exposing concurrency bugs is challenging, requiring not only bug-triggering inputs but also special orders of shared-memory accesses (i.e., interleavings). A multi-threaded program can take many different inputs, and follow many different interleavings while executing each input. Facing the huge input space and the even bigger interleaving space, it is impractical for in-house testing to expose all hidden bugs. How to expose as many bugs as possible given the time pressure and resource budget is a critical and open question.

The state-of-the-art techniques for in-house concurrency-bug detection and testing often involve three steps:

1. Input Design: a set of test inputs are designed to provide *code coverage*;
2. Bug Detection: for each test input, the program is executed and analyzed by a dynamic bug-detection tool that identifies potentially buggy interleavings;
3. Bug Validation: for each test input, the program is executed for several more times to exercise suspicious interleavings identified above.

Ideally, the second and third steps are repeated for every test input to provide *interleaving coverage*, so that concurrency bugs can be thoroughly detected and validated.

Much research has been done to improve each *individual* step mentioned above. For the first step, many techniques are developed to automatically generate inputs that provide good code coverage for single-threaded and multi-threaded programs [19, 54, 55]. For the second step, many tools are proposed to detect various types of buggy interleavings, including data races [7, 15, 29, 42, 50, 69], atomicity violations [6, 14, 31, 32, 61, 65], abnormal data-communication patterns [33, 59, 67, 70], and others. For the third step, different

schemes are designed to efficiently exercise suspicious interleavings [37, 38, 43, 44, 53].

Unfortunately, even with state-of-the-art techniques, bug detection (the second step) and bug validation (the third step) still each introduces 10X – 100X slowdown for each input. Applying them to every test input hence becomes unacceptably expensive, as software companies already spend more than 30% of their development resources in testing [51]. As a result, a lot of concurrency bugs inevitably escape into production runs, with concurrency-bug detection and validation techniques only applied to few inputs.

Recent work, MAPLE [68], speeds up the above process by looking at a set of inputs altogether, rather than one input at a time. Specifically, MAPLE speeds up the bug validation step (the third step) by exercising each suspicious interleaving only once across different inputs. For example, once a data race is exercised under input A, it will not be exercised again under input B. Although inspiring and promising, MAPLE does not change the bug detection step (the second step), still requiring 10X – 100X slow down to identify suspicious interleavings for each input.

Clearly, better techniques are needed to detect concurrency bugs for a set of inputs with limited in-house testing resources.

1.2 Contribution

This paper proposes a new approach to concurrency-bug detection for a *set* of inputs. This approach reduces redundant analysis that is repeated under different inputs and generates duplicate bug reports, and hence significantly improves the bug-detection performance for a set of inputs.

Our approach is based on the following observation: existing concurrency-bug detectors are inefficient for a set of inputs, because they tend to report the same suspicious interleaving under different inputs. This observation applies to race detectors, atomicity-violation detectors, and others. It is also consistent with previous work [68]: MAPLE is effective in speeding up the bug validation step largely because there are many duplicate reports across inputs.

This observation implies both opportunities and challenges. If we could reduce the redundant effort that produces duplicate bug reports, bug-detection efficiency could improve significantly for a set of inputs. Unfortunately, which bug report has duplicates across inputs is easy to check after the expensive bug-detection process, yet is very difficult to predict beforehand.

To address the challenge and exploit the opportunity, we need an easy-to-measure metric to characterize a program's interleaving space under a given input. That is, the metric should be able to approximate what sequences of shared-memory accesses could occur during the program's execution under certain input. Such a metric can approximate how interleaving spaces overlap across inputs and guide bug detection to reduce redundant analysis.

Overall, this paper makes the following contributions:

1. Identifying the inefficiency of existing concurrency-bug detectors on a set of inputs. Section 2 will discuss this observation in detail using representative real-world software and test-input sets.
2. Proposing a new interleaving-coverage metric, Concurrent Function Pair (CFP), to guide concurrency-bug detection, as well as a carefully designed algorithm to efficiently measure this metric. The CFP metric measures which functions can be executed concurrently under a given input. It strikes a nice balance between measurement complexity and interleaving-space characterization accuracy. It will be presented in detail in Section 3.
3. Designing and implementing a new data-race and atomicity-violation detection framework based on CFP. This new framework applies bug detection for a set of inputs in three steps. At Step 1, the CFP for each input is measured and the aggregated CFP for all the test inputs is calculated. At Step 2, selected inputs and selected functions under each selected input are identified so that they provide a complete coverage for the aggregated CFP. At Step 3, existing data-race and atomicity-violation detectors are applied to only the selected inputs and selected functions. The details will be presented in Section 4.

We have evaluated the new bug-detection framework on five representative open-source applications and their test input sets. The evaluation shows that our framework can effectively reduce the bug-report duplication rate[1] from 3.6–4.5 to 1.0– 2.2 for data races, and from 2.7– 5.4 to 1.1– 2.8 for atomicity violations. Overall, the speedup of bug-detection time is 1.5X– 6.2X for race detection and 1.6X– 5.6X for atomicity violation detection, with *no* false negatives among failure-inducing bugs.

2. Inefficiency of Cross-Input Bug Detection

In this section, we apply common concurrency-bug detection algorithms to representative multi-threaded software, and compare the bug reports generated across test inputs.

2.1 Methodology

Applications As shown in Table 1, this study uses 5 open-source applications that represent different types of software. All applications are written in C/C++ and use pthread library as the underlying concurrency framework.

Test inputs Click and Mozilla-js have test inputs written by their developers and released together with their source code. For Click, we use all its test inputs that do not require OS-kernel changes. For Mozilla-js, we randomly group the 20 single-threaded JavaScript requests provided by developers into 7 multi-threaded inputs.

FFT, LU, and PBZIP2 do not have publicly available test inputs. We have designed test inputs for them based on their

[1] Measured by the average number of inputs that report each bug.

App.	Description	LOC	Test Input Set		
			# Inputs	#Threads	Description
Click 1.8.0	A software router [8]	290K	6	2	Designed by Click developers
FFT	A scientific computing program from SPLASH2 [64]	1.2K	8	4	Designed based on command-line options
LU	A scientific computing program from SPLASH2 [64]	1.1K	8	4	Designed based on command-line options
Mozilla-JS m10	A JavaScript engine [36]	87K	7	4	Designed by Mozilla-JS developers
PBZIP2 0.9.4	A parallel-compression application [16]	2.0K	8	4	Designed based on command-line options

Table 1: Applications and test inputs in study

command-line options. FFT, LU, and PBZIP2 each have 8 non-trivial command-line configuration options. Therefore, we write 8 test inputs for each of them, with each input specifying a unique command-line option not specified by any other inputs. For example, our 8 FFT test inputs exercise different computation settings, such as normal FFT, inverse FFT, printing per-thread statistics, and others. As another example, our PBZIP2 inputs exercise compression, decompression, error-message suppression, compression-integrity testing, and other configurations.

To assess the quality of our test inputs, we measure their statement coverage using `gcov` [17]. The result shows that for each program, every test input covers some unique statements that are not covered by any other inputs in the test set.

Bug Detectors Our study focuses on the two most common types of concurrency bugs: data races [42, 50] and single-variable atomicity violations [14, 32, 44, 61]. A data race occurs when two threads can simultaneously access a shared variable, with at least one access being a write [50]. A single-variable atomicity violation occurs when two consecutive accesses to a shared variable from one thread is unserializably interleaved by a third access from another thread, as shown in Figure 1.

To detect these two types of bugs, we first use a run-time tool implemented in Pin [34] to collect per-thread execution traces of global/heap memory accesses and synchronization operations. We then analyze traces to detect bugs. Both detectors were implemented and used in our previous work [44, 70].

Our race detection uses a lock-set/happens-before hybrid algorithm, similar to those in many open-source race detectors [41, 56]. Specifically, two instructions are reported as a data race, if they satisfy three conditions: (1) they access the same memory location from different threads; (2) they are not protected by any common lock; and (3) they have concurrent vector timestamps calculated based on order-enforcing synchronization operations such as `barrier`, `pthread_join`, and `pthread_create`.

