Understanding the Interleaving-Space Overlap across Inputs and Software Versions

Dongdong Deng Wei Zhang Borui Wang Peisen Zhao Shan Lu
University of Wisconsin, Madison
dongdong@cs.wisc.edu, wzh@cs.wisc.edu, bwang29@wisc.edu, pzhao5@wisc.edu, shanlu@cs.wisc.edu

Abstract

In the multi-core era, it is critical to effectively test multi-threaded software and expose concurrency bugs before software release. Previous work has made a lot of progress in exercising the interleaving space and detecting concurrency bugs under a given input. Unfortunately, since software often has many test inputs and constant pressure to release new versions, existing techniques are still too expensive in practice. In this position paper, we use open-source software to study how interleavings, data races and atomicity violations particularly, overlap across test inputs and software versions. We also conduct preliminary explorations to improve the testing efficiency of multi-threaded software by avoiding redundant analysis across inputs and software versions.

1. Introduction

The rise of the multi-core era dictates the prevalence of multi-threaded software. Unfortunately, concurrency bugs widely exist in multi-threaded software [22] and have caused severe damages in the real world [51]. Therefore, effective software 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 share-memory access (i.e., interleavings). Facing the huge input space, interleaving space, and the pressure of releasing new versions of software, existing in-house testing allows many concurrency bugs to escape to production runs.

In today's practice, concurrency-bug detection and testing usually involve two steps *for each version of software*: first, testers design a set of inputs to provide code coverage; second, *for each test input*, the program is executed multiple times to exercise different interleavings. A lot of research is done to improve the second step by focusing on bug-prone interleaving patterns, such as races [52], atomicity violations [44], and others [40]. Unfortunately, even with state-of-the-art techniques, the second phase still introduces 10X – 100X slowdown, not affordable for a large set of test inputs and an application with constant pressure to release new versions.

This position paper proposes to improve testing efficiency by exploiting the interleaving-space overlap across test inputs and software versions. To support this proposal, Section 2 will study how interleavings, races and atomicity violations in particular, overlap across inputs and software versions. Section 3 will discuss how to improve bug detection across inputs and software versions by leveraging this overlap and avoiding redundant analysis.

2. Understand the interleaving-space overlap

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 they use pthread library as the underlying concurrency framework. None of them requires a JIT compiler.

App.	Description	# inputs
Aget 0.4.1	A parallel-downloading application [1]	8
Click 1.8.0	A software router [9]	6
FFT	A scientific computing benchmark [61]	8
Mozilla-js m10	A JavaScript engine [39]	8
PBZIP2 0.9.4	A parallel-compression application [18]	8

Table 1. Applications and test inputs in study

Test inputs Click and Mozilla-js both have test-input sets designed by their developers. For Click, our study uses all its test inputs that do not require OS-kernel changes. For Mozilla-js, we randomly mix their test inputs into 8 groups of multi-threaded JavaScript requests. For the other three applications, we design 8 inputs for each to cover different command line options and corresponding functionalities. Our Aget inputs cover different Aget functionalities, such as normal file downloading, download suspending, download resuming, and others; our FFT inputs exercise different FFT-computation settings and different functionalities, such as normal FFT, inverse FFT, printing per-thread statistics, and others; our PBZIP2 inputs exercise different options, including compression, decompression, error-message suppressing, compression-integrity testing, and others.

Interleaving counting Our study focuses on two most common types of interleaving patterns: data races [42, 50] and single-variable atomicity violations [15, 33, 44, 58]. We use a tool implemented in PIN [36] to collect per-thread execution traces of global/heap memory accesses and synchronization operations. We then analyze traces to detect

