# Which Compiler Optimization Options Should I Use for Detecting Data Races in Multithreaded Programs?†

Changjiang Jia
City University of Hong Kong
Tat Chee Avenue, Hong Kong
cjjia.cs@my.cityu.edu.hk

W.K. Chan††
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Different compiler optimization options may produce different versions of object code. To the best of our knowledge, existing studies on concurrency bug detection in the public literature have not reported the effects of different compiler optimization options on detection effectiveness. This paper reports a preliminary but the first study in the exploratory nature to investigate this aspect. The study examines the happened-before based predictive data race detection scenarios on four benchmarks from the PARSEC 3.0 suite compiled under six different GNU GCC optimization options. We observe from the data set that the same race detection technique may produce different sets of races or different detection probabilities under different optimization scenarios. Based on the observations, we formulate two hypotheses for future investigations.**

***Index Terms*—Race detection, compiler optimization option, empirical study, hypothesis formulation**

## I. Introduction

Concurrency bugs [4][14] can be disastrous: A race condition causes the 2003 North American Blackout [17]. Two race conditions cause radiation overdoses of the Therac-25 radiation therapy accident in 1980s [13]. The classical priority inversion problem among concurrent tasks causes deadlock occurrences, leading to periodic resets of the Mars Rover Pathfinder when carrying out its mission on Mars in 1997[11].

There is a large body of work (e.g., [6][8][21]) to assist developers to test multithreaded programs with respect to concurrency bugs. They range from specific techniques to empirical studies. To the best of our knowledge, almost all such techniques assume that the *object code* or an *intermediate representation* of a multithreaded program is the subject under test. On the given object code, many techniques attempt to either passively monitor an execution trace [7] or actively guide the thread schedules through randomized scheduling [2][19]. Existing empirical studies on concurrency bugs in multithreaded programs largely focus on surveys (e.g., [14] [18]) or mining artifacts from the bug repositories of these programs, possibly with the source code of the programs, to identify interesting correlations, patterns, or their trends (e.g., [8]). Existing work provided invaluable results, but has ignored the fact that developers often write program in the source code format instead of in the object code (or intermediate representation) format.

Developers use compilers to translate their source code into the object code subject to testing. Such transformations made by a compiler are configurable. For instance, the GNU GCC compiler [9] provides a number of compiler optimization options [10]. Developers thus face the problem of selecting more favorable options to detect failures from their programs.

To the best of our knowledge, no existing work in the public literature has systematically reported the influence of such options for concurrency bug detection. In this paper, we present the *first* work in this area. Fig. 1 illustrates an instance of the problem setting presented in this paper.

In this paper, we report a preliminary study in the exploratory nature on four benchmarks of the PARSEC 3.0 suite [1][16] compiled with the GNU GCC compiler [9] using six different optimization options (see TABLE I). We examine the differences in the detection effectiveness on the same benchmark. We observe that detecting data races from the same benchmark under different optimization options may lead a detector to report different sets of races with different detection probabilities. Based on the observations, we formulate two follow-up hypotheses for further investigations.

The rest of the paper is organized as follows. Section II formulates the research questions to be investigated in the exploratory study. Section III describes the experimental setup of the exploratory study. Section IV presents the data analysis. Section V further discusses the implication from the finding. Section VI concludes the paper.

## II. Research Questions

The aim of this study is to generate hypotheses for future investigation based on the following two research questions.

**RQ1**: Does a precise race detector detect the same number of data races from the same source code but compiled under different optimization options?

**RQ2**: Does a precise race detector report similar detection probability among data races in common from the same source code but compiled under different optimization options?

† This work is supported in part by the Early Career Scheme of the Research Grants Council of Hong Kong (project number 123512).
†† contact author.



Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

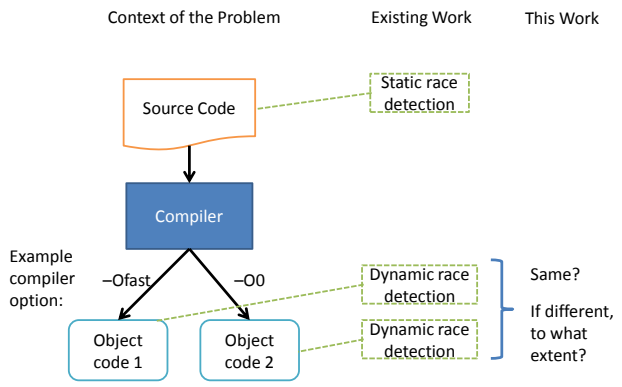


Fig. 1. Illustration of the problem context under investigation in this paper

## III. Exploratory Setup

Fig. 2 shows the theoretical framework of our study.

### A. Control Variables

To study the effects of compiler optimization options on detection effectiveness, a large-scale study is necessary. As the first step, we limit the scope of our study. A control variable is a setting unchanged throughout our experiment.

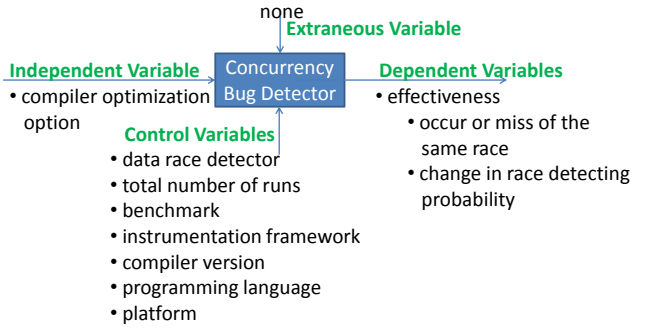


Fig. 2. A Framework for Investigating the Influences of Compiler Options on Concurrency Bug Detectors

The **platform** to conduct the experiment was a machine running the 32-bit Ubuntu desktop Linux 12.04.1 with 3.16GHz Duo2 processor and 3.8GB physical memory. The platform provided the system threads that map to multiple user threads in the program executions. Moreover, the platform provided a genuine thread control of the execution of each benchmark. This platform has been demonstrated to be able to repeat the third-party experimental results [7]. Using other platforms to conduct the same experimental procedure may produce different results.

We used C++ as the **programming language**.

The **compiler version** was GNU GCC 4.6.3 [9]. This series of compiler is likely a popular compiler used by many Linux installations. The selection of the compiler version was a major threat in our study because all the compiler optimization options that we examined their effects are limited by this selection. We had chosen the platform pre-installed version of GNU GCC compiler (i.e., gcc 4.6.3) at the time that we conducted the experiment.

The **instrumentation framework** used was PIN 2.11 [15]. It has been widely used by many research-based tools.

We used four programs (blackscholes, dedup, x264, and vips) in the PARSEC 3.0 suite [1][16] configured with eight working threads as our **benchmarks**. We used the simsmall input set [1]. An input set may contain multiple test cases.

The **total number of runs** was set to be 100.

We used LOFT, a happened-before [12] based tool, as the precise **data race detector**.

We leave on the generalization of the study by increasing the possible levels of each control variable as a future work.

### B. Independent Variable

We studied one independent variable: **compiler optimization option**. TABLE I shows the options examined in the exploratory study.

TABLE I. Compiler Optimization Options Used

| Option | Description |
|---|---|
| `-O0` | This is the default setting if no other `-O` option is inputted in the command line. It shortens the compilation time and makes the debugging to produce the expected results. |
| `-O1` | "Optimizing compilation takes somewhat more time, and a lot more memory for a large function." (It is the same as `-O`) |
| `-O2` | "GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code." |
| `-O3` | "`-O3` turns on all optimizations specified by `-O2` and also turns on the `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` and `-fipa-cp-clone` options. " |
| `-Ofast` | "Disregard strict standards compliance. `-Ofast` enables all `-O3` optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on `-ffast-math` and the Fortran-specific `-fno-protect-parens` and `-fstack-arrays`." |
| `-Os` | "Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size." |

Note: All quoted descriptions are taken from the following URL: http://gcc.gnu.org/onlinedocs /gcc/Optimize-Options.html#Optimize-Options

### C. Dependent Variables

The experiment had two dependent variables to study the differences in detection effectiveness as shown in Fig. 2.

*Metric* 1: We adopted to use the source code statement pair defined in [19] to represent the same race: If two races reported in different executions are triggered by the data access events associated with *the same pair of statements in the source code*, we treated these races as the *same race* in the data analysis.