Our atomicity-violation detection follows an algorithm described in CTrigger [44]. Specifically, an atomicity viola-

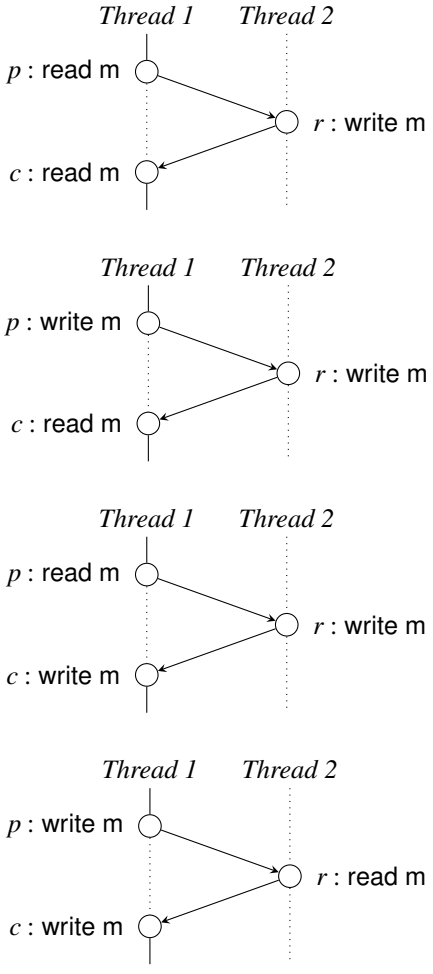

Figure 1: Four scenarios of single-variable atomicity violations that CTrigger [44] detects. "⟶" shows execution order. "m" is a shared memory location.

App.	# All Race Pairs			# All Atom. Vio.			# Buggy Race Pairs			# Buggy Atom. Vio.		
	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**
Click	3114	848	**3.6**	6145	2298	**2.7**	6	1	**6.0**	6	1	**6.0**
FFT	300	66	**4.5**	1423	369	**3.8**	28	4	**7.0**	35	5	**7.0**
LU	238	58	**4.1**	874	163	**5.4**	28	4	**7.0**	21	3	**7.0**
Mozilla	1991	481	**4.1**	2459	723	**3.4**	42	6	**7.0**	42	7	**6.0**
PBZIP2	293	65	**4.5**	499	143	**3.5**	32	8	**4.0**	39	11	**3.5**

Table 2: Inefficiency in data-race and atomicity-violation detection across inputs (DupRate measures the average number of inputs that expose the same data race or atomicity violation)

tion is reported, if three instructions p, c, and r satisfy three conditions: (1) p and c consecutively access a memory location m from one thread, while r accesses m from another thread; (2) neither locks nor order-enforcing synchronization operations can prevent r from executing between p and c; (3) the read/write access types of p, c, and r fall into one of the four scenarios shown in Figure 1.

To carry out the above bug-detection algorithms, our detectors recognize the following synchronization operations in C/C++ programs: `pthread_mutex_(un)lock`, `pthread_create`, `pthread_join`, and the `barrier` macro in SPLASH2 benchmarks (FFT and LU) [64]. Like almost all other detectors, our detectors do not recognize custom synchronization operations. This can lead to false positives.

The above data-race and atomicity-violation detection algorithms are designed so that their bug-detection results are fairly stable across runs under one input. That is, executing a program once is sufficient to obtain most, if not all, of the bug reports under one input. Executing the program under the same input for more times could occasionally produce more bug reports, but the marginal benefit is usually too small to justify the huge extra overhead in practical software testing. In our experiments, the bug detection results rarely change across runs under one input. Therefore, in this section, we only show the results obtained by executing each application once under each test input for data-race or atomicity-violation detection. We will discuss the impact of multiple bug-detection runs in Section 5.

We count each unique pair of static race instructions as one unique data race, and each unique triplet of static instructions that compose a single-variable atomicity violation as one unique atomicity violation. We tried our best to eliminate the background noise effect. For example, for Click, under each test input, we conducted our experiment using its previously stored workload trace.

2.2 Observations

Table 2 summarizes the result of applying race detection and atomicity-violation detection to 5 benchmarks. We use "All Race Pairs" and "All Atom. Vio." to represent the raw results from the two detectors. Since many races and atomicity violations do not lead to externally visible failures [4, 40], we use "Buggy Race Pairs" and "Buggy Atom. Vio." to represent failure-inducing reports from the two detectors. These would be the final results presented to the developers after the bug-validation step discussed in Section 1.

As shown in Table 2, each data race or atomicity violation is reported by 2.7 – 5.4 inputs on average across all benchmarks. The duplication rates for truly buggy reports are similar, ranging from 4 to 7 for data races and 3.5 to 7 for atomicity violations. These duplication rates could get even larger with larger test-input sets.

Theoretically, a race or an atomicity violation may lead to failures under one input and remain benign under another input. Therefore, we further investigate which races and atomicity violations could lead to software failures under which inputs. We found that the goodness/badness of a race/atomicity-violation is always the same under different inputs in the studied applications and test sets.

In conclusion, interleaving patterns, such as races and atomicity violations, overlap significantly across inputs. Concurrency-bug detection would waste substantial effort, if the detection process is not coordinated across inputs. If we have an ideal scheme that can eliminate bug-detection effort spent identifying duplicate bug reports, we can potentially speed up existing bug detection by up to 6 times for even a small set of inputs, with little or no harm to the bug-exposing capability.

3. Concurrent Function Pairs (CFP)

Terminology The *interleaving space S* of a program P includes all run-time instruction permutations that are possible for P under all possible inputs. The interleaving space of P under an input i includes those instruction permutations that are possible when P executes i.

Section 2 demonstrates the inefficiency of applying existing concurrency-bug detectors to a set of inputs. To avoid redundant analysis and reduce duplicate bug reports across inputs, we need to *predict* how different inputs' interleaving spaces overlap, so that heavy-weight bug detection can be guided to focus on unique interleavings.

```
/*Thread 1*/
foo1(){
  lock(L);
  foo2();
  unlock(L);                    /*Thread 2*/
  ...                           lock(L);
  ...                           bar();
}                               unlock(L);
```

Figure 2: An example of concurrent functions (For illustration purpose, the vertical position of each code statement in the figure represents when the statement is executed)

Section 3.1 will introduce a metric, CFP, designed to characterize the interleaving space and approximate interleaving-space overlap with low cost. Section 3.2 will describe the CFP-measurement algorithm, followed by a qualitative discussion about how CFP can guide concurrency-bug detection in Section 3.3.

3.1 Definition of CFP

Our metric design follows two principles:

1. Characterization Accuracy: it has to characterize the interleaving space with decent accuracy, so that it can guide concurrency-bug detection to reduce redundant analysis without missing many bugs.
2. Measurement Complexity: it has to be relatively cheap to measure. Otherwise, its measurement cost would outweigh its benefit in concurrency-bug detection.

Following these two principles, we have designed Concurrent Function Pairs, short as CFP. The CFP of a program P includes all unique pairs of functions that can execute in parallel with each other (i.e., concurrent), denoted as CFP^P. The CFP of a program P under a specific input i includes all unique pairs of functions that can execute in parallel with each other under input i (i.e., concurrent in i), denoted as CFP_i^P or simply CFP_i.

In the definition above, we say a pair of functions f_1 and f_2 can execute in parallel with each other, if and only if the following scenario can occur at run time: a thread t_1 is executing f_1 or a callee of f_1, while another thread t_2 is executing f_2 or a callee of f_2. For example, `foo1()` and `bar()` in Figure 2 are executing in parallel, while `foo2()` and `bar()` can never execute in parallel.

Note that CFP addresses which functions *can* execute in parallel. Whether the functions *did* execute in parallel in a particular run does not matter.

We believe CFP strikes a good balance between characterization accuracy and measurement complexity. Roughly speaking, CFP should be much cheaper to measure than data races and atomicity violations, because the number of functions is much smaller than that of memory accesses in a program. CFP should also provide a decent accuracy in characterizing the interleaving space. The content of P's interleaving space under an input i is determined by which instructions can be executed by P under i and which instructions can be executed concurrently. Since a function is a natural unit of instructions in a program, intuitively, CFP provides a decent characterization of the interleaving space. We will elaborate on these in the next two sub-sections.

```
/* .ent, .exi: function entrance, exit;
 .ent_exi.lockset: lockset protecting the critical
      section that holds both entrance and exit;
 .vec_time: vector timestamps calculated using
      order−enforcing synchronization. */
Bool concurrentFunction(f1, f2)
{
  if(f1.thread_id == f2.thread_id) return false;

  /*Can f1 start between f2's entrance and exit?*/
  if ((f1.ent.lockset ∩ f2.ent_exi.lockset) != ∅) return false;
  if (f1.ent.vec_time < f2.ent.vec_time) return false;
  if (f1.ent.vec_time > f2.exi.vec_time) return false;

  /*Can f2 start between f1's entrance and exit?*/
  if ((f2.ent.lockset ∩ f1.ent_exi.lockset) != ∅) return false;
  if (f2.ent.vec_time < f1.ent.vec_time) return false;
  if (f2.ent.vec_time > f1.exi.vec_time) return false;

  return true; /*f1 and f2 are concurrent*/
}
```

Figure 3: Pseudo code that judges concurrent functions

3.2 How to measure CFP?

3.2.1 When are two functions concurrent?

To measure CFP, we should first figure out whether two given functions, f_1 and f_2, can execute in parallel.

