To appear in Proceedings of 13th International Conference on Quality Software (QSIC 2013)

A Study on the Efficiency Aspect of Data Race Detection: A Compiler Optimization Level Perspective

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

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

***Abstract*—Dynamically detecting data races in multithreaded programs incurs significant slowdown and memory overheads. Many existing techniques have been put forward to improve the performance slowdown through different dimensions such as sampling, detection precision, and data structures to track the happened-before relations among events in execution traces. Compiling the program source code with different compiler optimization options, such as reducing the object code size as the selected optimization objective, may produce different versions of the object code. Does optimizing the object code with a standard optimization option help improve the performance of the precise online race detection? To study this question and a family of related questions, this paper reports a pilot study based on four benchmarks from the PARSEC 3.0 suite compiled with six GCC compiler optimization options. We observe from the empirical data that in terms of performance slowdown, the standard optimization options behave comparably to the optimization options for speed and code size, but behave quite different from the baseline option. Moreover, in terms of memory cost, the standard optimization options incur similar memory costs as the baseline option and the option for speed, and consume less memory than the option for code size.**

Keywords-data race; race detection; compiler optimization option; empirical study; efficiency

I. INTRODUCTION

Multithreaded programs are more difficult to develop correctly than their sequential counterparts due to the non-deterministic interference among threads in program executions. Programmers may design various locking mechanisms such as consistently applying the same lock object to protect the same shared memory location.

On the one hand, adding more locks may compromise the degree of parallelism, and may introduce deadlocks. On the other hand, adding insufficient thread locking to protect shared memory locations may lead to harmful data races. Moreover, it is still impractical to implement a program that is free of bugs. Hence, although the locking mechanism designs behind a multithreaded program could be perfect, the actual implementations may be wrong. Many well-known accidents [13][14][16][20] are due to data races [7], a common type of concurrency bugs.

The source code of a multithreaded program written in many programming languages such as C/C++ should be firstly compiled into the object code format so that dynamic race detection can be performed on it. In the compiling process, programmers may optimize their code for different objectives such as reducing the object code size. The compilers support such program optimizations by providing a suite of different options for programmers to choose. For instance, the GNU GCC compiler [9] has a long list of optimization options [10], and some of them have been popularly used by many real-world multithreaded programs.

Many existing dynamic race detection strategies either passively monitor a program execution to detect races [3][7] or actively manipulate the thread schedules of a program execution with the aim of exposing races with higher probabilities [2][24]. Almost all existing empirical studies either are in the form of surveys of bug samples (e.g., [17]), or mine the artifacts in bug repositories (e.g., [8][23]) to reveal certain concurrency bug characteristics. To the best of our knowledge, Jia and Chan [12] reported the first study on systematically examining the effects of multiple compiler optimization options on the effectiveness of dynamic race detection for multithreaded programs. However, none of them systematically examines and compares the efficiency aspect of race detection for multithreaded programs compiled with different compiler optimization options.

We thus ask the following question in this paper: Is the efficiency (performance in particular) of the precise online dynamic race detection sensitive to the change in the compiler optimization option used to produce the object code? Fig. 1 illustrates our research question.

This paper reports a pilot study for this research question based on four benchmarks of the PARSEC 3.0 suite [21] and six representative compiler optimization options [10] of the GNU GCC compiler [9]. We studied the slowdown factor and memory footprint of the precise online race detection. To the best of our knowledge, this paper is the first work that examines the efficiency aspect of data race detection in multiple compiler optimization option settings.

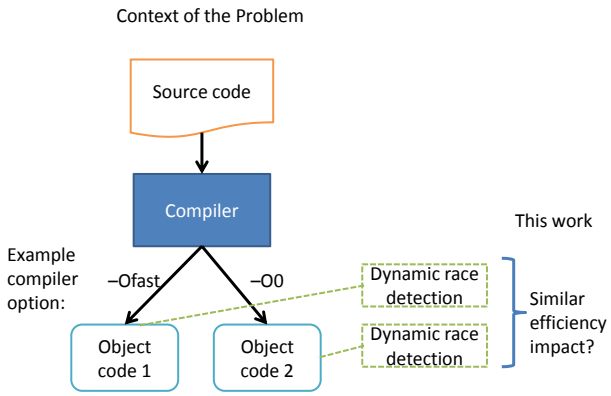

Figure 1. Our research question in illustriaton.

The empirical result shows that different standard optimization options do not make any significant difference among them in terms of both slowdown factor and memory footprint. In terms of slowdown factor, they are also comparable to the optimization options for speed and code, but they are quite different from the baseline option. It is surprising to observe from the experiment that in three out of the four benchmarks, using a standard optimization option is less efficient than using the baseline option. In terms of memory footprint, the standard optimization options consume comparable memory costs as the baseline option and the option for speed, but consume less memory than the option for code size.

