Experimental Performance Comparison of Dynamic Data Race Detection Techniques

Misun Yu, Seung-Min Park, Ingeol Chun, and Doo-Hwan Bae

Data races are one of the most difficult types of bugs in concurrent multithreaded systems. It requires significant time and cost to accurately detect bugs in complex large-scale programs. Although many race detection techniques have been proposed by various researchers, none of them are effective in all aspects. In this paper, we compare the performance of five recent dynamic race detection techniques: FastTrack, Acculock, Multilock-HB, SimpleLock+, and causally precedes (CP) detection. We experimentally demonstrate the strengths and weaknesses of these dynamic race detection techniques in terms of their detection capability, running time, and runtime overhead using 20 benchmark programs with different characteristics. The comparison results show that the detection capability of CP detection does not differ from that of FastTrack, and that SimpleLock+ generates the lowest overhead among the hybrid detection techniques (Acculock, SimpleLock+, and Multilock-HB) for all benchmark programs. SimpleLock+ is 1.2 times slower than FastTrack on average, but misses one true data race reported from Mutilock-HB on the large-scale benchmark programs.

Keywords: Data race, Dynamic detection, Multithreaded programming, Debugging, happens before, Lockset, Causally precedes.

Manuscript received Dec. 1, 2015; revised Oct. 4, 2016; accepted Nov. 15, 2016. This work was supported by Dual Use Technology Program through Civil Military Technology Cooperation Center funded by Ministry of Trade, Industry & Energy and Defense Acquisition Program Administration.

Misun Yu (corresponding author, msyu@etri.re.kr), Seung-Min Park (minpark@etri.re.kr), and Ingeol Chun (igchun@etri.re.kr) are with the SW & Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

Doo-Hwan Bae (bae@se.kaist.ac.kr) is with the Software engineering Laboratory, KAIST, Daejeon, Rep. of Korea.

This is an Open Access article distributed under the term of Korea Open Government License (KOGL) Type 4: Source Indiction + Commercial Use Prohibition + Change Prohibition (http://www.kogl.or.kr/news/dataFileDown.do?dataIdx=71&dataFileIdx=2).

I. Introduction

Multithreading is an essential programming technique commonly used in fields ranging from operating systems to mobile-multimedia applications for the purpose of improving the performance or responsiveness of a particular program. However, writing a correctly executing multithreaded program is far more difficult than writing a correctly executing sequential program owing to the nondeterminism in concurrent thread executions. Nondeterministic thread interleaving may produce nondeterministic outputs for the same input when the threads are not properly synchronized. When this nondeterministic thread behavior causes a system failure or incorrect results, it is referred to as a concurrency bug. Owing to the difficulty of detection, concurrency bugs may be unintentionally left in a program after release, which can have disastrous results [1], [2].

A data race is one type of concurrency bug that occurs when two different threads access the same memory location without an ordering constraint enforced between the accesses, where at least one access is a write access [3]. Because a data race can easily occur and is very common, a large number of studies [4]–[18] on dynamic detection techniques have been conducted to accurately detect as many as data races as possible by analyzing the execution trace of a program. Among the previous techniques, several [4], [5], [17] proposed fundamental algorithms for developing precise or high-coverage dynamic race detection techniques.

Based on these basic algorithms, some [6]–[16] proposed various techniques to increase the performance of the previous techniques. All of these dynamic detection techniques verified their superiority by showing the running time, runtime overheads, detection capability, and accuracy (ratio of false positives), which are the important factors for the practical use of a race detector. A low runtime slowdown frequently leads

 © 2017 ETRI

developers to use race detectors during the development phase of a concurrent program, which can increase the reliability of the program by rapidly removing problematic concurrency bugs. With the detection speed, the detection capability and accuracy should be considered to develop and select a race detection tool because a high detection capability implies a small number of repeated executions of a program for detection, and a high accuracy implies a low laborious effort to filter false warnings.

In this paper, we analyzed the performance of state-of-the-art and current dynamic techniques in terms of detection speed, capability, and accuracy on the same platform for a fair comparison. We collected the performance data on these techniques through extensive experiments using the same benchmark programs and configurations. Our work makes the following specific contributions:

- We select five pure dynamic data race detection techniques: FastTrack, Acculock, Multilock-HB, SimpleLock+, and Causally precedes (CP) detection. We evaluate their performance on the same platform with the same input.
- We compare the running time, runtime overhead, detection capability, and accuracy of all five detection techniques using 12 small programs containing various data races, and the most-used eight large-scale programs categorized into three groups with different numbers of access and synchronization events.
- Our experiments show that Multilock-HB and CP-detection can accurately and precisely detect actually occurred and potential data races. However, both methods require significant processing time. In addition, we showed that SimpleLock+ has almost the same detection capability and accuracy as Multilock-HB, and generates a detection speed similar to that of FastTrack.