A naïve solution is to compare every instruction in f_1 or f_1's callees with every instruction in f_2 or f_2's callees, and see whether two instructions could execute in parallel. This is clearly too expensive.

A simpler solution, which we use in this paper, is to compare the entrances and exits of functions: if one function's entrance can execute in between the other's entrance and exit, these two functions can execute in parallel.

Clearly, to know whether a function's entrance $f_{1\text{ent}}$ can potentially execute in between another function's entrance and exit, $f_{2\text{ent}}$ and $f_{2\text{exi}}$, we need to check the synchronization operations in the program. Mutual-exclusion synchronization, such as `pthread_(un)lock`, can prevent $f_{1\text{ent}}$ from executing between $f_{2\text{ent}}$ and $f_{2\text{exi}}$, if and only if $f_{2\text{ent}}$ and $f_{2\text{exi}}$ are inside a critical section of lock l that also protects $f_{1\text{ent}}$. In addition, order-enforcing synchronization, such as `pthread_create`, `pthread_join`, and `barrier`, can prevent $f_{1\text{ent}}$ from executing between $f_{2\text{ent}}$ and $f_{2\text{exi}}$, if and

only if it forces f_{1ent} to happen before f_{2ent} or happen after f_{2exi}. Figure 3 illustrates the above algorithm.

3.2.2 The basic CFP-measurement algorithm

To compute the CFP of a program under input i, we simply need to check every pair of functions that can be executed under i to see whether they are concurrent, using the algorithm shown in Figure 3. This analysis can be conducted either statically or dynamically. In this paper, we calculate CFP_i by analyzing the run-time trace. Run-time information, such as which thread executes which functions and which instruction accesses which memory location, will allow our trace analysis to make more informed decision than static analysis.

An instrumentation tool in LLVM [27] is implemented to log the execution behavior of each thread. Each trace is a list of run-time events following a thread's execution order. There are two types of events in our trace: function entrance/exit and synchronization operation. Whenever the program enters or exits a function at run time, the corresponding thread's trace is appended with a record that specifies the unique ID of this function and whether this is an entrance or an exit. Whenever the program executes a synchronization operation, the corresponding trace is appended with a record that specifies the type of synchronization operation and some extra information. For `pthread_(un)lock`, the address of the lock variable is recorded. For `pthread_create` and `pthread_join`, the thread IDs of participating threads are recorded. For the `barrier` macro in FFT and LU, the address of the barrier variable is recorded.

Our basic trace-analysis tool computes CFP in two steps. First, for every function entrance/exit record, we calculate the set of locks protecting it, as well as its vector timestamp. The vector timestamp is calculated using only order-enforcing synchronization operations, including `pthread_create`, `pthread_join` and `barrier` in our implementation, and *not* mutual-exclusion operations like `pthread_(un)lock`, which do not force any specific order between two events. We also pay special attention to computing the locks that protect a function's entrance and exit in one critical section (i.e., `.ent_exi.lockset` in Figure 3). Specifically, we pair each function-entrance record in the trace with its corresponding exit record, which is referred to as one *function instance*. We then look for locks that are acquired before the entrance and not released until after the exit. The complexity of this step is linear to the total number of function instances.

The second step identifies all pairs of concurrent functions using the lockset and vector-timestamp information calculated above and the algorithm shown in Figure 3. To conduct this computation efficiently, our analysis first groups similar function instances together. Specifically, we consider two function instances to be similar, if they share the same (1) function ID, (2) thread ID, (3) the lockset protecting its entrance, (4) the lockset protecting both its entrance and exit in one critical section, and (5) the vector-timestamps of its entrance and exit. Based on the algorithm shown in Figure 3, two similar function instances have exactly the same set of concurrent functions and hence only need to be processed once. After grouping similar function instances, we simply go through every pair of functions and check whether they have at least one pair of instances that are concurrent with each other. The complexity of the last step is quadratic to the number of unique static functions in the trace.

```
void lu0(double* a, int n)
{
  int j, k;
  for (k=0; k<n; k++) {
    for (j=k+1; j<n; j++) {
      daxpy(...,...,...,...);
      /*Parameters omitted for presentation simplicity*/
    }
  }
}
```

Figure 4: Functions concurrent with `lu0` must be concurrent with `daxpy` (`n` is a positive integer).

3.2.3 Optimization for CFP measurement

Although CFP measurement is much cheaper than concurrency-bug detection, it could still take time for large long-running programs with many function calls. Therefore, we further optimize the basic algorithm in two ways.

Optimization 1: skip functions that only access stack variables. This is a generic optimization that is also conducted by our concurrency-bug detectors discussed in Section 2.1. If f only accesses stack variables,[2] there is no chance it will help guide concurrency-bug detection.

Optimization 2: skip functions that inherit their callers' concurrent functions. This is an optimization specially designed for CFP measurement. It does not work for generic concurrency bug detection. This optimization can be demonstrated by a real example from LU shown in Figure 4. In this example, we can guarantee that every function concurrent with `lu0` is also concurrent with `daxpy` for two reasons: (1) whenever `lu0` is executed, it invokes `daxpy`; and (2) `lu0` does not contain any synchronization operations. Therefore, there is no need to record or analyze the entrance/exit of `daxpy` inside `lu0` for CFP-measurement. We simply need to append the set of functions that are concurrent with `lu0` to that of `daxpy` at the end of our analysis.

Formally speaking, suppose function f_1 calls function f_2 through instruction i. We can guarantee that every function concurrent with f_1 is also concurrent with f_2, if f_1 and i satisfy two conditions: (1) i post-dominates the entrance instruction of f_1, which guarantees that f_1 always invokes

[2] Theoretically, developers could make a stack variable shared among threads. Since it is rare and not recommended, it is not considered here.

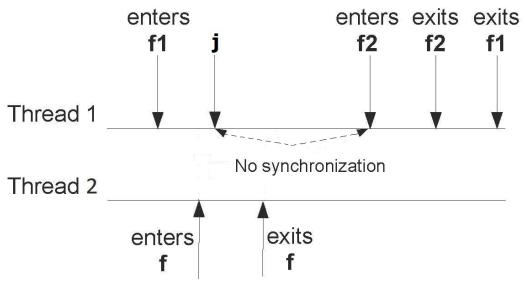

Figure 5: f can execute in parallel with f_2 in another run

f_2; (2) no synchronization operation is executed by f_1 or its callees, except for f_2 through the call-site i.

We briefly prove this concurrency-inheritance relationship as follows. Suppose a function f is concurrent with f_1. By definition, there exists a run, during which an instruction of f or f's callees is executed simultaneously with an instruction, denoted as j, of f_1 or f_1's callees (shown in Figure 5). If j is an instruction inside f_2, f must be concurrent with f_2. If j is outside f_2, the two conditions mentioned in the previous paragraph imply that there exists an invocation or a return of f_2, so that no synchronization operations are executed between it and that particular instance of j. Therefore, any code region (e.g., f) that can execute simultaneously with the latter can also execute simultaneously with the former, which proves that f is also concurrent with f_2. The high-level idea of this proof is depicted in Figure 5.

Given the above proof, how to conduct Optimization 2 is straightforward. Our static analysis goes through a program's call graph for several passes. It identifies every pair of caller f_1 and callee f_2 that satisfy the two optimization conditions mentioned above. In our current algorithm, we further check whether the callee f_2 satisfies the optimization conditions with all its caller functions. If so, we remove all LLVM instrumentation described in Section 3.2.2 that logs the entrance and the exit of f_2. Consequently, not only the tracing time is shortened, the trace size and trace-analysis time are also reduced.

Our current algorithm does not apply any optimization to a function that only has some, but not all, of its caller functions satisfying the optimization conditions. This design can be changed in the future by differentiating different callers of a function. That is, as long as f_2 satisfies the optimization conditions with one of its caller f_1, we can skip the logging and trace analysis discussed in Section 3.2.2 for dynamic instances of f_2 that are invoked by f_1. More performance improvement will be achieved in that way.

There are some caveats in our optimization.

First, we assume that every loop in a program will execute at least one iteration. Take Figure 4 as an example. Without any knowledge about the input range, static analysis cannot determine whether the loops will always execute at least one iteration, and hence cannot guarantee that every instance of `lu0` would invoke `daxpy`. Therefore, unless we use this at-least-one-iteration heuristic, static analysis will lose the opportunity to optimize the `daxpy` instrumentation. Of course, this heuristic could lead to CFP false positives: some concurrent function pairs may never execute in parallel. However, we believe the optimization benefit of this heuristic significantly outweighs any potential negative impact, because in our experience the chance of CFP false positives is rare. In addition, false positives in CFP measurement would at worst cause unnecessary and redundant bug detection, and hence slow down our CFP-guided bug detection. It would *not* directly lead to false positives or negatives in concurrency-bug detection.

