2018 42nd IEEE International Conference on Computer Software & Applications

GBRAD: A General Framework to Evaluate Design Strategies for Hybrid Race Detection†

Jialin Yang, W. K. Chan‡, Y. T. Yu, Jacky Keung
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Kowloon
Hong Kong
jialiyang4@gapps.cityu.edu.hk, {wkchan, csytyu, Jacky.Keung}@cityu.edu.hk

***Abstract*—Data race detection is a method in testing multithreaded programs to ensure their reliability against concurrency errors. In this paper, we present the GBRAD framework to support the initialization of various hybrid race detection techniques, which also supports the evaluation of these strategies at two decision points based on two major design factors of hybrid race detectors. In the GBRAD framework, one decision point consists of six skipping strategies and another decision point consists of eight reduction strategies. By combining these strategies, 48 hybrid detection techniques are initialized. We report a controlled experiment on the PARSEC benchmark suite as well as four real-world applications to evaluate these 48 techniques and their strategies in terms of runtime slowdown, memory overhead, and race detection effectiveness. The experiment identified 9 previously unknown techniques that are comparable to the state-of-the-art hybrid race detection technique.**

Keywords-multithreaded program; data race; lockset discipline violation; dynamic analysis; controlled experiment

I. INTRODUCTION

A data race (or *race* for short) occurs when two threads in an execution trace access the same memory location without any protection of their access order by synchronization primitives, and at least one of these accesses is a write access [6]. Data race occurrences may lead to memory state corruption, application-non-responding (ANR), or program crashes. The presence of races in a program may also correlate to the presence of other concurrency bugs, such as linearizability error [2] and atomicity violations [6], [14].

Hybrid race detectors (e.g., [7]) combine both lockset-based [8] and happens-before based [4] techniques. For hybrid race detection, different design strategies can be used for achieving required performances in runtime slowdown, memory overhead, and detection effectiveness. However, to the best of our knowledge, no prior work in the literature has systematically evaluated those strategies.

In this paper, we present a framework called GBRAD (Generic Hybrid Race Detection) by formulating a generic hybrid race detection algorithm and supporting two decision points for designers to choose different strategies to initialize a technique through the framework. One decision point is the memory access skipping strategy which aims to reduce performance slowdown, and GBRAD included 6 such strategies. Another decision point is the history reduction strategy which aims to reduce memory consumption, and GBRAD included 8 such strategies. Based on the framework and the included design strategies, we conducted a controlled experiment to evaluate 48 hybrid race detection techniques initialized from these two decision points.

The contribution of this work is twofold: (1) We formulate a new framework with two decision points based on two major design factors on performance optimization for sound hybrid race detection. (2) We report the *first* controlled experiment to systematically evaluate 48 hybrid race detection techniques in terms of runtime slowdown, memory overhead, and race detection effectiveness. The results show that 9 *previously unknown* techniques achieved comparable performance to the state-of-the-art.

The rest of this paper is organized as follows: Section II reviews the preliminaries. Section III introduces our framework. Section IV presents a controlled experiment to evaluate different design strategies and techniques initialized by our framework. Section V discusses the closely related work. Section VI concludes this paper.

II. HYBRID RACE DETECTION

Contemporary hybrid race detectors (e.g., [11][12]) aim to detect Ø-races [11] by employing *mhb* relation [11] analysis and lockset [8] comparison among memory accesses in an execution trace.

An execution trace is a sequence of events $[e_1, e_2, \ldots, e_n]$. Each event e has a meta type, which can be memory access or synchronization primitive. Memory access is further divided into two sub-types: $write(t, x)$ and $read(t, x)$, which denote update and read memory location x by thread t, respectively. Synchronization primitive also has a number of sub-types: $fork(t, u)$ models thread t forking a child thread u, $join(t, u)$ models child thread u joining thread t, while $acquire(t, m)$ and $release(t, m)$ model thread t acquiring and releasing lock m, respectively.