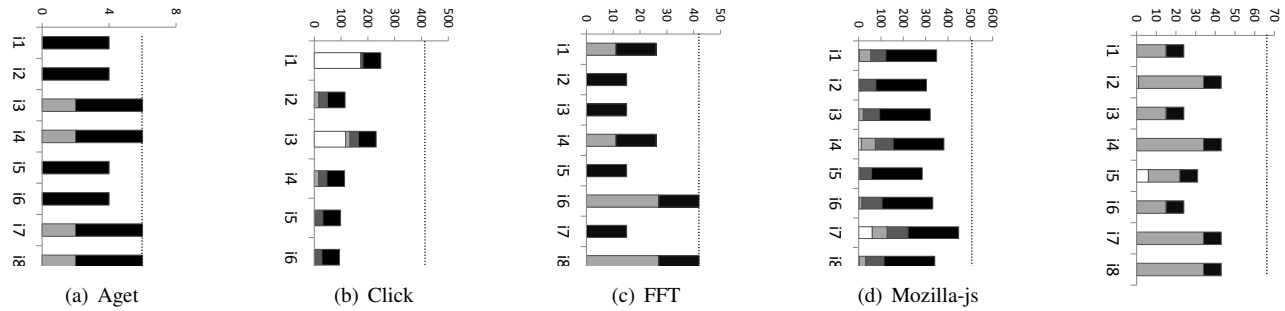


Figure 1. Data-race overlaps across inputs (The Y-Axis shows the number of data races. Each bar represents races detected by one test input. The dotted line in each sub-figure marks the total number of unique races reported by all inputs. Colorless strips represent races exposed by 1 input; light-gray strips represent races exposed by 2–4 inputs or 2–3 inputs in Click; dark-gray strips represent races exposed by 5–7 inputs or 4–5 inputs in Click; black strips represent races exposed by all inputs.)

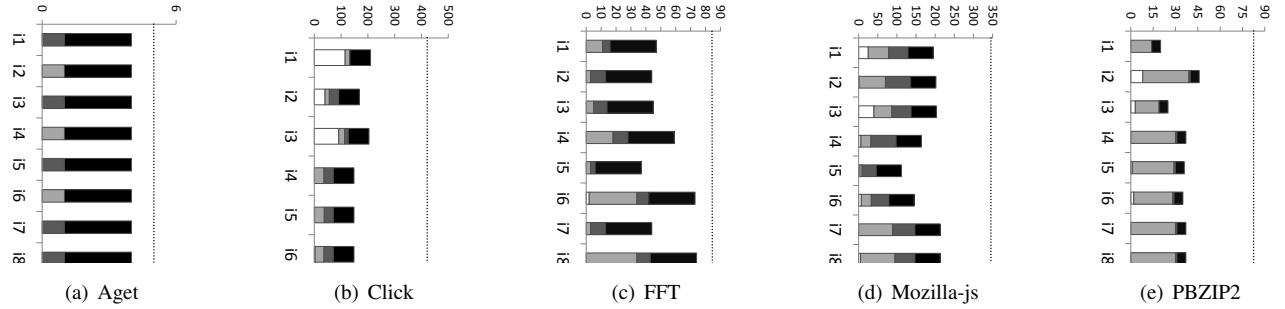


Figure 2. Atomicity-violation overlaps across inputs (The Y-Axis, X-Axis, bars, strips, and the dotted lines have the same meanings as those in Figure 1.)

data races and potential atomicity violations. Our race detection uses a lock-set/happens-before hybrid algorithm, similar to those in many open-source race detectors [41, 56]. Our atomicity-violation detection follows an algorithm described in CTrigger [44]. Both detectors have been implemented and used in our previous work [44, 65, 66].

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

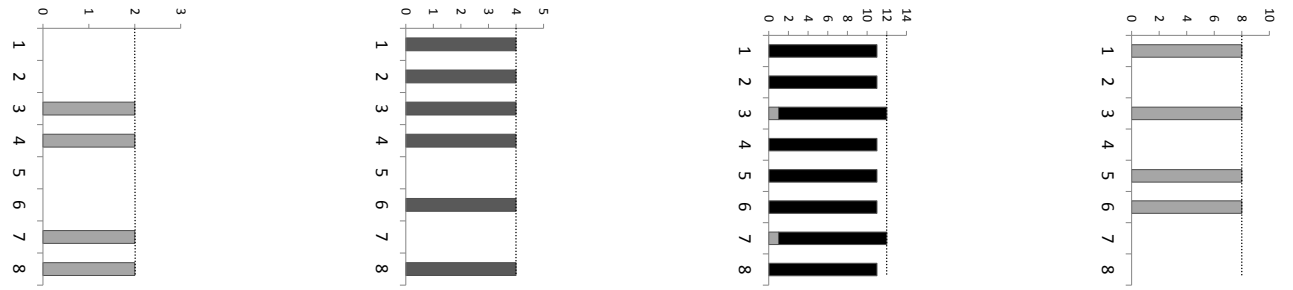
Our detectors recognize thread synchronization calls such like pthread_mutex_(un)lock, pthread_create , pthread_join , and the barrier macro in SPLASH2 [61]. Like almost all other detectors, our detectors do not recognize custom synchronizations and can have false positives and negatives.