The rest of the paper is organized as follows. Section II reviews the related work to set the context of our work. Section III revisits the preliminaries about the precise online dynamic race detection. Section IV formulates a suite of research questions to be investigated in the pilot study presented in this paper. Section V describes the experimental setup of the pilot study. Section VI analyzes the empirical data to answer the research questions. Section VII concludes the paper.

II. Related Work

Many concurrency bug detection techniques [2][7][24] have been proposed to detect bugs in multithreaded programs. Many mainstream software vendors such as Intel, Oracle, Microsoft, Google, and IBM also provide tools [18][25] to assist programmers to test their own programs. At the same time, many empirical studies on concurrency bugs in real-world multithreaded programs have been reported in the literature.

Lu et al. [17] described selected characteristics of concurrency bugs observed from their collected programs and how these bugs can be related to the program elements and structures to reveal them. Fonseca et al. [8] expressed that many such studies centered around the *causes* (i.e., error propagation chains) of the studied concurrency bugs, and they [8] studied the *effects* of concurrency bugs. Sahoo et al. [22] found that many concurrency bugs reported in the open-source bug repositories of the selected server applications could not be reproduced deterministically, and these reproducible bugs only constituted a small portion of all the bug records they have examined.

More recently, there were many pieces of work that reported their results from mining the bug repositories and code changes to study, say, the bug distributions and other characteristics in programs (e.g., [5][19][23]). However, they all have not studied how and to what extent different compiler optimization options may have affected the detection of concurrency bugs.

Both the work of Jannesari et al. [11] and the work of Yan and Chan [3] reported the results of race detection on benchmarks compiled with the GNU GCC compiler optimization option `-O2`. Both pieces of work have not reported the findings on other compiler optimization options. To the best of our knowledge, a vast majority of other validation experiments on C/C++ program benchmarks for concurrency bug detection and healing did not report their compiler settings in their respective work.

Jia and Chan [12] reported the first controlled experiment to study the effect of different compiler optimization options on the effectiveness of dynamic data race detection. Their results have shown that a precise online race detector could detect different sets of races with different probabilities from the same program compiled with different compiler optimization options. However, they have not reported results on any non-functional dimension, including the performance and memory footprints as what we did in the pilot study reported in this paper. The present paper complements the work of Jia and Chan [12].

III. Data Races

Data races are a popular kind of concurrency bugs. We define it according to one of the most popular definitions [7]. A *data race* is said to occur when two accesses to the same memory location, at least one of them being a write operation, and their order is not protected by the program according to a concurrency model. Some authors also refer to this type of data race as predictive data races. In our

previous work [3][12][28], we have used this predictive data race definition where the concurrency model is the classical happened-before model [15].

A precise online race detector monitors a set of critical events in an execution trace of a multithreaded program over an input to detect data races. A critical event is an event of interests for race detection. For example, a lock acquisition event performed by a thread at a particular program site [2][27] is a critical event. Other critical events for the precise online race detection include lock release events, thread management events, and memory read/write access events. In this paper, we refer to LOFT [3] and FastTrack [7] as examples of *precise online race detectors*.

A precise online race detector tracks the happened-before relations [15] among these critical events to construct an online version of a causality graph. At the same time, it uses this causality graph to determine whether any data race has occurred in this particular execution trace up to the moment of the latest memory access event occurrence. More specifically, if in the online version of the causality graph, two accesses to the same memory location are currently not related by any happened-before relation, and at least one of these two accesses is a write operation, a (predictive) data race on this memory location due to these two accesses is reported.

Data races may occur at multiple memory locations in a program execution. Some of them are harmful races that can affect the correctness of the program's execution, while the others are benign races. A (benign) race may be intentionally introduced by programmers to improve the program performance and do not compromise the program's correctness. However, without detecting data races in the first place, developers could not determine whether a potential race is harmful or not.

IV. RESEARCH QUESTIONS

In this section, we formulate a suite of eight research questions to be investigated in this pilot study. They examine four different levels of the compiler optimization options including the baseline, the standard options, the option for speed and the option for size. These option levels are summarized in Table I.

Comparison among the standard optimization options:

RQ1: [Slowdown] Does data race detection on a program compiled with a higher standard optimization option ***run faster*****?**

RQ2: [Memory footprint] Does data race detection on a program compiled with a higher standard optimization option ***incur a higher memory footprint*****?**

Comparison between the baseline option and standard optimization options:

RQ3: [Slowdown] Does data race detection on a program compiled with a standard optimization option have any advantage or disadvantage in terms of ***slowdown factor*** compared to that on the same program compiled with the **baseline** optimization option?

RQ4: [Memory footprint] Does data race detection on a program compiled with a standard optimization option have any advantage or disadvantage in terms of ***memory footprint*** compared to that on the same program compiled with the **baseline** optimization option?

Comparison between the other popular optimization options and the standard level optimization options:

RQ5: [Slowdown] Does data race detection on a program compiled with the ***optimization for speed*** option ***run faster*** than that on the same program compiled with one of the standard level optimization options?