Algorithm 1 shows the pseudo code of hybrid race detection. Generally, the hybrid race detection algorithm picks each event e in the given trace sequentially and tracks the *mhb* relation or updates lockset if e is a synchronization primitive (lines 1–4 in Algorithm 1). Otherwise, if e is a read access on a memory location x, it detects a write-read

† This research is supported in part by the General Research Fund of Research Grants Council of Hong Kong SAR (project numbers: 125113, 11200015, 11201114, 11208017, and 11214116).
‡ Correspondence author.

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00024

Algorithm 1: Hybrid Race Detection

Input: Trace τ of a program
Output: Ø-race report
Variable: W_x – list of historical writes on memory location x
R_x – list of historical reads on memory location x

```
1.  while τ is not empty do
2.      pop an event e from τ in the front
3.      if e is a synchronization operation then
4.          track mhb relation or update lockset
5.      else if e is a memory access then
6.          skip() // Factor 1: Skipping Strategy
7.          if e is a read on memory location x then
8.              for each write w in W_x do
9.                  detect write-read Ø-race between w and e
10.             end for
11.             append e to R_x
12.         else if e is a write on memory location x then
13.             for each write w in W_x do
14.                 detect write-write Ø-race between w and e
15.             end for
16.             for each read r in R_x do
17.                 detect read-write Ø-race between r and e
18.             end for
19.             append e to W_x
20.         end if
21.         reduce() // Factor 2: Reduction Strategy
22.     end if
23. end while
```

Ø-race between e and each prior write w on memory location x kept in the write history list W_x, and then appends e to the read history list R_x (lines 5–11 in Algorithm 1). If e is a write access on a memory location x, it detects a write-write Ø-race between e and each prior write w kept in W_x, detects read-write Ø-race between e and each prior read r kept in R_x, and then it appends e to W_x (lines 12–20 in Algorithm 1).

Factor 1 (Skipping Strategy): Since the number of events in a trace can be huge, hybrid race detectors incur severe slowdown overhead by analyzing each event consecutively. On the other hand, skipping memory accesses may miss races if the skipped memory access is involved in a race. As a result, *skipping strategy* for eliminating the checking on some memory accesses with high race detection effectiveness is a major design factor for hybrid race detectors (i.e., line 6 in Algorithm 1), which is modeled as `skip()` in the algorithm.

Note that skipping any synchronization primitive may lead to incorrect *mhb* relation or incorrect lockset for all subsequent events. Thus hybrid race detectors must analyze every synchronization primitive without skipping.

Factor 2 (Reduction Strategy): Keeping all (non-skipped) memory accesses in the history could be highly excessive. It will make a hybrid detector incur impractical memory overhead. Any practical hybrid race detector should maintain only a small subset of all memory accesses in the history. However, reducing the history may also lower race detection effectiveness if the removed memory access in the history is involved in a race. Thus, *reduction strategy* that makes trade-off between memory efficiency and detection effectiveness is another major design factor for hybrid race detectors (i.e., line 21 in Algorithm 1), which is modeled as `reduce()` in the algorithm.

III. Our Framework

We formulated the GBRAD framework to configure different strategies at the decision points (i.e., `skip()` and `reduce()` in Algorithm 1) of the two design factors discussed in last section.

A. Skipping Strategies

We included 6 skipping strategies in the framework: No Skipping (*X0*), Lock Acquisition (*X1*), Lock Release (*X2*), Fork-Join Synchronization (*X3*), Function Call (*X4*), and Thread Switching (*X5*). Fig. 1(*a*)–(*e*) show the examples of *X1* to *X5*, respectively. To the best of our knowledge, except *X0* and *X2* which had been employed by previous work (e.g., [11][12]), other strategies are newly proposed.

No Skipping (*X0*): Do not skip any memory access.

Lock Acquisition (*X1*): For each thread *t*, skip memory accesses other than the first read and the first write on each memory location *x* after every lock acquisition.

Lock Release (*X2*): For each thread *t*, skip memory accesses other than the first read and the first write on each memory location *x* after every lock release.