Second, our analysis only considers `pthread`-related operations and the `barrier` macro as synchronization operations. As most existing tools, we do not consider custom synchronization, which could again lead to false positives in CFP measurement.

The above optimization is useful for many programs, because many functions in multi-threaded programs do not contain synchronization operations. In addition, some utility functions like `daxpy` in LU are major sources of dynamic function instances. Section 5.8 will evaluate the impact of this optimization in detail.

There are other optimization opportunities for CFP measurement. For example, some functions, such as those executed by the main thread before any child thread is created, are not concurrent with any functions. Future work can try using static analysis to identify these functions, and then skip these functions during logging and CFP-trace analysis. This will lead to smaller CFP traces and faster CFP measurement. We leave further optimization to future work.

3.3 Can CFP guide bug detection?

As discussed earlier, CFP_i^P can characterize the interleaving space of program P under input i to some extent. In the following, we discuss in detail the relationship between CFP_i^P and the set of concurrency bugs reported from P under i.

3.3.1 Data races and CFP

A data race occurs when two memory accesses, with at least one write, to the same memory location from two threads could occur concurrently without synchronization in between.

If two functions f_1 and f_2 are not concurrent, such as `foo2` and `bar` in Figure 2, no data race can be found between instructions from them, because no instruction in f_1 can execute in parallel with instructions in f_2. On the other hand, if f_1 and f_2 are concurrent, such as `main` and `SlaveStart` in Figure 6, they likely contain data races when they read and write the same memory locations. Furthermore, when a concurrent function pair is in multiple inputs' CFPs, applying race detection on these inputs is at a high risk of generating duplicate race reports. For example, 7 out of 8 test inputs of FFT contain {`main`, `SlaveStart`}

```
/*Thread 1*/                    /*Thread 2*/
void main(...){                 void SlaveStart(...){
  ...                             ...
  printf("End at %f", Gend);      Gend = time();
  ...                             ...
}                               }
```

Figure 6: A data-race example from FFT

```
/*Thread 1*/                    /*Thread 2*/
js_InitAtomState(...){          js_FreeAtomState(...){
  state->table = ...;             ...;
  if(!state->table)               state->table = NULL;
    return false;                 ...;
}                               }
```

Figure 7: An atomicity-violation example from Mozilla

in their CFPs. Consequently, the data race shown in Figure 6 is repeatedly reported under 7 inputs.

Clearly, CFP is useful in coordinating race detection across inputs. For example, if we select inputs so that their CFPs do not overlap, applying race detection to them can guarantee not to generate duplicate bug reports. However, this could cause too few inputs to be selected and hence miss bugs. For example, suppose the test-input set contains only two inputs with overlapped CFPs. If we select both inputs, duplicate bug reports could occur due to the CFP overlap; if we only select one input, the concurrent function pairs uniquely covered by the other input will be missed in bug detection. We will discuss better ways to use CFP in Section 4.

3.3.2 Atomicity violations and CFP

Atomicity violation occurs when a sequence of memory accesses from one thread is unserializably interleaved by memory accesses from another thread [6, 14, 31, 32, 61, 65].

Similar to that in data-race detection, if two functions f_1 and f_2 are not concurrent, no atomicity violation can be found between instructions from them, because no instruction in f_1 (or f_2) can execute between a sequence of instructions from f_2 (or f_1). On the other hand, if f_1 and f_2 are concurrent, such as `js_InitAtomState` and `js_FreeAtomState` in Figure 7, they may contain atomicity violations. Furthermore, when a concurrent function pair is in multiple inputs' CFPs, applying atomicity-violation detection on these inputs risks generating duplicate atomicity-violation reports. For example, 7 out of 7 test inputs of Mozilla contain {`js_InitAtomState`, `js_FreeAtomState`} in their CFPs. Consequently, the atomicity violation shown in Figure 7 is repeatedly reported under 7 inputs.

4. CFP-Guided Bug Detection

4.1 Overview

Our CFP-guided bug detection aims to improve the detection efficiency over a set of inputs by eliminating redundant analysis. At a high level, this is achieved by turning on bug detection only when the program executes selected functions under selected inputs. The process of input selection and function selection is guided by CFP and has two goals for a set of inputs I:

1. Lower the chance of missing bugs: the bug detection process should provide a complete coverage of the CFP_I, the union of all test inputs' CFPs. We will refer to CFP_I as aggregated CFP.
2. Lower the chance of duplicate bug reports: the bug detection process should avoid repeatedly analyzing the same pair of concurrent functions across inputs, as long as this does not prevent us from achieving the first goal.

Consequently, the principle of our input and function selection is to skip the bug detection over a function f, if f does not contribute to previously unanalyzed concurrent function pairs. As a simple example, suppose there are two inputs in the test set, i_1 and i_2. Suppose CFP_{i_1} is $\{\{f_1,f_2\}\}$ and CFP_{i_2} is $\{\{f_1,f_2\},\{f_2,f_3\}\}$. If bug detection is applied to i_1 first, it should then skip f_1 under i_2, because f_1 does not contribute to any unanalyzed concurrent function pair. On the other hand, if bug detection is applied to i_2 first, it could then completely skip i_1 following the above principle.

Specifically, our CFP-guided concurrency-bug detection includes three steps for a set of inputs:

1. Compute the CFP of each input and get the aggregated CFP of the whole input set.
2. Select inputs and select functions for each selected input.
3. Apply a race detector or an atomicity-violation detector to selected functions under each selected input.

Next, we discuss these three steps one by one.

4.2 Step 1: CFP measurement

As discussed in Section 3.2, our CFP measurement contains several phases. The first phase is static analysis. We use static analysis to identify functions that are guaranteed to have the same concurrency property with their caller functions, as discussed in Section 3.2.3. We then statically instrument the program using LLVM to log the entrance/exit of every function, except those identified above, and every synchronization operation. This static analysis phase is conducted only once for all inputs.

The second phase calculates the CFP for each test input. As discussed in Section 3.2.2, we execute the instrumented program under an input once, and then analyze the run-time trace to identify concurrent function pairs. Note that, what we get from the trace analysis is not the final CFP yet. Since

Input1: CFP_1=$\{\{f_1,f_2\}, \{f_2,f_3\}, \{f_2,f_4\}, \{f_4,f_5\}\}$
Input2: CFP_2=$\{\{f_1,f_2\}, \{f_3,f_4\}, \{f_3,f_5\}\}$
Input3: CFP_3=$\{\{f_2,f_3\}, \{f_3,f_4\}\}$
$CFP_{Aggregated}$ =$\{\{f_1,f_2\}, \{f_2,f_3\}, \{f_2,f_4\}, \{f_4,f_5\}, \{f_3,f_4\}, \{f_3,f_5\}\}$

Step 1:
Selected input $--$ Input1
Selected functions $--$ $\{f_1, f_2, f_3, f_4, f_5\}$
$CFP_{Uncovered}$ = $\{\{f_3,f_4\}, \{f_3,f_5\}\}$

Step 2:
Selected input $--$ Input2
Selected functions $--$ $\{f_3, f_4, f_5\}$
$CFP_{Uncovered}$ = $\emptyset$

Figure 8: A toy example of input/function selection

we do not log functions that inherit their callers' concurrent functions, we need to compute the concurrent function pairs that involve these functions, as discussed in Section 3.2.3. This gives us the final CFP for each input.

The third phase calculates the aggregated CFP. It simply takes the union of every individual input's CFP.

Note that, during the second phase, we calculate CFP based on the log collected from only one run. Theoretically, different runs under the same input could produce different logs, and hence different CFPs. However, the concurrency relationship analyzed here is usually stable across runs, similar with that in data-race detection [50] and atomicity-violation detection [44]. Our experimental results also show that the reported CFP is very stable across runs. Even in the worst cases, we see fewer than 0.5% of CFP fluctuation among tens of runs. Therefore, we only run each program **once** under each input to collect the CFP, which is consistent with previous bug-detection work [44, 50].

4.3 Step 2: input and function selection

We aim to select the smallest set of inputs to cover the aggregated CFP, which unfortunately is an NP-hard problem. To efficiently solve it, we use a greedy algorithm that provides approximate results.

Specifically, our algorithm first selects the input that covers the most concurrent function pairs among all inputs. It then keeps selecting the input that covers the most uncovered pairs, until all pairs in the aggregated CFP are covered.