RQ6: [Memory footprint] Does data race detection on a program compiled with the ***optimization for speed*** option ***incur a higher memory footprint*** than that on the same program simply compiled with one of the

TABLE I. COMPILER OPTIMIZATION OPTIONS USED

Treatment level	Option	Description
Baseline	`-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.
Standard Optimization	`-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. "
Optimization for Speed	`-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`."
Optimization for Size	`-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

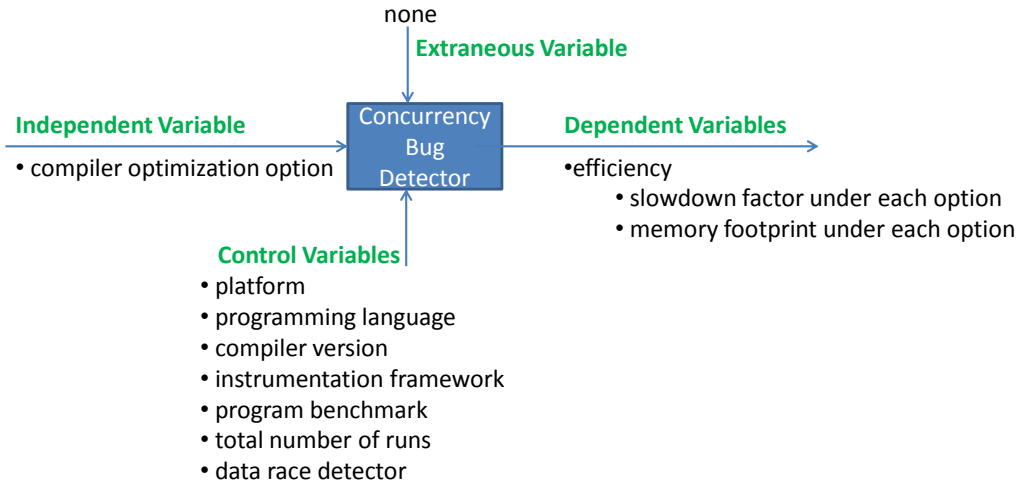

Figure 2. Theorectical Framework for Investigation of the Compiler Optimization Options on Concurrency Bug Detectors (modified from [12]).

standard level optimization options?

RQ7: [Slowdown] Does data race detection on a program compiled with the ***optimization for size*** option ***run slower*** than that on the same program compiled with one of the standard level optimization options?

RQ8: [Memory footprint] Does data race detection on a program compiled with the ***optimization for size*** option ***incur a higher memory footprint*** than that on the same program simply compiled with one of the standard level optimization options?

V. Experimental Setup

In this section, we present the setup of the experiment. We aim at studying the efficiency aspect with two dependent variables: slowdown factor and memory footprint. Fig. 2 shows the framework of the pilot study.

A. Control Variables

A control variable is a setting that remains unchanged throughout the experiment.

The **platform** to conduct the experiment was fixed. We performed the experiment on a Dell desktop machine running the 32-bit desktop Ubuntu Linux 12.04.1 with 3.16GHz Duo2 processor and 3.8GB physical memory. This platform provided the genuine thread control of the execution of each program benchmark in the experiment. Our previous work [3][12] has demonstrated that this platform was able to repeat the third-party experimental results [7]. However, the efficiency performance of the race detection is hardware-sensitive. Thus, using other platforms to repeat the same experimental procedure will likely change the data observed in a replication of the present experiment.

We used C/C++ as the **programming language**. C/C++ is one of the most influential languages used to develop real world large-scale and significant multithreaded programs.

The **compiler version** we used was GNU GCC 4.6.3 [9]. This series of compiler is arguably the de facto compiler used in the Linux operating systems. We chose the latest stable version of GNU GCC compiler (gcc 4.6.3) installed on the platform (i.e., Ubuntu Linux 12.04.1) at the time when we conducted the experiment. This compiler version selection limited all the compiler optimization options that we could select in the present experiment. We leave the generalization on the compiler version in a future work.

The **instrumentation framework** we used was Pin 2.11 [18]. Pin is released by Intel and has been widely used by many research-based program analysis tools [2][3][25] for dynamic instrumentation. It supports the Pthreads threading model only. As such, it also constrained us in selecting the program benchmarks in the experiment.

The **program benchmark** used in the experiment consisted of four programs selected from the PARSEC 3.0 suite [1][11][21]. These four programs (in the same or different versions) have been used in our previous data race experiments [3][4][12]. Note that the experiments reported in existing work [3][4] used the versions of the four benchmarks taken from PARSEC 2.0. We had verified that the versions of the four programs had been evolved to new versions in PARSEC 3.0. The description statistics of these benchmarks are shown in Table II.

To enable the experiment result to gain statistical powers, we set the **total number of runs** as 100 on each object code of each program benchmark. This value was also used by many other previous experiments on concurrency bug detection (see [2][12] for example).