Fork-Join Synchronization (*X3*): For each thread *t*, skip memory accesses other than the first read and the first write on each memory location *x* after every thread fork or join.

Function Call (*X4*): For each thread *t*, skip memory accesses other than the first read and the first write on each memory location *x* after every function call. For ease of presentation, we model the event that thread *t* calls a function *f* as *call*(*t*, *f*) in Fig. 1(*d*).

Thread Switching (*X5*): Along a trace, skip memory accesses other than the first read and the first write on each memory location *x* after every thread switching.

B. Reduction Strategies

We included 8 reduction strategies in the framework: No Reduction (*Y0*), Keep Latest (*Y1*), Fixed Window Size (*Y2*), Minimal Lockset (*Y3*), Clear on Release (*Y4*), Clear on Fork/Join (*Y5*), Clear on Function Call (*Y6*), and Clear on Thread Switching (*Y7*). To the best of our knowledge, except *Y0*–*Y3* and *Y6* which had been employed by previous work (e.g., [11][12]), other strategies are newly proposed.

No Reduction (*Y0*): Do not clear histories R_x and W_x.

Keep Latest (*Y1*): For each thread *t*, only keep the latest read and the latest write on each memory location *x* in history R_x and W_x respectively, and discard other outdated memory accesses.

Fixed Window Size (*Y2*): Configure a window size for histories R_x and W_x on each memory location *x*, and remove the oldest read or write once the window size of its corresponding history is reached. Following [12], we configured this size as 100.

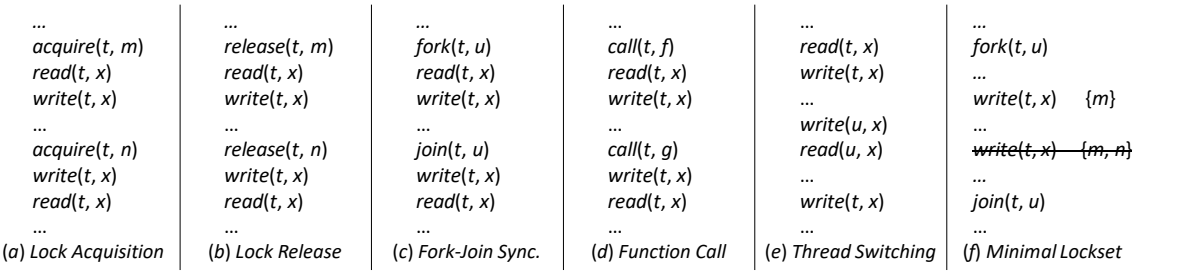

Fig. 1. Exemplified traces to illustrate skipping strategies and reduction strategies. There are 6 traces labeled as (*a*)–(*f*) in the figure. *t* and *u* are threads; *x* is a memory location; *m* and *n* are locks; *f* and *g* are functions; "…" denotes skipped memory accesses in (*a*)–(*e*) and irrelevant events in (*f*); strikethrough denotes discarded memory access in the history. Locksets are shown on the right in (*f*).

Minimal Lockset (*Y3*): For histories R_x and W_x on each memory location *x*, only keep the memory accesses with the smallest lockset within each fork-join span and discards any memory access that holds a larger lockset than (i.e., a superset of) the lockset held by other memory accesses in the history. For example, in Fig. 1(*f*), discard the second *write*(*t*, *x*) with lockset {*m*, *n*} in W_x since it has a larger lockset than the first *write*(*t*, *x*) with lockset {*m*} kept in W_x.

Clear on Release (*Y4*): Clear histories R_x and W_x on each memory location *x* when a lock release event is encountered. For example, clear R_x and W_x on *release*(*t*, *m*) and *release*(*t*, *n*) in Fig. 1(*b*).

Clear on Fork/Join (*Y5*): Clear histories R_x and W_x on each memory location *x* when a fork or join event is encountered. For example, clear R_x and W_x on *fork*(*t*, *u*) and *join*(*t*, *u*) in Fig. 1(*c*).