The rest of this paper is organized as follows. Section II discusses previous studies in this area. In Section IV, we briefly review the dynamic data race detection techniques compared in this paper. In Section III, an analysis of these techniques based on the experimental results is presented. In Section V, we provide some concluding remarks regarding this research.

II. Related Work

There have been a number of studies on increasing the detection speed and capability of dynamic data race detection techniques with high accuracy, especially for Java programs. Here, we discuss the relevant research along two main axes: dynamic analysis and static analysis. For dynamic analysis techniques, we additionally describe their performance evaluation methods.

Dynamic analysis: Djit+ [4] was the first vector clock (VC)-based HB detector. To complement the high detection overhead of Djit+, sampling techniques and a combination of Djit+ and a Lockset algorithm [5] such as LiteRace [6], RaceTrack [7], or MultiRace [8] have been proposed. Although these complementary techniques reduced the detection speed and memory requirement, Djit+ provided the highest precision before the appearance of FastTrack.

FastTrack is the fastest race detection algorithm that reduces the overhead of the VC operations of Djit+ without sacrificing accuracy using an epoch-based representation. FastTrack showed its effectiveness by comparing the memory overhead and runtime slowdown of Djit+ using 16 benchmark programs including small-sized (86–111k lines of code (LOC)) programs collected from the SPEC JVM98 [19] and the parallel Java Grand benchmark suite [20]. After FastTrack was proposed, sampling techniques such as Pacer [10] CARISMA [11], and a dynamic granularity algorithm [12] were suggested to improve its detection speed by minimizing the number of missing data races. The detection capability, runtime overhead, and memory usage of CARISMA was compared with those of FastTrack on the same RoadRunner framework [21] using the subset of SPEC JVM 98 and Dacapo benchmarks [22]. Pacer was implemented on the Jikes RVM and compared the detection capability and runtime overhead to LiteRace using the DaCapo benchmark suite and SPEC JBB2000.

A shortcoming of the HB detection technique is that it cannot detect hidden data races that are detectable in other thread interleaving sequences. To report hidden races as well as actually occurred data races, hybrid detection was introduced. This [13] is the first hybrid detection method that combines HB-relation and Lockset-based detection algorithms. Acculock and Multilock-HB [15] are epoch-based hybrid detectors based on Hybrid. Acculock provides a fast detection speed comparable to that of FastTrack, but may introduce additional false positives. Multilock-HB accurately detects actually occurred and potential data races; however, it incurs a large amount of computational runtime overhead. The latest study on hybrid detection [16] provides the fastest hybrid detection technique that may miss a data race when both accesses of the data race are protected by different locks. This study compared the detection capability and runtime slowdown among FastTrack, Multilock-HB, and SimpleLock+ on the same RoadRunner framework using the Dacapo and Grand benchmark suites.

To relax the sensitivity to thread interleaving while maintaining the precision of HB detection, CP detection based on a CP relation [17] was recently proposed. The authors of [17] demonstrated that CP can detect more data races than HB detection, and compared the detection speed to HB detection using small-sized (86–49k LOC) benchmark programs.

Maximal sound predictive detection (RVPredict) [18] is a dynamic detection technique that uses branch information from the source code of a program. It detects data races by formulating race detection as a constraint-solving problem. The authors of [18] demonstrated the detection capability and scalability of RVPredict, CP, HB detection, and Said and others [23] using the small-sized IBM Contest benchmark suite and the parallel Java Grand benchmark programs (with a reduced data set). All algorithms were implemented in RVPredict.

In this paper, we compare the detection speed, capability, and accuracy of the current pure dynamic data race detection techniques (FastTrack, Acculock, Multilock-HB, SimpleLock+, and CP) using the most-used large-scale benchmark suites without modification and small benchmark programs on the same platform.

Static analysis: To statically prevent data races, Flanagan and Freund [24] proposed a type-checking system based on previous research that describes a race-free type system for a concurrent object calculus [25]. This approach is extended by [26] and [27] using the concept of ownership. A type and effect system [28]–[30] provides deterministic semantics for object-oriented languages. These type systems have good scalability but require user annotations.