Compared to FastTrack [7], the LOFT tool [3] maintains the same race detection precision and improves the race detection efficiency by not recording the redundant lock events (i.e., those that do not contribute to happened-before relations between threads in the same execution). We have extended the implementation of the LOFT tool by handling

TABLE II. SIZES OF SOURCE CODE AND OBJECT CODE UNDER DIFFERENT OPTIMIZATION OPTIONS OF THE BENCHMARKS

Optimization Options	Program Benchmark			
	blackscholes	dedup	x264	vips
	Size of Lines of Source Code			
	914	3347	41933	138536
	Size of Object Code (in '000 bytes)			
`-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

more Pthreads primitives instead of merely the basic ones (e.g., *pthread_mutex_lock/unlock*) discussed in the paper [3]. We used this extended version of LOFT as the **data race detector** in our experiment.

The present pilot study selected one value for each control variable to investigate the research questions. All the chosen values of the control variables, however, posted limitations on this pilot study. We leave the generalization of the study such as increasing the number of possible values of each control variable as a future work.

B. Independent Variable

An independent variable is a variable whose effect is investigated in the experiment. We studied one independent variable: **compiler optimization option**. Table I shows the available GNU GCC compiler optimization options as well as their treatment levels studied in the present experiment.

Specifically, we treated `-O0` as the baseline. According to the descriptions in Table I, options `-O1`, `-O2`, and `-O3` are increasingly optimized, and we treated them as different standard optimization options. We treated the other two options `-Ofast` and `-Os` as the optimization for speed and the optimization for size, respectively.

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

C. Dependent Variables

A dependent variable is a variable that reflects the effect of the independent variable in the experiment. The experiment has two dependent variables: **slowdown factor** and **memory footprint**. We measured them to study the sensitivity of race detection on the efficiency aspect due to the change in the level of the independent variable presented in Section V.B.

Through this pilot study, we would like to answer the eight research questions stated in Section IV by analyzing the measured values of the two dependent variables.

D. Experimental Procedure

This section describes the experimental procedure of the study.

For each program benchmark (see Table II), we compiled the program source code using each of the six compiler optimization options (see Table I). As such, we generated six versions of the object codes for each program benchmark.

To measure the differences between different versions of the object code belonging to the same program benchmark, we firstly used *SLOCCount* [26] to count the lines of source code (LOC) of the program benchmark. Then we used the Linux utility `du` [6] to measure the size of each object code. Table II shows the source code size of each benchmark as well as the size of different versions of object codes.

The descriptive statistics in Table II show that the LOCs of program benchmarks in the experiment ranged from 914 to over 138536. Moreover, the differences in size among the object codes were noticeable in some benchmarks (e.g., more than 50% in vips) and small in some other benchmarks (e.g., less than 10% in blackscholes).

For each version of the object code, we set the total number of working thread number as 8. Following [4][11], we also set the test input as the simsmall suite taken from the PARSEC 3.0 suite.

To measure the two dependent variables, we used the Linux utility `time` to collect the *execution time* (i.e., %S + %U) and the *memory cost* (i.e., %M) of each execution enabling race detection (i.e., we refer to it as a *race run*) and each execution disabling race detection (i.e., we refer to it as a *native run*).

Then, we computed the *slowdown factor* of a race run as the execution time spent by the race run divided by the mean execution time spent by the native runs. Similarly, the *memory footprint* of the race run was computed as the memory cost of a race run divided by the mean memory cost of the native runs.

We then configured a test script to run each version of the object code with the above configuration for 100 times.

As such, for each version of object codes of each program benchmark, we got a data set that contained 100 entries for the two dependent variables.

Based on the data set of each version of the object code, we computed the statistics and compared them to help us to answer the research questions. Specifically, we used the boxplot graph (see Fig. 3 and Fig. 4) to study the distribution of the 100 value entries of each version of object code collected in the data set. We also tabulate the comparison between different versions of the object codes of the same program benchmark by their mean values and standard deviation values. To ease our comparison, we chose the option `-O0` as the baseline and computed the normalized values of the other options. Table III shows these normalized results. We will discuss them in Section VI.

E. Threats to Validity

This pilot study has a number of threats to validity.

Threats to Construct Validity: This pilot study only investigated the effect of compiler optimization options on detecting data races. It had not measured the effects of such options on the detection of other types of concurrency bugs (e.g., deadlocks, atomicity violations). The root causes of data races or the other concurrency bugs were also not examined in this work. Intuitively, the investigation framework shown in Fig. 2 could be applied to study the differences in effects due to differences in compiler optimization options chosen on detecting other types of concurrency bugs. So generalizing the study to other types of concurrency bugs can help to further consolidate the results of this pilot study. We used the Linux utility `time` to collect the raw data for the two dependent variables. The use of other methods to collect data for the two dependent variables may give different results from ours.