Clear on Function Call (*Y6*): Clear histories R_x and W_x on each memory location *x* when a function call is encountered. For example, clear R_x and W_x on *call*(*t*, *f*) and *call*(*t*, *g*) in Fig. 1(*d*).

Clear on Thread Switching (*Y7*): Clear histories R_x and W_x on each memory location *x* when a thread switching is encountered. For example, clear R_x and W_x on *read*(*t*, *x*), *write*(*u*, *x*), and *write*(*t*, *x*) in Fig. 1(*e*).

C. Hybrid Race Detection Techniques

From Algorithm 1, each distinct pair of skipping strategy (*X0* to *X5*) and reduction strategy (*Y0* to *Y7*) results in a distinct hybrid race detection technique. As such, we initialize 48 hybrid race detection techniques. We use *T*(*i*, *j*) to denote a technique modeled by *Xi* and *Yj*. For instance, we refer to the technique *T*(2, 6) modeled by *X2* and *Y6*, which is a technique modeled after the state-of-the-art hybrid race detector, HistLock [12] (without window size).

IV. Controlled Experiment

In this section, we report a controlled experiment on the design strategies and techniques initialized by GBRAD to answer three research questions.

A. Research Questions

We study the following three research questions:

RQ1: Does any skipping strategy (or technique) outperform the state-of-the-art in terms of *time* efficiency?

RQ2: Does any reduction strategy (or technique) outperform the state-of-the-art in terms of *memory* efficiency?

RQ3: Does any technique achieve comparable or better performance than the state-of-the-art in terms of both *efficiency* and *effectiveness*?

B. Implementation

Our framework was built atop Maple [13], which provides callback analysis function probes to instrument target events (e.g., memory accesses and synchronization primitives) of a program on-the-fly. We (1) realized the generic race detection algorithm (i.e., Algorithm 1) by implementing the callback analysis function on each interested event, and (2) implemented 6 skipping strategies and 8 reduction strategies as introduced in Section III.

C. Experimental Setup

We have initialized all the 48 techniques in GBRAD. For each execution trace, like previous work [10], every technique reports each racy pair of events involved in an ∅-race at the static statement level. We also note that like previous work (e.g., [10], [13], [33]), each technique treats different types of ∅-race (i.e., write-write ∅-race, write-read ∅-race, and read-write ∅-race) from the same racy statements as different ∅-race instances.

We chose the PARSEC [1] benchmark suite 2.1 as well as four real-world open-source applications Aget 0.57, Pbzip2 0.9.4, Apache HTTP server 2.0.48, and MySQL 4.0.12 as our benchmarks. The PARSEC suite is a set of C/C++ multithreaded programs also used in previous experiments (e.g., [3]) on race detection. Because *freqmine* was implemented by OpenMP API and our framework only supported POSIX Threads API (i.e., *pthreads*), and *facesim* crashed when we ran it under GBRAD, we excluded these two benchmarks from our evaluation. Our experiment used all the remaining 11 benchmarks and executed them with the shipped *simdev* [1] as test inputs.

Aget (*aget*) is a multithreaded download accelerator, and Pbzip2 (*pbzip2*) is a parallel compressor. Apache HTTP Server (*httpd*) and MySQL (*mysql*) are multithreaded web server and database server, respectively, widely used in practice. These four real-world applications are widely studied in the literature (e.g., [10], [13]) on concurrency bug detection. The test inputs of these four benchmarks are taken from [13].

TABLE I. BENCHMARK SUMMARY

Benchmark	Application Domain	KLOC	Thds.	Base Time	Base Memory
blackscholes	Financial Analysis	1.2	4	0.9	103
bodytrack	Computer Vision	11.0	5	62.0	284
canneal	Engineering	4.3	4	1.8	118
dedup	Enterprise Storage	3.7	12	54.0	2,864
ferret	Similarity Search	10.8	18	18.0	181
fluidanimate	Animation	2.1	4	101.0	2,625
raytrace	Rendering	14.7	4	116.0	300
streamcluster	Data Mining	2.8	8	20.0	2,590
swaptions	Financial Analysis	1.6	4	1.5	103
vips	Media Processing	134.3	3	8.7	167
x264	Media Processing	39.0	5	3.8	162
aget	Downloading	2.7	2	1.4	112
pbzip2	Data Compression	2.1	3	6.4	485
httpd	Web Server	231.3	4	8.3	189
mysql	Database	368.7	12	12.7	1,073
Total	-	830.3	-	416.5	11,356