2.2 Interleaving spaces across inputs

Figure 1 shows how races detected by different inputs overlap. As demonstrated by the colorless strips in Figure 1, data races exposed by only one input out of the test-input set are not common in most applications. They contribute to fewer than 18% of unique data races in Aget, FFT, Mozilla, and PBZIP2. In the same time, there are 14% – 67% of unique data races in these five applications that are exposed by *all* test inputs, as shown by the black strips in the figure. There are also many races covered by some, yet not all, test inputs, as demonstrated by gray strips in Figure 1.

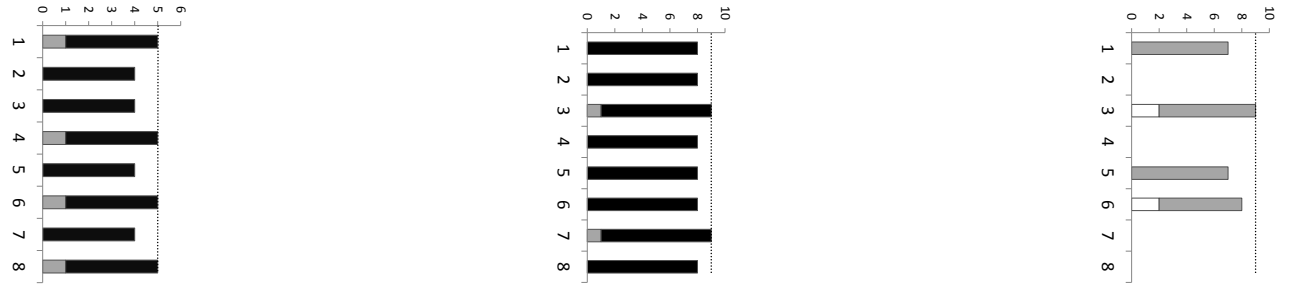
Figure 2 illustrates single-variable atomicity violations detected by different inputs. The trend here is similar to that of data races. All applications, except Click, have 0–30% of unique atomicity violations that are exposed by only one input (white strips in Figure 2). Meanwhile, all applications, except PBZIP2, have more than 20% of unique atomicity violations that are exposed by *all* test inputs (black strips in Figure 2).

Theoretically, a race or an atomicity violation may lead to failures under one input and maintain benign under another input. Therefore, we further investigate which races



(a) Aget (b) FFT (c) Mozilla-js (d) PBZIP2

Figure 3. Failure-inducing data-race overlaps across inputs. We use the same graphical representation as in Figure 1.



(a) FFT (b) Mozilla-js (c) PBZIP2

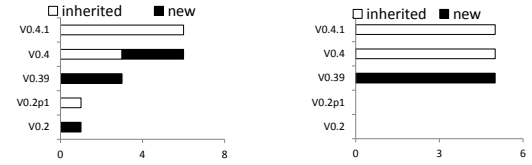
Figure 4. Failure-inducing atomicity-violation overlaps across inputs. We use the same graphical representation as in Figure2.

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. Furthermore, failure-inducing races/atomicity-violations are all exposed by more than one input, as shown in Figure 1 and Figure 2.

Similar to Figure 1 and Figure 2, Figure 3 and Figure 4 illustrate how failure-inducing races and atomicity violations overlap across inputs. As can be seen from these two figures, the same failure-inducing race or atomicity violation can be repeatedly detected in many different inputs, similar to the non-failure-inducing ones.