Threats to Internal Validity: The major threat to internal validity was the reliability of the experimental tools, including the data race detector and data analysis tool. We had assured the two tools with some small programs. Our race detector was built on top of a third-party framework (i.e., Pin). Any race activated by this underlying framework in the experiment posted a threat to our study, and yet we had not observed abnormal situations in the experiment. Although we had described all control variables and independent variables in Section IV.A and Section IV.B, there may be confounding factors that we may not be aware of.

Threats to External Validity: This study only investigated one value for each control variable except the program benchmark. For the program benchmark, the four PARSEC 3.0 programs used in its experiment certainly exhibit different behaviors from some other real-world programs. A further experiment on other settings can make the result drawn from the pilot study more persuasive. This work can be generalized to all precise online race detectors, yet our experiment should not be generalized to imprecise (race) detectors.

VI. Data Analysis

Fig. 3 presents the summary on the slowdown factor for each version of the object code in boxplot. For different versions of the object code compiled from the same program benchmark, their data were shown in the same sub-graph in Fig. 3. For example, the sub-graph blackscholes in Fig. 3 shows the distribution of 100 slowdown factor data entries of each of the six different versions of the object code for blackscholes. Fig. 4 shows the distribution of memory footprint organized in the same way as that in Fig. 3.

In Fig. 3, the *x*-axis shows from left to right the compiler optimization options shown in Table I listed from top to bottom. The *y*-axis shows the value of the slowdown factor of the race detection. We observed from all four sub-graphs that the length of the box in each bar is very short, which indicates that the slowdown factors of different race runs on the same version of the object code did not change significantly in the experiment.

The slowdown factors among the four sub-graphs range from 22x to almost 100x. The difference is large.

In Fig. 4, the *x*-axis can be interpreted similar to the *x*-axis in Fig. 3. The *y*-axis shows the value of memory footprint of the race detection. The length of the box in each bar in Fig. 4 is also very short.

The memory footprints among the four sub-graphs range from 15x to 134x. The difference is also large. Comparing the corresponding bars between Fig. 3 and Fig. 4 however do not show a consistent trend between the slowdown factor and the memory footprint.

Both Fig. 3 and Fig. 4 show that the value differences in the two dependent variables among different race runs were small in the experiment. Therefore, we used the mean value and the standard deviation value for our further comparison. To make the presentation clean to readers, we took the option `-O0` as the baseline and normalized the mean value of other options by taking the value of this baseline as 1. The normalized results are shown in Table III.

Table III consists of two sections. The sections entitled "*Normalized mean slowdown factor*" and "*Normalized mean memory footprint*" show the normalized mean value of the slowdown factor and the memory footprint on each version of the object code of each program benchmark, respectively. The section entitled "*Standard derivation*" shows the standard deviations of the slowdown factor and the memory footprint. (Note that we did not normalize the standard derivation.)

We observed that compared to the baseline option `-O0`, the other five options incurred higher overheads in terms of the slowdown factor on all benchmarks except blackscholes. We also observed that the differences in the normalized slowdown factor on the five options were not large (except on blackscholes for the option `-Ofast`).

For the memory footprint, we found that all standard optimization options and the option for the optimization for speed showed very small differences among them. On the other hand, the option for the optimization for size `-Os` incurred significantly higher memory footprints on two benchmarks (i.e., dedup and x264) than the other five options.