Table I shows the summary of our benchmarks. Column "Application Domain" indicates the application domain of the benchmark. Column "KLOC" reports the size (i.e., thousands of lines of code) of each program. Column "Thds." shows the number of *worker* threads allocated by the benchmark, excluding the test harness thread that started the test run. Columns "Base Time" and "Base Memory" show the execution time (in second) and the memory consumption (in MB) that each benchmark ran on GBRAD with empty callback analysis functions.

Our experiment was performed in VMs each installed with Ubuntu 12.04 (64-bits) and configured with 2.2GHz 8-core processor and 16GB physical memory managed by VMware ESXi (version 6.0). Our framework and benchmarks were compiled by GCC 4.6.3. We ran each benchmark 100 times, and recorded the execution time, memory consumption, and distinct ∅-races reported at statement level of each technique.

D. Data Analysis

1) Answering RQ1

To factor out the overhead introduced by GBRAD itself, we calculated slowdown factor by the formula below:

$$Slowdown\ factor = (TechT - BaseT) / BaseT$$

In the formula, *TechT* is the execution time of each technique, and *BaseT* is the value in column "Base Time" in Table I. The lower slowdown factors are, the better the technique is.

After having measured the slowdown factors of each technique on each benchmark in 100 runs, we grouped the results of the same technique. The slowdown factors of all 48 techniques are shown in Fig. 2, where *x*-axis represents the technique and *y*-axis represents slowdown factor. In the figure, there are 8 boxplots. Each plot groups the results by reduction strategy from *Y0* to *Y7*. In each group, the results are sorted by skipping strategy from *X0* to *X5*.

From Fig. 2, overall speaking, the slowdown factors are observably sorted as $X0 > X1 = X2 > X4 > X5 > X3$ for all groups. The observation seems to indicate that skipping strategies based on coarse span (e.g., thread fork/join span) achieve higher time efficiency than skipping strategies based on fine span (e.g., lock acquisition /release span).

Since a majority of hybrid race detectors (e.g., [12]) employed *X2* (underlined in the *x*-axis of Fig. 2) as skipping strategy, we compared the slowdown factors of other skipping strategies with the slowdown factor of *X2*. Following the data analysis approach presented in [15], we compared the medians of the bars for other strategies with median of the bar for *X2* in each plot. If the median of the bar for a strategy is higher than the median of the bar for *X2*, we highlight its label in bold.

According to the figure, we found that *X3*, *X4*, and *X5* always performed better than *X2* under all reduction strategies (i.e., *Y0* to *Y7*) in slowdown factor. It indicates that the three skipping strategies, *X3*, *X4*, and *X5*, can be alternatives of *X2* in terms of time efficiency.

To the best of our knowledge, HistLock [12] is the state-of-the-art hybrid race detector in the literature. Thus, we selected *T*(2, 6) (which is modeled after HistLock) as the controlled technique and performed Fisher's LSD test [9] to compare other techniques with *T*(2, 6) at the 5% significance level in slowdown factor.

The statistical test results are shown as Table II. In the table, if the slowdown factor of a technique is larger than, the same as, and smaller than the slowdown factor of *T*(2, 6) by the LSD test, we mark the corresponding cell with ">", "=", and "<", respectively.