*Metric* 2: We want to study the race detection probability. Specifically, if a race $r1$ can be detected in $x$ out of 100 rounds, then the race detection probability of $r1$ is said to be $x$%.

### D. Experimental Procedure

We compiled the source code of each benchmark under each compiler optimization option to generate one object code of the benchmark. We then ran this object code over the input suite of the benchmark with the precise data race detector, and recorded the values of each dependent variable. We ran the each program over the input set for 100 times.

### E. Threats to Validity

We studied the race detection effectiveness. Using more metrics and applying hypothesis test on the current data obtained can further consolidate the results of the study. The correctness of the tools used is a threat. We have assured them with small examples. We have not observed anomalies yet. Dynamic profiling may perturb program executions to make the comparison less accurate. We have not conducted statistical tests to confirm the differences observed so far to be statistically meaningful. There may also be confounding factors that we may not be aware of. The generalization with respect to each independent variable or control variable is necessary to make the result drawn from the exploratory study more accessible and reliable.

## IV. Data Analysis

For the same benchmark, compiling the same source code with different compiler optimization options will generate different object codes in terms of size and content. For the four benchmarks, TABLE II shows their source code size as well as the object code size under each of the six optimization options. For the source code size, we used the SLOCCount [20] to count the physical source lines of code for each benchmark. We used the Linux command *du* [5] to collect the size of the object code for each benchmark.

TABLE II. Sizes of Source Code and Object Code under Different Optimization Options of the Benchmarks

| Optimization Options | Benchmark | | | |
|---|---|---|---|---|
| | blackscholes | dedup | x264 | vips |
| | Size of Source Code (LOC) | | | |
| | 914 | 3347 | 41933 | 138536 |
| | Size of Object Code (KB) | | | |
| `-O0` | 37 | 242 | 2786 | 113140 |
| `-O1` | 41 | 353 | 3519 | 147157 |
| `-O2` | 41 | 377 | 3912 | 160314 |
| `-O3` | 41 | 406 | 3970 | 182035 |
| `-Ofast` | 41 | 410 | 3970 | 181507 |
| `-Os` | 41 | 271 | 2241 | 121779 |

### A. Answering *RQ1*: *Effectiveness* (*Number of Races*)

TABLE III shows the overall result on detection effectiveness. The number in each cell is the total number of data races detected in the entire experiment: On blackscholes, no options can detect any races. On dedup, `-Ofast` detects 13 out of the 20 races. On x264, every option can detect the same race. Lastly, on vips, all options except `-Os` detect all the 11 races, and `-Os` can only detects 8 races.

To answer RQ1, we find that not all optimization options may be equally effective for data race detection. We further formulate the hypothesis *H*1 with respect to dynamic precise data race detection for multithreaded programs:

> Hypothesis *H*1: Detecting data races from a program under different optimization options may result in reporting different sets of data races.

### B. Answering *RQ2*: *Effectiveness* (*Detection Probability*)

In this section, we analyze whether the race detection probabilities are consistent across different optimization options. We do not analyze blackscholes because no race was detected on it in the study.

TABLE IV, TABLE V, and TABLE VI show the detection probability of each race in dedup, x264, and vips, respectively. To ease our presentation, for the same benchmark, we assign each distinct race with a unique identity. The probability shown in each cell in these three tables is the total number of rounds that the same race has been detected in 100 rounds of executing the same object code over the entire input set.

TABLE III. Numbers of Detected Races under Different Optimization Options

| Benchmark | `-O0` | `-O1` | `-O2` | `-O3` | `-Ofast` | `-Os` | Total |
|---|---|---|---|---|---|---|---|
| blackscholes | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dedup | 8 | 10 | 7 | 7 | 13 | 8 | 20 |
| x264 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| vips | 11 | 11 | 11 | 11 | 11 | 8 | 11 |