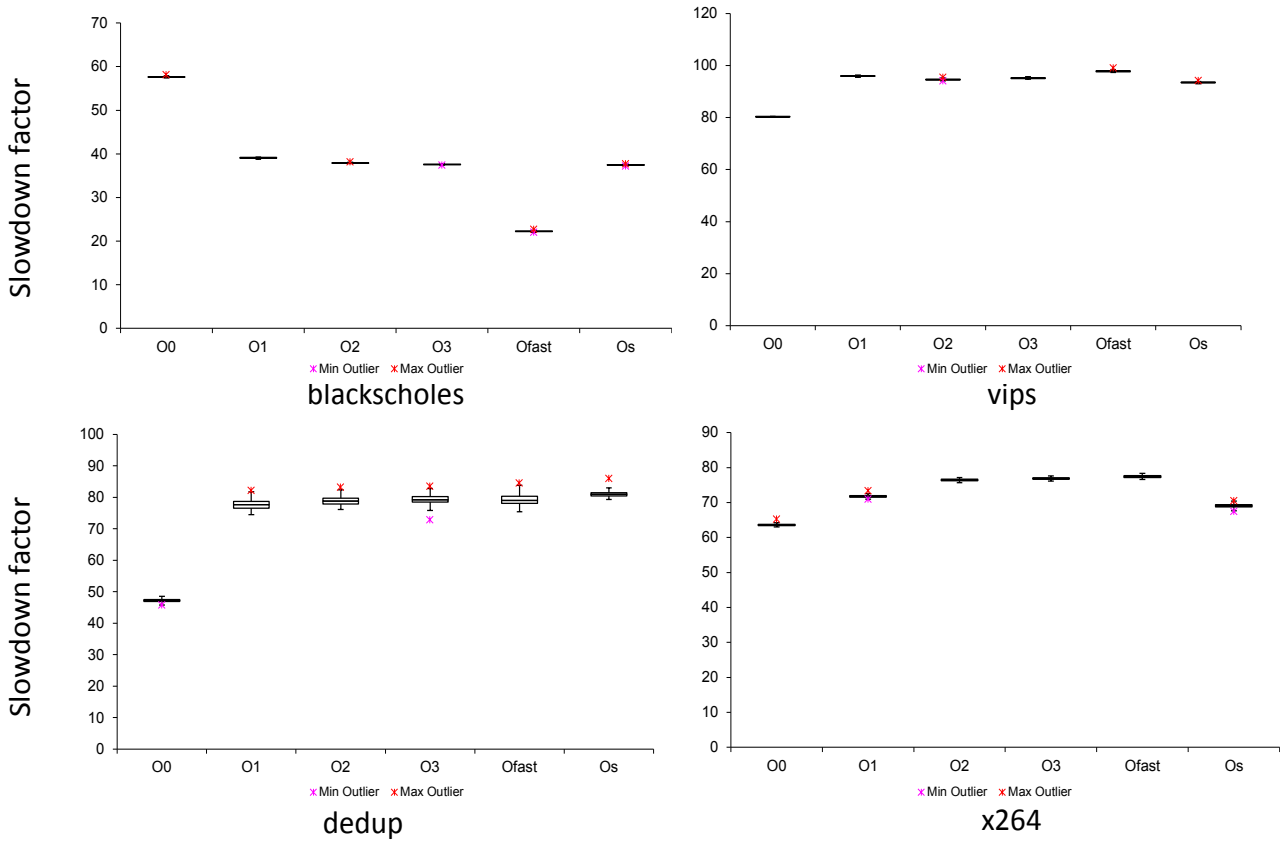

Figure 3. Slowdown factor of race detection on benchmarks for each of the six optimization options.

We also observed that the standard deviations in all cases ranged from 0.08 to 1.69 for the slowdown factor and ranged from 0.04 to 0.68 for the memory footprint. Both ranges were small, compared to values of the slowdown factor and the memory footprint as shown in Fig. 3 and Fig. 4, respectively.

For RQ1 and RQ2, we found that the normalized slowdown factor and memory footprint across the standard level optimization options (`-O1`, `-O2`, `-O3`) were only different marginally by less than 7% on average.

For RQ3, we found that in 9 out of 12 cases (75%) in the experiment, using a standard optimization option made race detection incur a higher slowdown factor than using the baseline option. In the remaining 25% of cases, using the former option made race detection faster to complete. Intuitively, the primary purpose of turning on a standard optimization option to compile a program by a programmer is to improve the performance of the corresponding object code. The increases in the slowdown factor observed in the experiment surprised us. It seems indicating that the adoption of a standard optimization option may *not* improve the running time performance of programs executing with a race detector, and in some cases, it worsens the situation significantly.

For RQ4, we found that applying each of the three standard optimization options and the baseline options consumed comparable amounts of memory cost (i.e., on blackscholes, dedup and x264), but the former three options consumed less memory than the latter option (on vips).

For RQ5, the option for the optimization for speed (`-Ofast`) may *not* make race detection complete earlier compared to the standard level optimization options, which is also surprising. At the same time, in the experiment, this option seldom made a program run much slower either. We only observed one case (blackscholes) that this option made a significant difference and made the race detection to complete faster than a standard optimization option.

For RQ6, we did not observe any significant difference in terms of memory footprint between the option for the optimization for speed (`-Ofast`) and the standard optimization options on three benchmarks (vips, dedup and x264). On blackscholes, the option for speed consumed slightly more memory than each standard optimization option, and the difference was only less than 7%.

For RQ7, the option for the optimization for size (`-Os`), compared to the standard level optimization options, had neither the advantages nor the disadvantages in terms of the slowdown factor in our experiment.