Function selection is straightforward once we know how to select inputs. During the above input-selection process, for every selected input i, we know which concurrent function pairs covered by i are not covered by previously selected inputs. Functions involved in those concurrent function pairs become the selected functions for i.

A toy example that demonstrates the input/function selection process is shown in Figure 8.

4.4 Step 3: guided bug detection

At this step, we simply apply existing concurrency-bug detectors to the selected inputs and selected functions.

The only non-trivial issue here is that most run-time concurrency-bug detectors conduct bug detection unselectively on all executed functions. Small modifications are needed to integrate an existing bug detector into our bug-detection framework.

In our current implementation, we slightly modified the Pin-based execution-tracing tool described in Section 2.1 to take a command-line input file. This input file contains the instruction-address range of every function identified by Step 2, with each range represented by the function entrance instruction address and exit instruction address. Our modified tracing tool only logs memory accesses whose instruction addresses fall into one of the ranges specified by the input file. We then apply the same trace-analysis algorithms described in Section 2.1 to detect data races and atomicity violations. We believe similar modifications can be easily done for many other existing concurrency-bug detectors.

4.5 Bug-detection quality assessment

The quality of bug detection is usually measured using three metrics: (1) performance; (2) false negatives; and (3) false positives. In the following, we qualitatively compare our CFP-guided bug detection (short as *CFP* below) with traditional bug detection that applies unchanged existing bug detectors to each and every test input (short as *Full* below).

In terms of false positives, *CFP* will never introduce more false positives than *Full*.

In terms of performance, the third step of *CFP* is clearly faster than *Full*, because only selected inputs and selected functions are involved. The more inputs and the more CFP-overlap the inputs have, the more advantage *CFP* has over *Full* here. However, the first step and the second step of *CFP* incur extra cost that is not incurred by *Full*. In reality, the benefit of *CFP* tends to significantly outweigh its cost, because CFP measurement takes much less time than concurrency-bug detection. We will see more detailed quantitative results in Section 5.

There are two ways to measure false negatives. One is to ignore the practical resource limit and measure how many unique bugs can be discovered given unlimited resource in unlimited amount of time. The other one, which is more realistic, is to measure how many unique bugs can be discovered with limited resources in a limited amount of time. Since the resource limitation is a real concern for a set of inputs, the second way of measurement is more suitable in this paper's context. We believe *CFP* will incur fewer false negatives under this way of measurement, because *CFP* would spend less resource in producing duplicate bug reports than *Full*. We will discuss more about this in Section 5.

Of course, it is worth pointing out that, with unlimited resources, *CFP* will inevitably have more false negatives

than *Full*. In general, *CFP* guarantees to apply concurrency-bug detection to every pair of concurrent functions under at least one input. However, some bugs may only be detected when a pair of concurrent functions are executed under a special input. For example, some bugs may hide in special basic blocks that are not always exercised when its calling function is executed. As another example, different inputs could bring different states to the same code region, causing two instructions to access different memory locations under one input and the same memory location under a different input. In addition, there is an extra source of false negatives for *CFP* atomicity-violation detection. Sometimes, the intended atomic region of an atomicity violation involves code statements from more than one function. Concurrent function pairs cannot well predict the existence of these bugs.

Overall, *CFP* is not a panacea. It improves the performance of cross-input bug detection at the risk of missing bugs when there is abundant resource to conduct full-blown bug detection on all inputs. It is a good fit for bug detection and testing when the resource is limited, which is the common case in practice. In fact, in reality, software companies may not have the resource and time to finish even the CFP-guided bug detection. In this case, the CFP metric can provide developers a quantitative measurement about the completeness of in-house testing. We believe this metric will be useful for multi-threaded software testing, just like how statement coverage and branch coverage are crucial for sequential software testing.

5. Evaluation

5.1 Methodology

We run our experiments on an 8-core Intel Xeon machine and LLVM 2.8 compiler. The applications and test-input sets used here are identical to those described in Section 2.1.

Our evaluation will compare our CFP-guided concurrency-bug detection approach, short as *CFP*, with the traditional approach that applies full-blown detection to every input, short as *Full*. We will try both data-race detection and atomicity-violation detection, using the detectors described in Section 2.1. The only difference is that the detectors used by *CFP* can be configured to monitor only specified functions, as discussed in Section 4.4.

Our evaluation will compare *CFP* with *Full* from several aspects: performance, false negatives, bug-report duplication rate, and trace size. The impact of our optimization algorithms and other details of *CFP* will also be evaluated.

5.2 Overall results

As shown in Table 3 and Table 4, our CFP-guided approach can significantly improve the concurrency-bug detection performance, with few to no false negative and huge reduction in bug-detection trace size. The impact on race detection and atomicity-violation detection is similar.

App.	Speedup	False Neg. Rate		# False Neg.		Trace Re-
	(X)	All	Buggy	All	Buggy	duction (%)
Click	**3.5**	2.0%	0%	17	0	82%
FFT	**6.2**	4.5%	0%	3	0	82%
LU	**4.9**	1.7%	0%	1	0	76%
Mozilla	**1.5**	2.9%	0%	14	0	39%
PBZIP2	**3.8**	3.1%	0%	2	0	75%
Average	**4.0**	2.8%	0%	7	0	71%

Table 3: Overall results of CFP-guided race detection, with the traditional *full* race detection as the baseline. The baseline bug counts are shown in the "# All Race Pairs (Unique)" column and "# Buggy Race Pairs (Unique)" column of Table 2.

App.	Speedup	False Neg. Rate		#False Neg.		Trace Re-
	(X)	All	Buggy	All	Buggy	duction (%)
Click	**2.5**	2.0%	0%	46	0	82%
FFT	**5.6**	3.0%	0%	11	0	82%
LU	**4.8**	4.3%	0%	7	0	76%
Mozilla	**1.6**	1.9%	0%	14	0	39%
PBZIP2	**2.4**	4.2%	0%	6	0	75%
Average	**3.4**	3.1%	0%	17	0	71%

Table 4: Overall results of CFP-guided atomicity-violation detection, with the *full* atomicity-violation detection as the baseline. The baseline bug counts are shown in the "# All Atom. Vio. (Unique)" column and "# Buggy Atom. Vio. (Unique)" column of Table 2.

In terms of performance, *CFP* achieves 4.0X and 3.4X speedup on average for data-race and atomicity-violation detection, respectively, with the best performance improvement achieved for FFT and the worst for Mozilla.

In terms of false negatives, *CFP* does not miss *any* failure-inducing data race or atomicity violation, as shown by the "Buggy" columns in Table 3 and Table 4. As discussed in Section 2, the race detector and atomicity-violation detector also report many data races and atomicity violations that do not lead to software failures. When we consider all these reports, both failure-inducing and non-failure-inducing ones, *CFP* incurs 1.7% to 4.5% false negative rate, a small number considering the speedup, as shown by the "All" columns in Table 3 and Table 4. Among all benchmarks, Click generates the most data-race and atomicity-violation bug reports, as shown in Table 2. Consequently, it also incurs the most false negatives under *CFP*.

CFP also significantly reduces the trace size in concurrency-bug detection. As discussed in Section 2.1, both the race detector and the atomicity-violation detector used in our implementation analyze execution traces to discover bugs. By selecting inputs and functions, *CFP* reduces the trace size by 71% on average for 5 benchmarks.

Overall, the above results demonstrate that our CFP-guided approach can significantly speed up the bug-detection process for a set of inputs during in-house bug detection and testing, with negligible effect on the bug-detection coverage.

5.3 Input and function selection

Our CFP-guided approach runs faster than *Full* bug detection, because it only applies bug detection to selected inputs and selected functions, as shown in Table 5.

	#Inputs		#Functions		Trace Size (MB)	
App.	Full	CFP	Full	CFP	Full	CFP
Click	6	4	3689	724	94	17
FFT	8	1	135	21	1011	182
LU	8	1	122	18	1012	256
Mozilla	7	5	1583	857	10	6
PBZIP2	8	2	782	135	132	33

Table 5: Input/function selection and trace-size changes ("#Functions" and "Trace Size" are both aggregated across inputs)

As we can see, 1 – 5 inputs are selected for the 5 benchmarks. For FFT, LU, and PBZIP2, only 1 or 2 inputs are sufficient to provide a complete CFP coverage.

Apart from input selection, our CFP approach also selects which functions to monitor while executing each input. The "#Functions" column in Table 5 shows the sum of the number of static functions that are executed and monitored by bug-detection tools for every input. As we can see, the number of functions monitored in *CFP* across inputs is only 15% – 54% of those in *Full*. In Click, even though 4 out of 6 inputs are selected, only few functions are monitored when executing some of the selected inputs. As a result, *CFP* only monitors 20% of functions monitored by *Full* across inputs, which leads to over 80% of trace-size reduction shown by the "Trace Size" columns in Table 5.