2.3 Interleaving spaces across software versions

This study takes a close look at open-source software Aget, a utility program designed for parallel downloading [1]. Aget is a widely used benchmark for concurrent software reliability research [34][65][10], because it contained some rather interesting concurrency bugs. Since its initial release in 2002, Aget has been updated for 4 times, with its latest update in 2009. To emulate the history of concurrent bug detection and study how the (buggy) interleaving spaces evolve through different versions, we applied race detection and atomicity-violation detection to all five versions of Aget, with the result illustrated in Figure 5.



(a) Races (b) Atomicity violations

Figure 5. How interleavings evolve along software versions

As shown in Figure 5(a) and Figure 5(b), among the four updates in Aget, only one update has introduced new atomicity violations and only two updates have introduced new races. The biggest interleaving-space change came at the second update. This update (v0.39) doubled the code size of Aget and introduced a significant amount of code refactoring and new functionality, such as download suspension and download resuming. It also brought 3 new races and 5 new atomicity violations among newly introduced code.

There have been two failure-inducing races in Aget[1]. They are both introduced by the third update (version 0.4). Each race is between two instructions that access a newly introduced shared variable within old code regions.

[1] Sequential consistency memory model is assumed here.

Summary Interleaving patterns, such as races and atomicity violations, overlap significantly across inputs and software versions. Interleaving testing and concurrency-bug detection would waste a lot of effort, if they are not coordinated across inputs and software versions.

3. Exploit the interleaving-space overlap

App.	# Race-Testing Runs		# Atom-Testing Runs	
	Base*	Strawman#	Base	Strawman
Aget	80	12	32	5
Click	1798	820	1026	420
FFT	392	84	423	83
Mozilla	5524	1010	1486	345
PBZIP2	550	132	273	82

Table 2. Removing unnecessary testing by the strawman (*: twice the number of all race reports; #: twice the number of unique races)

3.1 A strawman approach

A strawman approach to improving race-guided testing [52] and atomicity-guided testing [44] is simply recording what race orders and atomicity violations have been tested, and then avoiding the same race orders or atomicity violations during the testing runs of different inputs or different code versions. If we consider baselines as interleaving-testing frameworks [44] that use one program run to test one race-order/atomicity-violation under each test input, the strawman approach can save a significant number of interleaving-testing runs. Actually, it can reduce 54–85% of testing runs for applications described in Section 2.1 (Table 3), and save up to 100% of runs across multiple versions of Aget.

3.2 A better approach for cross-input bug detection

The limitation of the strawman approach is that it relies on detecting data races and atomicity violations under each input, which often incurs 10X – 100X [49] slowdown and is unaffordable for large sets of inputs in practice. In the following, we explore how to coordinate race detection across inputs.

3.2.1 Design

We aim to coordinate race detection and avoid reporting data races already reported by previous inputs. The challenge is to *predict* which part of memory-access monitoring and synchronization analysis will end up with redundant race reports *before* any heavy-weight monitoring and analysis.

To address this challenge, we propose coordinating race detection with a metric that characterizes interleaving spaces at a coarse granularity and hence is light-weight to measure. Our preliminary exploration designs such a metric: *Concurrent Function Pairs* (CFP). We define CFP as the set of all function pairs that can execute in parallel with each other.

Our CFP-guided race detection includes three steps for a set of inputs. The first step executes every input and obtains the CFP of each input. The second step selects the smallest set of inputs that can cover all unique concurrent function pairs. The output of this step includes a list of selected inputs and one list of selected functions for each selected input. The third step applies a race detector to selected functions under each selected input.

3.2.2 Implementation

For the first step of CFP-guided race detection, our preliminary implementation calculates the CFP of an input by analyzing the run-time log of the entrance and the exit of every function call. We implemented an LLVM-based tool to conduct the logging for every input. Our trace analysis considers an invocation of function f1 to be concurrent with an invocation of function f2 from a different thread, if and only if the entrance, or exit, of f1 and the entrance, or exit, of f2 have concurrent logical timestamps[2] and are not protected by the same lock. Experiment results show that the reported CFP are very stable across testing runs. Even in the extreme cases like Click where thousands of concurrent function pairs are reported each time, we don't see more than 1% fluctuation between each run. Since the fluctuation is small, we only run each program **once** under each input to collect the CFP.