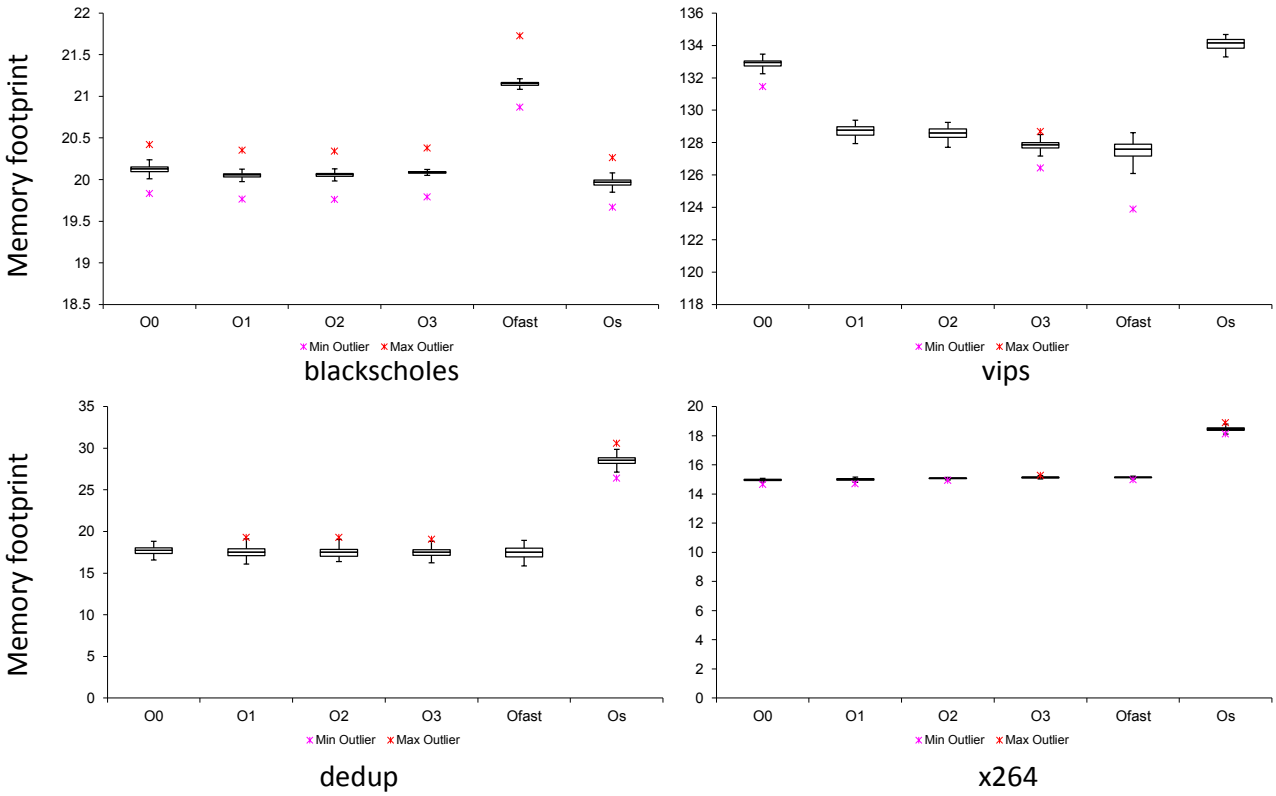

Figure 4. Memory footprint of race detection on benchmarks for each of the six optimization options.

TABLE III. DETECTED RACES OF EACH OJECT CODE OF EACH BENCHMARK

Benchmark	-O0	-O1	-O2	-O3	-Ofast	-Os	-O0	-O1	-O2	-O3	-Ofast	-Os
	Normalized mean slowdown factor						***Standard deviation***					
blackscholes	1.00	0.68	0.66	0.65	0.39	0.65	0.10	0.10	0.09	0.08	0.12	0.09
dedup	1.00	1.65	1.67	1.68	1.68	1.72	0.55	1.59	1.43	1.57	1.69	0.96
x264	1.00	1.13	1.20	1.21	1.22	1.08	0.32	0.34	0.32	0.32	0.35	0.56
vips	1.00	1.20	1.18	1.19	1.22	1.16	0.11	0.18	0.19	0.18	0.35	0.18
Average	1.00	1.17	1.18	1.18	1.13	1.15	0.27	0.55	0.51	0.54	0.63	0.45
	Normalized mean memory footprint						***Standard deviation***					
blackscholes	1.00	1.00	1.00	1.00	1.05	0.99	0.15	0.16	0.16	0.16	0.17	0.17
dedup	1.00	0.99	0.99	0.99	0.99	1.61	0.48	0.60	0.63	0.53	0.60	0.68
x264	1.00	1.00	1.01	1.01	1.01	1.23	0.08	0.09	0.04	0.05	0.04	0.15
vips	1.00	0.97	0.97	0.96	0.96	1.01	0.32	0.35	0.35	0.36	0.85	0.32
Average	1.00	0.99	0.99	0.99	1.00	1.21	0.26	0.30	0.30	0.28	0.42	0.33

For RQ8, the option for the optimization for size (`-Os`), compared to the standard level optimization options, incurred higher overheads in two (`dedup` and `x264`) out of the four benchmarks by 60% and 23%, respectively, in terms of the memory footprint. This margin of difference was large and should not be simply ignored.