Overall, for all benchmarks, CFP can effectively guide us to identify selective inputs and functions for concurrency-bug detection. Note that the input selection and function selection are conducted only once for a program and a set of inputs. Later on, both race detection and atomicity-violation detection will use the same selection result.

5.4 Bug-report duplication rate

As discussed in Section 2.2, directly applying traditional concurrency-bug detection to a set of inputs is inefficient, because many duplicate bugs will be reported. Table 6 shows that our CFP-guided approach can effectively reduce the bug-report duplication rate. Specifically, the average number of inputs under which each race is reported drops from 3.6 – 4.5 to 1.0 – 2.2 for the five benchmarks. The duplication rate for atomicity violation also drops from 2.7 – 5.4 to 1.1 – 2.8. In fact, the duplication rate of *CFP* drops to below 2 for all benchmarks except for Mozilla.

The reason that we failed to decrease the duplication rate to 1 can be explained by an example. Suppose we choose input i_1 to cover a concurrent function pair $\{f_2, f_3\}$, and i_2 to cover pairs $\{f_1,f_3\}$ and $\{f_1,f_2\}$. A race between instructions in f_2 and f_3 could be reported by both i_1 and i_2, because f_2 and f_3 are monitored in both inputs. Future work can design better monitoring schemes or input/function selection schemes to further decrease the duplication rate.

	Data Race		Atomicity Violation	
App.	Full	CFP	Full	CFP
Click	3.6	1.2	2.7	1.8
FFT	4.5	1.0	3.8	1.3
LU	4.1	1.0	5.4	1.1
Mozilla	4.1	2.2	3.4	2.8
PBZIP2	4.5	1.2	3.5	1.2

Table 6: Bug-report duplication rate ($\frac{\text{\# all reports}}{\text{\# unique reports}}$)

Overall, our CFP-guided approach significantly reduces the bug-report duplication rate across inputs for both data races and atomicity violations. This reduction will naturally lead to more effective bug detection — less detection time and similar detection coverage. It will also relieve the developers from identifying and discarding duplicate bug reports.

5.5 False negatives

Our CFP-guided detection has missed only 1.7 – 4.5% of all races and atomicity violations reported by *Full*. More importantly, it incurs **no** false negative among failure-inducing races and atomicity violations. Almost all false negatives occur when different inputs cover different basic blocks in a function and the selected input happens to miss those basic blocks containing data races or atomicity violations. Future work can potentially refine the granularity of CFP metric to achieve fewer false negatives. We will discuss this in Section 5.9.

Considering the 1.5X – 6.2X speedup achieved by our CFP-guided bug detection, the above false negative rate is very low and is a worthy tradeoff in a practical setting where developers only have a limited amount of time for bug detection and testing.

5.6 Performance breakdown

Table 7 shows the detailed performance breakdown among the three steps in CFP-guided bug detection. All the numbers shown in the table are normalized, where 1 is the total time of executing a benchmark through all test inputs without any monitoring or instrumentation (shown in the "Base" column).

The Step 1 of our CFP-guided approach measures the CFP of every input. In general, it is fast, incurring less than 100% of overhead comparing with simply running the program without any monitoring or instrumentation. Mozilla incurs the largest overhead for several reasons. Most importantly, it contains many small functions with just a few accesses to heap/global variables. These functions lead to considerable overhead in CFP measurement. Furthermore, many functions in Mozilla execute synchronization operations, either directly or indirectly through their callees. As a result, our CFP-measurement optimization does not help Mozilla as much as it does for some other benchmarks.

App.	Base (sec.)	Data Race CFP Step1	CFP Step2	CFP Step3	**CFP Tot.**	**Full Tot.**	Atomicity Violation CFP Step1	CFP Step2	CFP Step3	**CFP Tot.**	**Full Tot.**
Click	1.71	1.79	0.04	23.72	25.55	87.27	1.79	0.04	10.63	12.46	30.40
FFT	0.03	1.94	0.01	170.66	172.61	1078.22	1.94	0.01	109.95	111.90	628.71
LU	0.94	1.89	0.0005	41.53	43.42	212.76	1.89	0.0005	41.26	43.15	207.12
Mozilla	0.30	4.48	0.10	60.39	64.88	99.24	4.48	0.10	39.85	44.43	69.88
PBZIP2	5.40	1.06	0.0006	4.87	5.93	19.84	1.06	0.0006	1.38	2.44	5.93

Table 7: Performance breakdown of concurrency-bug detection (all the numbers are normalized by the time in "Base" column, which shows the total time in seconds of running a benchmark through all test inputs without any instrumentation or bug detection)

Note that measuring the CFP of every input ("CFP-Step1") takes much less time than running full-blown bug detection on every input ("Full-Tot."). For example, the former takes only 0.2%–5.3% of the time of full-blown race-detection for all benchmarks. This confirms that CFP has successfully stuck to the "simplicity" design principle.

The Step 2 selects inputs and functions. It takes the smallest amount of time among all three steps.

The Step 3 applies concurrency-bug detection to selected inputs and functions. Not surprisingly, it is the most time-consuming step. Of course, it is significantly faster than full-blown concurrency-bug detection ("Full Tot." columns).

Finally, "CFP Tot." and "Full Tot." columns compare the total time of our CFP-guided detection with that of traditional full-blown bug detection. "CFP Tot." is the sum of the above three steps. As we can see, our CFP-guided approach is significantly faster than the traditional approach, which is also illustrated in Table 3 and Table 4.

The speedup of our CFP-guided approach is mainly determined by the number of selected inputs and functions. Intuitively, the fewer selected, the faster our CFP-guided bug detection is. Strictly speaking, the CFP-measurement time would also affect the speedup. However, since it takes much less time than bug detection (Step 3), its impact is negligible.

For example, our CFP-guided race (atomicity-violation) detection is more than 6 (5) times as fast as the full race (atomicity-violation) detection for FFT. The reason is that only 1 out of 8 test inputs is selected. On the other hand, only about 1.6X speedup is achieved by our CFP-guided approach for Mozilla, because 5 out of 7 test inputs are selected and about 54% of functions still need to be analyzed across inputs. Note that, our CFP-guided approach achieves about 3X speedup in Click, although as many as 4 out of 6 inputs are selected. The reason is that only about 20% of functions are selected across inputs.

As can be seen from the performance breakdown, although the first two steps of our CFP-guided approach incurs additional cost, this cost can be easily compensated by the reduction of bug-detection time in the third step of our CFP-guided approach. In addition, the CFP-measurement and input/function-selection results of the first two steps can be shared by race detection and atomicity-violation detection. The speedup will become more significant when we consider these two together.

5.7 Multiple bug-detection runs for each input

As discussed in Section 2.1, by default, we execute each application only once under each (selected) input for data-race or atomicity-violation detection, which is the common practice in resource-limited software testing. In this subsection, we investigate the impact of multiple bug-detection runs under each input. We will use the subscript "M" to differentiate these new settings from the default settings used earlier, with the details shown in Table 8. Since **no** extra failure-inducing bug reports are generated under multi-run settings, we only discuss the results about all bug reports below.

	# of runs for each application under each input
Full	1
CFP	1 in Step 1; 1 in Step 3
Full_M	10
CFP_M	1 in Step 1; 10 in Step 3
CFP_M^+	10 in Step 1; 10 in Step 3

Table 8: Different settings evaluated by our experiments. Full_M, CFP_M, and CFP_M^+ conduct multiple bug-detection runs for each input, and are evaluated in Section 5.7.

App.	Data Race Full_M	CFP_M	$\text{CFP}_M{}^+$	Atomicity Violation Full_M	CFP_M	$\text{CFP}_M{}^+$
Click	6	3	3	10	5	5
FFT	0	0	0	0	0	0
LU	0	0	0	0	0	0
Mozilla	3	1	6	5	2	8
PBZIP2	1	1	1	2	2	2

Table 9: The numbers of **extra** bug reports generated by $Full_M$ (compared with *Full*), CFP_M and CFP_M^+ (compared with *CFP*).