Given the list of concurrent function pairs of each input, selecting the smallest set of inputs to cover all pairs is actually an NP-hard problem. We tried two implementations: one based on branch-and-bound algorithm [29] that provides optimal results and one that provides approximated results. We ended up with the latter, because the former is too expensive in our experiments. Our approximated 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 are covered.

For the third step, we slightly modified the race detector described in Section 2.1 to monitor only specified functions.

3.2.3 Preliminary evaluation

App.	Speedup (X)	Race False Negative(%)	Bad-Race False Neg.(%)	Trace Reduction (%)
Aget	0.76*	0%	0%	94%
Click	2.14	0.2%	N/A	64%
FFT	7.13	0%	0%	83%
Mozilla	1.27	4%	0%	33%
PBZIP2	1.72	3%	0%	44%

Table 3. Overall results of CFP-guided race detection, with the traditional *full* race detection as the baseline (*: a slowdown due to the I/O intensive nature of Aget).

Our preliminary experiments compare two approaches of race detection. One is the traditional approach that applies race detection to each and every test input. It uses the race detector described in Section 2.1 and will be referred to as *full* detection. The other is our CFP-guided approach.

[2] We compute logical timestamps based on synchronization operations such as barriers, thread creations, and thread joins.

Overall results As shown in Table 3, through cross-input coordination, our CFP approach effectively reduces trace sizes (33 – 94% of reduction) and improves the performance of all but one benchmark (up to 7X speedup), with only 0 – 4% false-negative rates among all races and **no** false negative among failure-inducing races. These results demonstrate the potential of coordinating bug detection across inputs.

Redundancy avoidance Our CFP-guided approach tries to avoid reporting one data race under multiple inputs. Table 4 shows how this goal is achieved by selecting inputs and functions in race detection. As shown in the table, 1 – 7 out of 8 inputs are selected for Aget, FFT, Mozilla, and PBZIP2. All 6 test inputs are selected for Click race detection. Fortunately, only a few functions need to be monitored for most of these inputs, which significantly reduces the trace size from 127 MB to 46 MB.

By selecting inputs and functions, our CFP approach has successfully decreased the average number of inputs under which each race is reported from 2.19 – 6.67 to 1.0 – 2.73 (the Overlap columns of Table 4). The reason that we failed to decrease the Overlap to 1 can be explained by an example. Suppose we choose input i1 to cover a concurrent function pair {f2, f3}, and i2 to cover pairs {f1,f3} and {f1,f2}. Under this selection scheme, a race between instructions in f2 and f3 could be reported by both i1 and i2, because f2 and f3 are monitored in both inputs. Future work can design better monitoring schemes or input/function selection schemes to further decrease the Overlap.

App.	Overlap		#Used Inputs		Trace Size (MB)	
	Full	CFP	Full	CFP	Full	CFP
Aget	6.67	1.00	8	3	11.4	0.63
Click	2.19	1.64	6	6	127	45.5
FFT	4.67	1.00	8	1	34.2	5.98
Mozilla	5.47	2.73	8	7	12.4	8.35
PBZIP2	4.17	1.16	8	2	402	227

Table 4. Input/function selection and overlap reduction (Overlap = $\frac{\text{\# all race reports}}{\text{\# unique race reports}}$)

False negatives Our CFP-guided detection has missed very few races reported by full race detection, with 0 – 4% false-negative rates. Furthermore, there is no false negative among failure-inducing races. Almost all false negatives occur when different inputs cover different basic blocks of a function and the input selected by us happens to miss the race-containing blocks.