Detectors based on data flow analysis can report potential data races but produce many false alarms and are difficult to scale to large programs. The object use graph-based technique [31] reports object races using an approximation of the HB relation of accesses to an object by different threads. Chord [32] improved the precision of lockset computation using k-object context sensitivity. Follow-up research [33] refined this study using a *conditional must not aliasing* property to reduce false positives.

Recent static detectors specialized in specific kinds of program structures or data races to improve precision and scalability. IteRace [34] is a set of three techniques that are specialized to the parallel loops for collections that are introduced in Java 8. CTADetector [35] uses static analysis to detect a misused CHECK-THEN-ACT idiom that is a composition of two operations where a check on the concurrent collection precedes an action.

III. Background

Dynamic data race detection algorithms can be categorized into Lockset, HB, and hybrid detection algorithms. Among them, Lockset algorithms are not used alone because of their high false-positive rate. Therefore, we exclude Lockset algorithms from our comparison.

FastTrack is a state-of-the-art HB detection technique. Although many sampling and vector clock-sharing techniques [10], [11], [12], [36] have been proposed, there is no HB algorithm better than FastTrack at this writing. On the other hand, various hybrid detection techniques have been continuously suggested. We selected three current techniques with different characteristics in terms of accuracy and runtime overhead: Acculock, Multilock-HB, and SimpleLock+.

CP detection is the latest dynamic detection method used to relax scheduling sensitivity and remove the possibility of false positives that may be generated by hybrid detection techniques. For a precise comparison of detection overhead and detection capability, we include CP detection in our comparison, although it has yet to be commonly used for program testing.

1. FastTrack

FastTrack reduces the complexity of most VC comparisons ($\sqsubseteq$) from $O(n)$ to $O(l)$ by introducing an epoch-VC comparison ($\preccurlyeq$). Herein, an epoch includes only a clock and a thread id (tid). The epoch is used to record the last write and read of the totally ordered writes and reads. Algorithm 1 shows the FastTrack algorithm. FastTrack maintains vector clocks C_t and L_m for each thread t and lock m. The clock entry $C_t(u)$ records the clock for the last event of thread u that happens before the current event of thread t. In Algorithm 1, $E(t)$ returns the current epoch of thread t. The clock-update algorithm of FastTrack for the lock acquire and release, as well as for explicit synchronizations (fork and join), is the same as in previous VC-based race detectors including Djit+.

Algorithm 1. FastTrack

Lock acquire: *t* acquires a lock *m*

1: $C_t \leftarrow C_t \sqcup L_m$;

Lock release: *t* releases a lock *m*

2: $L_m \leftarrow C_t$;

Fork and Join: *t* creates *u*; *t* blocks until *u* terminates

3: $C_t \leftarrow C_t \sqcup L_m$;

4: $C_t[t] \leftarrow C_t[t] + 1$;

Read: *t* reads from *x*

5: if $(R_x = E(t))$ return;

6: if $(W_x \npreccurlyeq C_t)$ report a warning; //write-read race

7: if $(|R_x| = 1 \wedge R_x \preccurlyeq C_t)$

8: $R_x \leftarrow E(t)$;

9: else

10: $R_x[t] \leftarrow C_t[t]$;

Write: *t* writes to *x*

11: if $W_x = E(t)$ return;

12: if $(W_x \npreccurlyeq C_t)$ report a warning; // write-write race

13: if $|(R_x| \leq 1)$ {

14: if $(R_x \npreccurlyeq C_t)$ report a warning; // read-write race

15: } else {

https://doi.org/10.4218/etrij.17.0115.1027

```
16:        if (R_x ⊈ C_t) report a warning;     //read-write race
17:    }
18:    R_x ← ∅ ;
19:    W_x ← E(t);
```

FastTrack reports a write-read race if the last write access to shared memory location x does not occur before the current read access to x (line 6 of Algorithm 1). In addition, it reports a read-write race if the last read of totally ordered read access to x does not occur before the current write access to x (line 14 of Algorithm 1). In another case, FastTrack reports a read-write race if the VCs of the last read access and current write are not partially ordered (line 16 of Algorithm 1).