As we can see in Table 9, extra bug-detection runs produce few extra bug reports. For example, comparing $Full_M$ with *Full*, only 0 – 10 extra bug reports are generated by the extra bug-detection runs. Click produces the most extra bug

App.	CFP-Trace Size (MB)				CFP-Step1 Time (normalized)			
	No Opt.	Opt.1	Opt.2	Opt.1+2	No Opt.	Opt.1	Opt.2	Opt.1+2
Click	93.40	93.24	9.28	9.25	18.36	17.24	1.95	1.79
FFT	0.95	0.94	0.93	0.92	3.10	2.89	2.44	1.94
LU	836.27	835.43	4.85	4.31	74.13	72.65	1.97	1.89
Mozilla	10.21	8.41	8.68	7.81	7.44	5.78	5.81	4.48
PBZIP2	0.98	0.93	0.87	0.67	1.14	1.09	1.09	1.06

Table 10: The effect of CFP-measurement optimization (CFP-Trace Size and CFP-Step1 Time are both the summation across all inputs; CFP-Step1 Time is normalized as that in Table 7)

reports: 6 extra data races and 10 extra atomicity violations, which contribute to only 0.7% and 0.4% increase of the *Full* bug-report numbers.[3] This result shows that conducting one bug-detection run for each selected test input is sufficient in resource-limited software testing environment.

Click and Mozilla are the only two applications where the numbers of extra bug reports are different among $Full_M$, CFP_M, and CFP_M^+.

In Mozilla, the CFP_M^+ setting discovers 3 extra pairs of concurrent functions during its CFP measurement. Consequently, CFP_M^+ discovers more concurrency bugs than CFP_M. On the other hand, no extra concurrent function pair is discovered by CFP_M^+ in Click. As a result, CFP_M^+ and CFP_M generate the same number of extra bug reports.

In both Mozilla and Click, CFP_M generates fewer extra bug reports than $Full_M$. Some of these are caused by concurrent function pairs that are not identified by one-run of CFP measurement. Some of these are located in rarely executed paths. By exercising the same functions for multiple times under more inputs, $Full_M$ has more chances to discover these bugs.

Overall, full-blown concurrency-bug detection can benefit a little bit more from extra bug-detection runs for some applications than our CFP approach. However, this benefit is usually too small to justify the extra cost incurred by extra bug-detection runs during resource-limited testing.

5.8 Other results

Optimization effect Section 3.2.3 presents two optimizations for CFP measurement. Table 10 shows their impact on reducing CFP-trace size and CFP-measurement time.

Overall, the combined optimization effect is significant (comparing "Opt.1+2" with "No Opt." in Table 10). Since both optimizations skip many function entrance/exit records during CFP measurement, up to 99% reduction is achieved for CFP-measurement trace size and up to 98% reduction is achieved for total CFP-measurement time. Without these two optimizations, CFP measurement can take up to 74 times the base-line program execution time. With these optimizations, the slowdown is mostly below 2X.

[3] The total numbers of bug reports generated by *Full* are shown in the "Unique" columns of Table 2

Clearly, the optimization effects are different for different applications. For these five benchmarks, Optimization 2 is more effective than Optimization 1. However, neither one of them can replace the other. For example, Optimization 1 is the most effective for Mozilla, where 22% of CFP-measurement time and 18% of CFP traces are saved. The other four benchmarks all have fewer functions that only access stack variables, and hence benefit much less from Optimization 1. Optimization 2 is the most effective for LU and Click, reducing more than 90% of CFP traces and saving more than 90% of trace-analysis time.

The size of CFPs One might wonder how many concurrent function pairs are there for a benchmark. Table 11 presents the size of aggregated CFP for each benchmark — the total number of unique pairs of concurrent functions across all inputs. For reference, the total number of classes, the total number of static functions of each program and the total number of unique functions executed by test inputs are also listed. Naturally, programs with more (executed) functions have more concurrent functions. At the same time, many executed functions clearly are not concurrent with each other due to synchronization.

App.	# Class	# Fun.	# Executed Fun.	Size of Aggregated CFP
Click	133	1889	964	2504
FFT	0	23	23	42
LU	0	21	21	62
Mozilla	0	1050	268	22970
PBZIP2	0	125	125	426

Table 11: Total number of classes, functions and CFPs

The Benefit of Function Selection The performance improvement of our CFP approach over the Full approach comes from two sources: the reduction in the number of test inputs and the reduction in the number of functions monitored during Step 3. To better understand the contributions from these two sources, we evaluate the total testing time with all functions monitored for selected inputs during Step 3, and compare it with the Full approach and the CFP approach.

As shown in Table 12, even without selecting functions, input selection alone can provide significant speedup over the *Full* approach — 3.5X for race detection and 3.1X for

App.	w/ function selection		w/o function selection	
	Race	Atom. Vio.	Race	Atom. Vio.
Click	3.5X	2.5X	1.9X	1.4X
FFT	6.2X	5.6X	6.2X	5.6X
LU	4.9X	4.8X	4.9X	4.8X
Mozilla	1.5X	1.6X	1.3X	1.3X
PBZIP2	3.8X	2.4X	3.2X	2.3X
Average	4.0X	3.4X	3.5X	3.1X

Table 12: The speedup over *Full* approach with and without function selection.

atomicity-violation detection on average. Function selection further improves the performance, achieving 4.0X speedup for race detection and 3.5X speedup for atomicity-violation detection on average.

The impact of function selection is different for different benchmarks. Click gets the most benefit. Without function selection, the testing time almost doubles for Click. On the other hand, function selection makes no difference for FFT and LU. The reason is that only one input is selected for FFT and LU, respectively. Consequently, all functions are selected to test this input under the CFP approach.

Overall, both function selection and input selection are useful in shortening the testing time. Among these two, input selection has a larger impact for our benchmarks.

5.9 Limitations and discussion

What about random input selection? An alternative of our CFP-guided approach is to randomly select inputs for concurrency-bug detection. We believe that our CFP-guided approach has advantages for several reasons.

First, our CFP-guided approach allows us to select not only inputs but also functions for concurrency-bug detection. The capability of selecting functions for a selected input is crucial for bug-detection efficiency (Click is an example for this). This cannot be achieved by random input selection.

Second, random input selection incurs unpredictable false negatives. It may happen to expose many bugs and may as well cause many false negatives. For example, in FFT, when we randomly select one input from the 8-input test set, there is **87.5**% probability that applying race detection to this selected input would incur a false-negative rate between **41**% and **100**%, comparing with the full-blown detection on all inputs. In contrast, our CFP-guided approach will deterministically select the input with the largest CFP and incur only 4.5% false-negative rate. As another example, in PBZIP2, each failure-inducing bug report can be exposed by fewer than half of all the test inputs. In fact, there are two failure-inducing atomicity violations in PBZIP2 that can only be discovered by two and one input out of the 8 test inputs, respectively. It is difficult to predict the bug-detection capability when only a couple of inputs are randomly selected.

Third, the CFP-guided approach is more informative. Random input selection does not tell developers how thorough the bug detection is and when the bug detection can stop. As a coverage metric, CFP provides a quantitative measurement to developers just like how traditional statement/branch coverage metrics help developers test sequential software. Developers can also combine CFP with other information, such as which part of the program is more prone to bugs, to further enhance the testing quality.

Fourth, the only advantage of random input selection over our CFP-guided approach is that it takes less time to select inputs. However, since the time spent running concurrency-bug detectors is huge (Step 3 in Table 7), much longer than the time spent measuring CFP and selecting inputs/functions based on CFP (Step 1 and Step 2 in Table 7), this small performance advantage is negligible in the big picture.

We should also note that traditional coverage metrics designed for sequential software, such as counting how many functions/branches/statements are executed, are not effective at guiding concurrency-bug detection. As a simple example, in FFT and LU, we can design two inputs that have exactly the same command-line options, except that one executes the program in single-threaded mode and the other in multi-threaded mode. These two inputs would cover exactly the same set of functions. However, they clearly have different capabilities in exposing concurrency bugs.

What if there are more test inputs? In practice, test-input sets used during in-house testing are much larger than the ones available to us and evaluated by us in this paper. We believe the benefit of our CFP approach will not diminish and can likely get more significant for larger input sets.

Most importantly, the phenomenon of inputs sharing common pairs of concurrent functions and exposing the same concurrency bugs widely exists in reality. During software testing, test input sets are usually designed to achieved good control-flow coverage and/or good data-flow coverage. In order to cover a previously uncovered statement, function, or define-use pair, a test input usually needs to execute a lot of statements or functions already covered by other inputs. As a result, there is a natural code-coverage overlap across test inputs. Consequently, CFP-coverage overlap and bug-report duplication naturally exist among test inputs. Since the time used to measure CFP is only a small percentage of that used to detect bugs, we believe the CFP approach will maintain its advantage over traditional bug detection for large test-input sets in reality. In fact, for a given program, as the input set grows larger, more bug reports are likely to be duplicates and more CFP are likely to overlap. Thus, the performance advantage of our CFP approach would likely increase.