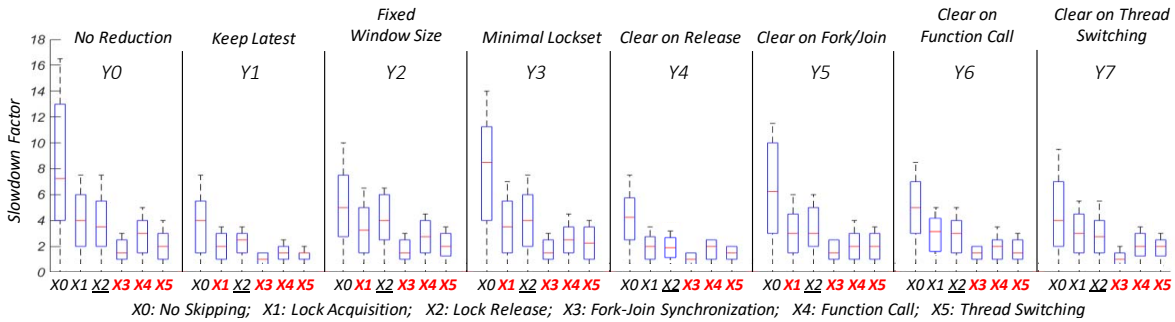

Fig. 2. Boxplots of slowdown factors for the 48 techniques grouped by reduction strategy (*Y0*–*Y7*) and sorted by skipping strategy (*X0*–*X5*) in each group. In the plot, *x*-axis represents the technique and *y*-axis represents slowdown factor. The lower the slowdown factors are, the better the technique is.

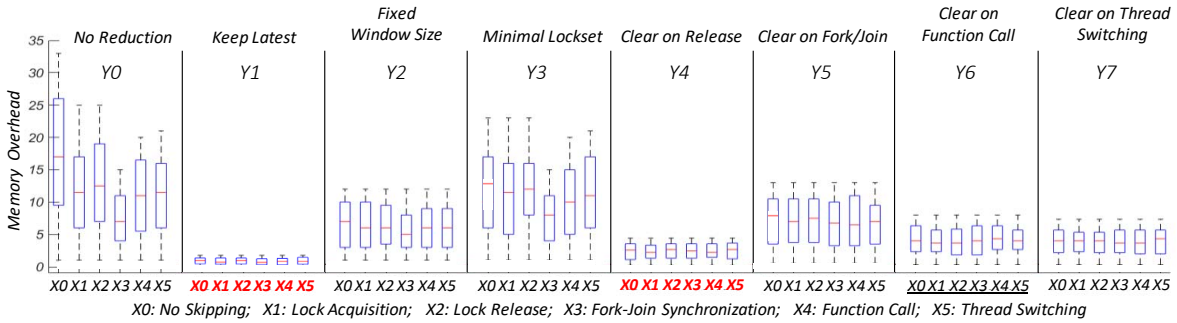

Fig. 3. Boxplots of memory overheads for the 48 techniques grouped by reduction strategy (*Y0*–*Y7*) and sorted by skipping strategy (*X0*–*X5*) in each group. In the plot, *x*-axis represents the technique and *y*-axis represents memory overhead. The lower memory overheads are, the better the technique is.

We shaded those cells with "<" in Table II. As the table shows, 27 out of 48 techniques can be more efficient than *T*(2, 6) in time efficiency.

2) Answering RQ2

Like last subsection, to factor out the overhead introduced by GBRAD itself, we calculated memory overhead by the formula below:

$$Memory\ overhead = (TechM - BaseM) / BaseM$$

In the formula, *TechM* is the memory consumption of each technique, and *BaseM* is the value in column "Base Memory" in Table I. The lower memory overheads are, the better the technique is.

Like analyzing slowdown factor in last subsection, after having measured the memory overheads of each technique on each benchmark in 100 runs, we grouped those results of the same technique regardless of the benchmarks.

Similar to Fig. 2, the memory overheads of all the 48 techniques are shown in Fig. 3. From the figure, the memory overheads under all reduction strategies are observably in the following order: *Y0* > *Y3* > *Y5* = *Y2* > *Y6* = *Y7* > *Y4* > *Y1*. The observations seem to indicate that reduction strategies with finer granularity (e.g., lock release) achieve higher memory efficiency than reduction strategies with coarser granularity (e.g., thread fork/join).