TABLE IV. Probability of Race Detection on Dedup (in %)

| Race ID | `-O0` | `-O1` | `-O2` | `-O3` | `-Ofast` | `-Os` |
|---|---|---|---|---|---|---|
| 1 | 17 | 10 | 16 | 15 | 13 | 1 |
| 2 | 12 | 8 | 13 | 11 | 5 | 8 |
| 3 | 3 | 4 | 2 | 2 | 5 | 1 |
| 4 | 3 | 7 | 2 | 3 | 4 | 5 |
| 5 | 3 | 10 | 2 | 4 | 3 | 3 |
| 6 | 2 | 0 | 0 | 0 | 0 | 0 |
| 7 | 2 | 0 | 0 | 3 | 3 | 5 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 5 | 0 | 0 | 0 | 0 |
| 10 | 0 | 5 | 1 | 0 | 0 | 0 |
| 11 | 0 | 2 | 0 | 0 | 0 | 0 |
| 12 | 0 | 6 | 1 | 0 | 0 | 0 |
| 13 | 0 | 3 | 0 | 2 | 4 | 1 |
| 14 | 0 | 0 | 0 | 0 | 1 | 0 |
| 15 | 0 | 0 | 0 | 0 | 1 | 0 |
| 16 | 0 | 0 | 0 | 0 | 1 | 0 |
| 17 | 0 | 0 | 0 | 0 | 1 | 0 |
| 18 | 0 | 0 | 0 | 0 | 2 | 0 |
| 19 | 0 | 0 | 0 | 0 | 2 | 0 |
| 20 | 0 | 0 | 0 | 0 | 0 | 2 |

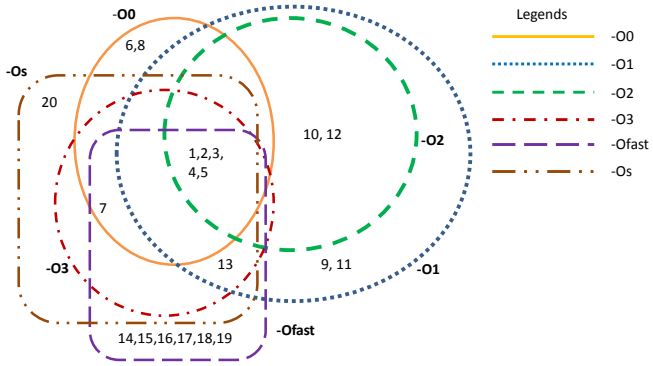


Fig. 3. The Venn diagram of race sets reported under different options

The Venn diagram of TABLE IV is shown in Fig 3. We observe that every option can detect the first 5 races (i.e., from race 1 to race 5). For the first 5 races in common, the six options show certain diverse detection probabilities. For race 1, the option `-Os` shows much lower detection probability than the other five options. However, the detection probabilities on the other four races are quite different. For the other 15 races (i.e., from races 6–20), none of the six options can individually

detect all of them, but we also observe that the race set reported under one option (e.g., `-O2`) may be a subset of the race set reported under another option. So to improve race detection effectiveness, we may select a selective combination of optimization options (`-O0`, `-O1`, `-Ofast`, and `-Os`) instead of using all these optimization options.

TABLE V. PROBABILITY OF RACE DETECTION ON X264 (IN %)

| Race ID | -O0 | -O1 | -O2 | -O3 | -Ofast | -Os |
|---|---|---|---|---|---|---|
| 1 | 100 | 100 | 100 | 100 | 100 | 100 |

TABLE VI. PROBABILITY OF RACE DETECTION ON VIPS (IN %)

| Race ID | -O0 | -O1 | -O2 | -O3 | -Ofast | -Os |
|---|---|---|---|---|---|---|
| 1 | 100 | 100 | 100 | 100 | 100 | 100 |
| 2 | 100 | 100 | 100 | 100 | 100 | 100 |
| 3 | 100 | 100 | 100 | 100 | 100 | 100 |
| 4 | 100 | 100 | 100 | 100 | 100 | 100 |
| 5 | 100 | 100 | 100 | 100 | 100 | 100 |
| 6 | 100 | 100 | 100 | 100 | 100 | 0 |
| 7 | 100 | 100 | 100 | 100 | 100 | 0 |
| 8 | 100 | 100 | 100 | 100 | 100 | 0 |
| 9 | 100 | 100 | 100 | 100 | 100 | 100 |
| 10 | 100 | 100 | 100 | 100 | 100 | 100 |
| 11 | 100 | 100 | 100 | 100 | 100 | 100 |

TABLE V shows that all options detect the same race with the same probability on x264. TABLE VI shows that the race detection probabilities on vips for races 6, 7, and 8 are very high and consistent across all options except the option `-Os`.