Does CFP work for other detectors? As a general metric characterizing the interleaving space, we believe CFP can help many concurrency-bug detectors to work on a set of inputs. Of course, the benefit would decrease, if a concurrency-bug detector runs much faster than the detectors used in this paper (e.g., more than 10 times faster). However, techniques used to speed up concurrency-bug detection [63, 71] can po-

tentially also help speed up CFP measurement, in which case the benefit of CFP-guided bug detection will remain.

Does CFP work for all applications? Different applications may benefit differently from the CFP approach. For example, some I/O-intensive applications have relatively small overhead in run-time concurrency-bug detection, and hence will benefit less from CFP-guided bug detection.

As discussed in Section 5.5, most false negatives of our CFP approach occur in functions that are long and have complicated control flows. Applications with more functions of this type may suffer more false negatives. At the same time, the benefit of the CFP approach is unlikely to decrease when the application gets larger, as long as the ratio of long functions does not increase. Considering that long functions can hurt software modularity and maintainability, we expect most applications, no matter small ones or large ones, to benefit from the CFP approach with only small numbers of false negatives.

How to further improve CFP? What is presented above is just a starting point to improving multi-input concurrency-bug detection. There is still room for improvement. In terms of performance, future work can explore more efficient CFP-measurement techniques with help from static analysis. In terms of functionality, a function may not be the best unit for interleaving-space characterization. Sometimes, a function may be too small as a unit: monitoring the entrances and exits of utility functions that only have a couple of global or heap memory accesses leads to a huge overhead. Sometimes, a function may be too big as a unit. For example, synchronization operations inside a function can cause different parts of a function to have different logical timestamps; some large functions may include different paths accessing completely different global/heap variables. As described in Section 5.5, many false negatives in our current implementation occur within these big functions. Future work can explore how to extend CFP to better guide cross-input bug detection.

6. Related Work

Many tools are designed to detect data races [7, 15, 42, 49, 50, 69], atomicity violations [6, 14, 31, 32, 61, 65], and other types of concurrency bugs [25, 33, 59, 67, 70]. Since concurrency-bug detection often involves monitoring many memory accesses across threads and complicated concurrency analysis, most of these tools incur large overhead.

Sampling [2, 13, 24, 35], hardware support [21, 46, 47, 57, 63], and other optimization techniques [29] have been proposed to improve the performance of each concurrency-bug detection run. This paper has a different perspective and can well complement the above techniques. Specifically, all the previous works are oblivious to the selection of inputs. This paper prioritizes test inputs by their potential to cover the most unexplored concurrent function pairs, so that bug detection across a set of inputs becomes more efficient. In addition, many existing performance-enhancing techniques focus on data-race detection. Our CFP-based approach can also help other types of concurrency bugs.

The work mentioned above all focuses on dynamic bug detection. The problem of how to efficiently detect bugs for a set of inputs applies to only dynamic tools, but not static tools [12, 39] — static tools do not take inputs into account. Of course, static tools encounter their own challenges in scalability and accuracy, especially for large C/C++ programs, which is partly why so much work has focused on dynamic techniques. We believe static analysis, such as may-happen-in-parallel analysis [1], can help further improve our CFP-guided bug detection in the future.

Different metrics have been proposed to measure the coverage of interleaving testing [10, 22, 26, 30, 58, 60, 66]. Apart from the traditional coverage-based adequacy criteria, saturation-based adequacy criteria [58] are also proposed to help apply these interleaving-coverage metrics into testing for multi-threaded software. Our CFP metric complements traditional interleaving-coverage metrics and adequacy criteria by focusing on a different target: to select bug-detection inputs from a given input set. Due to this different target, our design also faces different challenges. For example, the measurement of CFP coverage has to be efficient, and the CFP metric has to strongly correlate with the follow-up concurrency-bug detector(s). We did not directly reuse traditional coverage metrics, because they tend to be too expensive to measure *before* bug detection or not strongly correlated with data races or atomicity violations. Of course, the traditional adequacy criteria, especially the saturation-based adequacy criteria, can help our CFP approach to judge whether more test inputs are needed.

Many techniques are proposed [3, 10, 37, 44, 53] to effectively explore the interleaving space of each input. Different from these techniques, this paper tries to coordinate bug detection across inputs. It can help identify testing candidates more efficiently for these interleaving-testing techniques on a set of inputs.

A recent position paper [9] by the authors presents the preliminary version of this work. That position paper focuses on understanding the interleaving-space overlap across inputs and across different versions of a software project. A preliminary idea of CFP-guided race detection was proposed there. However, the algorithm to measure CFP in that paper was not accurate enough and may miss many concurrent function pairs. As no optimizations are proposed, the CFP measurement in that paper was also slow, causing as much as 30X slowdown for Mozilla. In addition, cross-input atomicity-violation detection was not discussed.

Symbolic execution has been used for testing sequential software [5, 19, 55] and unit testing multi-threaded software [54]. Model checking for multi-threaded software has been well studied [11, 18, 23, 48, 62]. The observation that interleavings overlap across inputs is not new in model check-

ing and partial-order reduction is often used to avoid repeatedly exploring the same state [18]. Unfortunately, this observation has never been studied in the context of dynamic concurrency-bug detection and related testing. Due to the different goals and approaches in these two fields, new approaches are needed to exploit interleaving-space overlap.

7. Conclusions

This paper proposes improving the quality of concurrency-bug detection and multi-threaded software testing by avoiding redundant analysis across inputs. Our study of open-source applications shows that a significant number of races and single-variable atomicity violations overlap across inputs. Based on this study, we propose a new metric, concurrent function pairs (CFP), to guide multi-input concurrency-bug detection. Our evaluation using 5 open-source applications shows that CFP-guided concurrency-bug detection can effectively reduce redundant bug detection and improve the overall bug-detection efficacy.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback which has substantially improved the content and presentation of this paper. We thank Borui Wang and Peisen Zhao for their assistance in building an early protocol of this work. This work is supported in part by NSF grants CCF-1018180, CCF-1054616, and CCF-1217582; and a Clare Boothe Luce faculty fellowship. Last but not the least, one of our authors, Shan Lu, wants to thank her daughter, Silu, for waiting patiently for seven days beyond her due date. Coming to the world three hours before the OOPSLA submission deadline, Silu gave her mom enough time to finish writing this paper.

References

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPoPP*, 2007.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[3] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[4] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[6] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE*, 2008.

[7] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[8] Click. The Click Modular Router Projec. http://read.cs.ucla.edu/click/click.

[9] D. Deng, W. Zhang, B. Wang, P. Zhao, and S. Lu. Understanding the interleaving-space overlap across inputs and software versions. In *4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[10] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 2002.

[11] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL*, 2011.

[12] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[13] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[14] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[15] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[16] J. Gilchrist. Parallel BZIP2, Data Compression Software. http://compression.ca/pbzip2/.

[17] GNU. gcov. http://www.linuxcommand.org/man_pages/gcov1.html.

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

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[20] P. Godefroid and N. Nagappan. Concurrency at Microsoft an exploratory survey. Technical report, Microsoft Research, MSR-TR-2008-75, May 2008.

[21] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. M. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[22] M. J. Harrold and B. A. Malloy. Data flow testing of parallelized code. In *Proceedings of the International Conference on Software Maintenance*, 1992.

[23] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[24] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[25] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[26] P. V. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent software. In *ISSTA*, 1996.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[28] N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. In *IEEE Computer*, 1993.

[29] D. Li, W. Srisa-an, and M. B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *OOPSLA*, 2011.

[30] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *FSE*, 2007.

[31] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[32] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[33] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[35] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[36] Mozilla. SpiderMonkey, Mozilla's JavaScript engine. https://developer.mozilla.org/en/SpiderMonkey.

[37] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[38] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, 2012.

[39] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[40] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI*, 2007.

[41] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[42] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[43] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[44] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[45] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[46] M. Prvulovic. Cord:cost-effective (and nearly overhead-free) order-reordering and data race detection. In *HPCA*, 2006.

[47] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, 2003.

[48] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[49] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *RV*, 2010.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[51] SDTimes. Testers spend too much time testing. http://www.sdtimes.com/SearchResult/31134.

[52] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[53] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[54] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FSE*, 2006.

[55] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, 2005.

[56] K. Serebryany and T. Iskhodzhanov. Thread-sanitizer, a valgrind-based detector of data races. http://code.google.com/p/data-race-test/wiki/ThreadSanitizer.

[57] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *ICSE*, 2011.

[58] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *FSE*, 2009.

[59] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[60] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 1992.

[61] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[62] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Soft. Eng. Journal*, 2003.

[63] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS*, 2010.

[64] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[65] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[66] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, 1998.

[67] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[68] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.

[69] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[70] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[71] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.