Performance The speedup of CFP is mainly determined by the input/function selection. Intuitively, the fewer inputs/functions selected, the faster is the CFP-guided race detection. For example, the CFP approach is 7 times as fast as the baseline full race detection for FFT, because only 1 out of 8 test inputs is selected for race detection. On the other hand, only 1.27X speedup is achieved in Mozilla, because 7 out of 8 test inputs and 67% of full-detection traces remain in CFP-guided detection. The detailed performance breakdowns of all benchmarks are shown in Table 5.

Among all benchmarks, Aget is a special case. Although the CFP approach effectively selects inputs and decreases the race-report overlap from 6.67 to 1, it is actually slower than full race detection. The reason is that the current CFP approach requires one run of each input to calculate CFP. In most applications, this CFP-calculation time is compensated by the reduction of race-detection time. Unfortunately, this does not apply for Aget, whose I/O-intensive nature gives it a low full race-detection overhead, 11%, to compete with. This shows that the current CFP approach may not help some I/O intensive applications that have low race-detection overhead. Of course, since CFP incurs negligible overhead to collect in these applications, collecting them are still worthwhile for software with many test inputs and for coordinating other types of concurrency-bug detection.

App.	Full	CFP			
		Step1	Step2	Step3	Total
Aget	1.11	1.05	0.000035	0.40	1.45
Click	313	1.87	0.49	144	146.36
FFT	211	1.85	0.09	27.7	29.64
Mozilla	260	29.5	5.07	175	209.57
PBZIP2	4.38	1.02	0.0016	1.55	2.57

Table 5. The total testing time across all inputs (For each application, the total execution time of all test inputs w/o any instrumentation is considered as 1)

The Step 1 and 2 in Table 5 show the time spent collecting function-entrance/exit traces of all inputs, calculate the CFP, and select inputs accordingly. In general, they incur relatively small overhead. Mozilla is the outlier here, because it has many utility functions with zero or just a few share-memory accesses. These functions lead to huge overhead in CFP monitoring, CFP calculation, and input selection. We expect the performance of Step 1 and 2 to significantly improve by pruning local-access functions, merging utility functions with their callers, and parallelizing our single-threaded trace-analysis and input-selection programs. The Step 3 in Table 5 corresponds to the overhead for race detection on the inputs and functions selected in Step 2. Finally, the Column "Total" represents the end-to-end time using CFP approach, which sums up the time from Step 1 to Step 3. Evidently, CFP is a significantly cheaper approach than the traditional approach, this speedup was also given in the second column of Table 3.

3.2.4 Discussion

What presented above is just a starting point to cross-input concurrency-bug detection. We believe there is a lot of room for improvement. First, not all functions and function pairs matter. In our current implementation, many functions that access no global/heap variable are involved in CFP monitor-

ing and analysis. Getting rid of these functions can bring significant performance improvement. Static analysis can further help remove pairs of functions that cannot access the same variable from CFP calculation, and further improve the quality of our cross-input coordination.

Second, a function may not be the best unit for interleaving-space depiction. Sometimes, a function may be too small as a monitoring/analysis unit. In fact, monitoring the entrances and exits of utility functions that only have a couple of global/heap memory accesses leads to 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, making CFP measurement inaccurate. Some large functions may include different paths accessing completely different global/heap variables. Many false negatives in our current implementation occur within these big functions.

Finally, new metrics similar to CFP are needed to coordinate the detection of other types of concurrency bugs.

3.3 Opportunities for across-version bug detection

We could use CFP to guide the race detection across software versions. Specifically, we can first calculate the CFP covered by test inputs in the new version, and then focus race detection on function pairs not covered by old versions.

We could also use the CFP metric in a smarter way. In Aget, the failure-inducing races are introduced when developers insert statements that access the same variable into function http_get and save_log. These two functions are a pair of concurrent functions in older versions of Aget. Since no synchronization is changed/inserted in the new version, we can tell that the inserted statements race with each other without any traditional race analysis! We leave further exploration along this direction to future work.