To answer RQ2, we find that different optimization options may show dissimilar race detection probabilities. We further formulate Hypothesis *H2* with respect to dynamic precise race detection on multithreaded programs:

> Hypothesis *H*2: Some races occur with high probability only under some optimization options.

## V. DISCUSSION

We have formulated two hypotheses from the current exploratory study. If these hypotheses could be established, they carry significant practice values.

Hypothesis *H*1, if established, reminds developers to test their programs under different optimization options (instead of merely one, say under the option `-O3`) that are used in their shipped releases. Hypothesis *H*2, if established, informs developers that the same races may occur with higher probabilities under particular optimization options. Hence, the two hypotheses (*H1* and *H2*), if established, together inform people to test a program under an array of optimization options to maximize the total number of distinct races to be detected within the limited test resources (budgets).

There is one tangible research direction: formulate a fault-tolerant strategy over a selective combination of compiler optimization options. The best option based on the results of any empirical studies is unlikely to be always applicable to deal with the individual cases faced by individual developers. For the testing of multithreaded programs, developers may require executing the same program over the inputs multiple times to test against different concurrency scenarios. As such, compiling the same program under different optimization options and spreading the total amount of test budgets over these compiled versions (e.g., evenly and iteratively as what we did in [21]) may be more effective to detect data races in them.

## VI. CONCLUSION

This paper reports the first work, in the form of preliminary exploratory study, on four benchmarks to study whether the optimization options of a compiler may affect the detection effectiveness of precise happened-before race detections. We observe that under different optimization options, race detection may report different sets of races from the same benchmark, possibly with different detection probabilities. Developers may consider these characteristics to select options to prepare their programs for race detections. We have also formulated two hypotheses for investigations in the future.

## REFERENCES


[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of PACT '08,* pp. 72–81, 2008.

[2] Y. Cai and W. K. Chan, "MagicFuzzer: scalable deadlock detection for large-scale applications," in *Proceedings of ICSE '12*, pp. 606–616, 2012.

[3] Y. Cai and W.K. Chan, "Lock Trace Reduction for Multithreaded Programs," in *IEEE Transactions on Parallel and Distributed Systems*. DOI: http://dx.doi.org/10.1109/TPDS, 2013.13.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *Proceedings of SOSP*'01, pp. 73−88, 2001.

[5] Du Linux/Unix command. Available at http://linux.about.com/library/cmd/blcmdl1_du.htm.

[6] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, "IFRit: interference-free regions for dynamic data-race detection," in *Proceedings of OOPSLA'12*, pp. 467-484, 2012.

[7] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *Proceedings of PLDI '09*, pp. 121–133, 2009.

[8] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Proceedings of DSN'10*, pp. 221–230, 2010.

[9] GCC Compiler. Available at http://gcc.gnu.org/.

[10] GNU. Option Summary. GNU C++ Compiler. Available at http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

[11] M. Jones, "What Really Happened on Mars Pathfinder Rover," *RISKS Digest*, 19, 49 (Dec. 1997), 1997.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM* 21, 7, pp. 558–565, 1978.

[13] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, 26, 7, pp. 18–41, 1993.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of ASPLOS XIII*, pp. 329–339, 2008.

[15] 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 *Proceedings of PLDI'05*, pp. 191–200, 2005.

[16] Parsec 3.0. Available at http://parsec.cs.princeton.edu/download.htm.

[17] K. Poulsen. Software bug contributed to blackout. Available at http: //www.securityfocus.com/news/8016.

[18] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proceedings of ICSE '10*, pp. 485–494, 2010.

[19] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of PLDI '08,* pp. 11–21, 2008.

[20] SLOCCount. Available at http://www.dwheeler.com/sloccount/.

[21] K. Zhai, B.N. Xu, W.K. Chan, and T.H. Tse, "CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications," in *Proceedings of ISSTA'12*, pp. 221–231, 2012.