Since HistLock employs *Y6* (underlined in the *x*-axis of Fig. 3) as the reduction strategy, we compared the memory overheads of other reduction strategies with the memory overhead of *Y6*. Similar to the data analysis presented in last subsection, if all the medians of the bars in one group are higher than all the medians of the bars in *Y6*, we highlighted the labels of that group in *x*-axis in bold.

According to Fig. 3, we found that groups *Y1* and *Y4* always performed better than *Y6* under all skipping strategies (i.e., *X0* to *X5*). It indicates that *Y1* and *Y4* can be the alternatives of *Y6* in terms of memory efficiency.

Similar to last subsection, we selected *T*(2, 6) as the controlled technique and performed Fisher's LSD test to compare other techniques with *T*(2, 6) in memory overhead at the 5% significance level. The results are shown as Table III. If the memory overhead of a technique is larger than, the same as, and smaller than the memory overhead of *T*(2, 6) by the test, we marked the corresponding cell with ">", "=", and "<", respectively.

We shaded those cells with "<" in Table III. As the table shows, all techniques under *Y1* and *Y4* (12 out of 48) consumed significant less memory than *T*(2, 6).

3) Answering RQ3

Table IV shows the sum of all distinct ∅-races reported in 100 runs under each technique. Since *T*(2, 6) is our controlled technique, we shaded the cells with the numbers of reported ∅-races not fewer than 500, which is 95% of the total number of ∅-races reported by *T*(2, 6). From the table, 25 out of 48 techniques (excluding *T*(2, 6) itself) achieved equal or higher race detection effectiveness than *T*(2, 6).

TABLE II. FISHER'S LSD TEST FOR COMPARING INDIVIDUAL TECHNIQUE WITH *T*(2, 6) IN TERMS OF SLOWDOWN FACTOR

	Y0	***Y1***	***Y2***	***Y3***	***Y4***	***Y5***	***Y6***	***Y7***
X0	>	>	>	>	>	>	>	>
X1	>	<	>	>	<	=	=	=
X2	>	<	>	>	<	=	(Control)	=
X3	<	<	<	<	<	<	<	<
X4	=	<	<	<	<	<	<	<
X5	<	<	<	<	<	<	<	<

TABLE III. FISHER'S LSD TEST FOR COMPARING INDIVIDUAL TECHNIQUE WITH *T*(2, 6) IN TERMS OF MEMORY OVERHEAD

	Y0	***Y1***	***Y2***	***Y3***	***Y4***	***Y5***	***Y6***	***Y7***
X0	>	<	>	>	<	>	=	=
X1	>	<	>	>	<	>	=	=
X2	>	<	>	>	<	>	(Control)	=
X3	>	<	>	>	<	>	=	=
X4	>	<	>	>	<	>	=	=
X5	>	<	>	>	<	>	=	=

TABLE IV. TOTAL NUMBER OF ∅-RACES REPORTED BY EACH TECHNIQUE

	Y0	***Y1***	***Y2***	***Y3***	***Y4***	***Y5***	***Y6***	***Y7***
X0	582	483	572	544	501	576	538	529
X1	542	457	523	528	472	546	513	502
X2	551	471	535	540	485	564	**526**	517
X3	301	253	292	287	263	291	283	271
X4	535	452	526	503	467	532	498	490
X5	527	430	513	486	451	518	482	476

From Tables II, III, and IV, we observe that most techniques with either equal or higher race detection effectiveness than $T(2, 6)$ incur either heavy slowdown factor or heavy memory overhead. Since memory is much cheaper than computational power, there are benefits to trade memory for execution time. Besides, as race detection effectiveness is the major property of a race detector, any detector should retain high detection effectiveness for it to be practical. Thus, we consider a technique $T(i, j)$ as *comparable* to the sate-of-the-art technique $T(2, 6)$ if (1) $T(i, j)$ has equal or higher race detection effectiveness than $T(2, 6)$, and (2) $T(i, j)$ is more time efficient than $T(i, j)$, no matter whether it consumes more memory. We also treated $T(1, 6)$, $T(1, 7)$, and $T(2, 7)$ as comparable to $T(2, 6)$.