4. Related Work

A lot of tools are designed to detect data races [8, 16, 42, 50, 64], atomicity violations [7, 15, 32, 33, 58, 62], and other types of concurrency bugs [17, 28, 30, 35, 57, 63, 65, 66]. Techniques, such as sampling [4, 14, 27, 37, 38] and hardware support [45, 46, 60], have been proposed to improve the performance of each concurrency-bug detection run. This paper has a different perspective with the above sampling methods. Specifically, all the previous works are oblivious to the selection of input. This paper prioritizes testing inputs by their potential to cover the most unexplored concurrent function pairs.

This paper is also orthogonal to some of the performance-enhancing techniques. Recent work such as DataCollider[14] has significantly reduced the data race detector overhead by leveraging hardware watchpoints. Our approach can help further reduce the overhead. For example, DataCollider could leverage information about which program locations are likely to participate in a previously unknown race through some prior analysis provided by this paper. Moreover, based on our experience, cross-input and cross-version overlaps also apply to other types of concurrency bugs.

Many techniques are proposed [5, 12, 40, 44, 53] to effectively explore the interleaving space of each input. Different from these techniques, this paper tries to coordinate testing across inputs and software versions.

Deterministic systems [2, 3, 31, 43] force software to deterministically follow one interleaving under an input. Since the schedule can be affected by subtle issues in these systems, such as the number of (write) instructions [11, 43] or synchronization operations [31] executed by a thread, problems like how to co-design input testing and interleaving testing and how to estimate the interleaving impact of a code change could become more interesting.

Symbolic execution has been used for testing sequential software [6, 21, 55] and unit testing multi-threaded software [54]. Model checking for multi-threaded software has been well studied [13, 19, 20, 23–25, 47, 48, 59]. The observation that interleavings overlap across inputs is not new in model checking and partial-order reduction is often used to avoid repeatedly exploring the same state. 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.

Recently, people start to pay attention to regression testing for concurrent software. CAPP[26] describes extensive heuristics for regression testing on concurrent JAVA software and their effectiveness. We studied the interleaving space overlap across different versions of an open-source application written in C/C++. Our study provides more motivation for regression testing in multi-threaded software, and points out heuristics that can be applied to concurrency-bug detection in an evolving software.

5. Conclusions

This position paper proposes a new direction to improve the quality of concurrency-bug detection and multi-threaded software testing by avoiding redundant analysis across inputs and software versions. Our study of open-source applications shows that a significant number of races and single-variable atomicity violations overlap across inputs and software versions. Our preliminary exploration also shows the potential of leveraging these overlaps to improve the performance of bug-detection and testing. We expect future work to design better metrics to help coordinate cross-input bug detection and design better change-impact analysis to efficiently detect concurrency bugs brought by code changes.

6. Acknowledgments

We would like to thank anonymous reviewers for their invaluable feedback. Shan Lu is supported by a Claire Boothe Luce faculty fellowship, and her research group is supported by NSF grant CCF-1018180 and CCF-1054616.

References

[1] Aget. Multithreaded http download accelerator. www.enderunix.org/aget, 2011.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

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

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

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] F. Chen, T. F. Serbanuta, and G. Rosu. jpredictor: a predictive runtime analysis tool for java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, 2008.

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

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

[10] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

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

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

[14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

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

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

[17] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndstrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

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

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

[20] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.

[21] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

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

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

[24] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.

[25] C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[26] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 133–143, New York, NY, USA, 2011. ACM.

[27] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *Proceedingsof the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.

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

[29] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[30] T. Li, C. Ellis, A. Lebeck, and D. Sorin. On-demand and semantic-free dynamic deadlock detection with speculative execution. In *USENIX Annual Technical Conference*, 2005.

[31] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[32] 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 *the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

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

[34] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 553–563, New York, NY, USA, 2009. ACM.

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

[36] 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.

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

[38] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

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

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

[41] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

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

[43] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[44] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *The 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS09)*, March 2009.

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

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

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

[48] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[49] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 34–41, New York, NY, USA, 2006. ACM.

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

[52] K. Sen. Race directed random testing of concurrent programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 11–21, New York, NY, USA, 2008. ACM.

[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*, pages 263–272, 2005.

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

[57] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

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

[59] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[60] 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.

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

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

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

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

[65] 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.

[66] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.