To sum up, this pilot study found that different standard compiler optimization options behave similarly in terms of both the execution time and the memory cost. However, we did observe that there was a noticeable difference between the baseline option and these three standard optimization options in terms of slowdown factors. It is also surprising to us that turning on such a standard optimization option, when compared to the baseline option, may worsen the time taken by race detection in some cases. Moreover, turning on the optimization for size may consume more memory than turning on a standard optimization option for race detection without a clear benefit to complete the race detection earlier. Last, but not the least, turning on the optimization for speed may make race detection complete earlier, but it may consume more than turning on a standard optimization option.

VII. CONCLUSION

Multithreaded programs written in some programming languages such as C/C++ should be compiled into the object code format before dynamic detections of concurrency bugs can be performed on them. Such a compilation process is highly configurable. Choosing different configurations may affect the generated object code to be used for concurrency bug detection, which may in turn affect the performance of such detection.

In this paper, we have reported a pilot study on the efficiency aspect of data race detection from the compiler optimization option perspective. We have conducted a controlled experiment using four benchmarks with six options of the GNU GCC compiler. The empirical results have shown that using different standard optimization options did not incur significantly differences among them in terms of the slowdown factor and the memory footprint. Moreover, compared to the optimization option for speed, the use of a standard optimization option did not incur significant disadvantages or advantages. Compared to the baseline option, surprisingly, the use of standard optimization options often incurred higher slowdown factors, but their memory footprints were usually comparable. Compared to the optimization option for size, use of a standard optimization option incurred a similar slowdown factor, but may potentially consume significantly less memory.

A deeper investigation to find out the underlying reasons of the reported observations can be an interesting future work.

ACKNOWLEDGEMENT

This research is supported in part by the General Research Fund of Research Grants Council of Hong Kong (project numbers 111410 and 123512).

REFERENCES

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The PARSEC benchmark suite: characterization and architectural implications," In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (PACT '08). ACM, New York, NY, USA, 72−81, 2008.

[2] Yan Cai and W. K. Chan, "MagicFuzzer: scalable deadlock detection for large-scale applications," In *Proceedings of the 2012 International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 606−616, 2012.

[3] Y. Cai and W.K. Chan, "LOFT: Redundant synchronization event removal for data race detection," In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering* (ISSRE '11). IEEE Computer Society, Washington, DC, USA, 160−169, 2011.

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

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (SOSP '01). ACM, New York, NY, USA, 73−88, 2001.

[6] The Du Linux/Unix command: estimate file space usage. Available at *http://linux.about.com/library/cmd/blcmdl1_du.htm*. Last access 6 Feb 2013.

[7] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '09). ACM, New York, NY, USA, 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 *Proc. of International Conference on Dependable Systems and Networks* (DSN '10). IEEE Computer Society, Washington, DC, USA, 221−230, 2010.

[9] GCC Compiler. Avaliable at *http://gcc.gnu.org/*. Last access 6 Feb 2013.

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

[11] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: an efficient dynamic race detector," In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing* (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1−13, 2009.

[12] C. Jia, and W.K. Chan, "Which compiler optimization options should I use for detecting data races in multithreaded programs?," In *Proceedings of Automated Software Test (AST 2013),* to appear, 2013.

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

[14] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering,* 30, 6, 418−421, 2004.

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

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

[17] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII). ACM, New York, NY, USA, 329−339, 2008.

[18] 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 the 2009 ACM SIGPLAN conference on Programming*

language design and implementation (PLDI '05). ACM, New York, NY, USA, 191−200, 2005.

[19] N. Palix, G. Thomas, S. Saha, C. Calves, J. L. Lawall, and G. Muller, "Faults in Linux Ten years later," In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems* (ASPLOS '11). ACM, New York, NY, USA, 305−318, 2011.

[20] K. Poulsen. Software bug contributed to blackout. Avaliable at http: //www.securityfocus.com/news/8016, Feb. 2004. Last access 12 Dec 2012.

[21] PARSEC benchmark 3.0. Avaliable at *http://parsec.cs.princeton.edu/*. Last access 6 Feb 2013.

[22] 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 the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE '10). ACM, New York, NY, USA, 485−494, 2010.

[23] C. Saowski, J. Yi., and S. Kim, "The Evolutin of Data Races," In *Proceedings of Mining Software Responsitories* (MSR '12). IEEE Computer Society, Washington, DC, USA, 171−174, 2012.

[24] Koushik Sen, "Race directed random testing of concurrent program," In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '08). ACM, New York, NY, USA, 11−21,2008.

[25] Konstantin Serebryany and Timur Iskhodzhanov, "ThreadSanitizer: data race detection in practice," In *Proceedings of the Workshop on Binary Instrumentation and Applications* (WBIA '09). ACM, New York, NY, USA, 62−71, 2009.

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

[27] Xinwei Xie and Jingling Xue, "Acculock: Accurate and efficient detection of data races," In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '11). IEEE Computer Society, Washington, DC, USA, 201−212, 2011.

[28] 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 the 2012 International Symposium on Software Testing and Analysis* (ISSTA '12). ACM, New York, NY, USA, 221−231, 2012.