Using the criteria stated in last paragraph, we found that 9 out of 48 techniques (excluding $T(2, 6)$ itself) are comparable to $T(2, 6)$. These techniques are $T(1, 6)$, $T(1, 7)$, $T(2, 7)$, $T(4, 2)$, $T(4, 3)$, $T(4, 5)$, $T(5, 0)$, $T(5, 2)$, and $T(5, 5)$. To the best of our knowledge, they are *previously unknown* in the literature. It indicates that when comparing to the state-of-the-art hybrid race detection technique, many alternative techniques are available for achieving similar overall performance (e.g., the first 3 techniques listed in this paragraph), or trading memory for time efficiency with either equal or higher detection effectiveness (e.g., the last 6 techniques listed in this paragraph).

V. Related Work

AccuLock [11] is the first epoch-based hybrid race detector, but AccuLock is incomplete and incurs the problem of reporting false ∅-race warnings under multiple (nested) locks. MultiLock-HB [11] is the first complete and sound epoch-based hybrid race detector aiming to address the limitations of AccuLock. However, it incurs significantly heavier runtime slowdown and memory overhead compared to AccuLock. For example, the experiment in [11] showed that MultiLock-HB incurred over 3x runtime slowdown and over 2x memory overhead than AccuLock.

HistLock [12] is a sound hybrid race detector. It employs the memory access skipping strategy based on lock release, as well as the history reduction strategy based on function call. HistLock extensively enlarges the race detection coverage of AccuLock and reduces both runtime slowdown and memory overhead compared to MultiLock-HB. Although it is incomplete under the notion used by MultiLock-HB, as shown in [12], HistLock achieved practically the same level of race detection effectiveness while halved both the runtime slowdown and the memory overhead of MultiLock-HB, which qualifies it as the state-of-the-art hybrid race detector.

VI. Conclusion

This paper has presented a framework called GBRAD with two decision points to choose different strategies for hybrid race detection. The framework includes 6 skipping strategies and 8 reduction strategies, from which 48 hybrid detection techniques can be initialized. We have performed a controlled experiment on the PARSEC benchmark suite as well as 4 real-world applications to evaluate these 48 techniques and their design strategies in terms of runtime slowdown, memory overhead, and race detection effectiveness. Finally, the experiment has identified 9 newly composed techniques comparable to the state-of-the-art.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (*PACT'08*), pp. 72–81, 2008.

[2] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'10*), pp. 330–340, 2010.

[3] Y. Cai and W. K. Chan. Lock trace reduction for multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), vol. 24(12), pp. 2407−2417, 2013.

[4] 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*), pp. 121–133, 2009.

[5] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21(7), pp. 558–565, 1978.

[6] 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'08*), pp. 329−339, 2008.

[7] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'03)*, pp. 167−178, 2003.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* (*TOCS*), vol. 15(4), pp. 391−411, 1997.

[9] L. J. Williams and H. Abdi. Fisher's least significance difference (LSD) test. *Encyclopedia of Research Design*, pp. 491–494, 2010.

[10] S. Wu, C. Yang, C. Jia, and W. K. Chan. ASP: Abstraction subspace partitioning for detection of atomicity violations with an empirical study. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), vol. 27(3), pp. 724–734, 2016.

[11] X. Xie, J. Xue, and J. Zhang. AccuLock: Accurate and efficient detection of data races. *Software: Practice and Experience*, vol. 43(5), pp. 543−576, 2013.

[12] J. Yang, C. Yang, and W. K. Chan. HistLock: Efficient and sound hybrid detection of hidden predictive data races with functional contexts. In *Proceedings of 2016 IEEE International Conference on Software Quality, Reliability and Security* (*QRS 2016*), pp. 31–24, 2016.

[13] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (*OOPSLA'12*), pp. 485–502, 2012.

[14] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (*ISCA'09*), pp. 325–336, 2009.

[15] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 35nd ACM/IEEE International Conference on Software Engineering* (*ICSE '13*), pp.192–201, 2013.