2. Hybrid Detection

Hybrid detection that combines HB and Lockset detection, and has its origins in Hybrid [10], aims to increase the detection capability by weakening the sensitivity to a thread interleaving sequence of HB detection. Hybrid detectors can report potential data races in a single execution trace. To detect additional potential data races, hybrid detectors monitor the sets of locks (locksets) that are protecting the shared memory accesses. In addition, they do not conduct vector clock operations for lock acquire and release events because the HB relation between two accesses of different threads owing to the lock acquire-release can be changed in other thread interleaving sequences. That is, a relaxed HB relation is used, which is held when two different accesses of different threads are ordered through explicit synchronizations such as a fork and join, with the exception of lock operations.

For hybrid detection, Lockset is applied for two accesses that are not ordered through explicit synchronizations. The underlying idea of Lockset is that, to prevent data races, all accesses of multiple threads to a shared memory location must be protected through the same lock. Algorithm 2 shows the basic algorithm of Lockset.

Algorithm 2. Lockset

```
1: For each x, initialize L_x to the set of all locks;
2: On each access to x by thread t
3: L_x ← L_x ∩ L_t;
4: if (L_x = ∅ ) report a warning;
```

Hybrid detection requires a high runtime overhead to accurately detect data races because it maintains the access history and the related information (the set of locks and clocks) for each access.

A. Acculock and Multilock-HB

Acculock first introduced an epoch representation, which was presented by FastTrack, into hybrid detection. In addition, Acculock only keeps a lockset that protects the last read access to shared memory location x for each thread. In addition, it maintains the intersection of the two locksets protecting the last and the previous write accesses to x when these two accesses are not ordered through explicit synchronizations. That is, Acculock keeps only one lockset and epoch for write accesses to x regardless of the number of threads. Therefore, Acculock has a low memory requirement and a fast detection speed comparable to that of FastTrack. However, Acculock generates additional false positives when nested locks are used because it must maintain the subset of the access information. Acculock provides $O(n\log l)$ complexity for the number of elements l in a lockset.

Multilock-HB was introduced to remove the false positives of Acculock. To remove the false positives, Multilock-HB maintains all read and write access histories for each thread. Each item of a read and write access history includes a lockset and a clock for each read and write access. To avoid duplicated race warnings to the same shared memory location and to reduce the runtime overhead, Multilock-HB provides an optimization technique but requires a large amount of memory and time for large-scale real-world programs. The complexity of Multilock-HB is $O(nm\log l)$ for the number of threads n, the length of access events m, and the number of elements l in a lockset.

B. SimpleLock+

SimpleLock+ was proposed to improve the performance of the previous accurate hybrid detector. SimpleLock+ improves the performance based on two assumptions: (1) most data races are caused by accesses without any lock protection to a shared memory location, and (2) the distance between two accesses that cause a data race is not long.

Based on the first assumption, SimpleLock+ only reports a data race when at least one access of a race is not protected through any locks (zero-locked access). This technique replaces the set intersection operations of the previous hybrid detection into Boolean operations that verify the existence of locks protecting the accesses. Based on the second assumption, SimpleLock+ maintains information on the reads and writes after the last explicit synchronization of all threads for each shared memory location. This information includes two clocks for a read and write, and two flags indicating whether a read or write not protected by any lock during the clocks occurs.

Owing to the above two improvement techniques, SimpleLock+ provides $O(n)$ complexity for the number of threads n, and generates a fast detection speed comparable to that of FastTrack. However, SimpleLock+ may miss data races if two accesses of different threads that are not ordered by

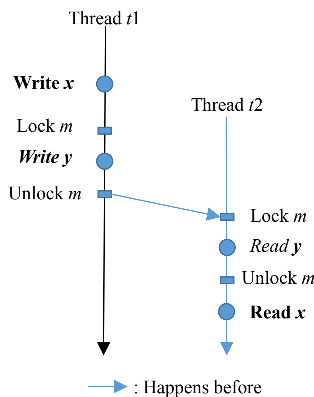

Fig. 1. Example of a predictable data race.

explicit synchronizations are protected through different locks.

3. CP Detection

CP detection reports a data race by checking the CP relation between two accesses to a shared memory location. A CP relation is a new relation proposed in 2012 that generalizes an HB relation to detect more data races without introducing false positives or false negatives. CP detectors can detect potential data races as well as actually occurred data races in one execution trace. Although detected data races are a subset of the results from hybrid detectors, CP detectors do not report false warnings that hybrid detectors may generate owing to the algorithmic limitations of hybrid detection.

A CP relation is a subset of an HB relation, and can partially detect potential data races that may be missed by HB detectors. Figure 1 shows an example of these data races.

In Fig. 1, HB detectors cannot detect a data race on x because a lock release of $t1$ happens before the lock acquisition of $t2$, masking the lack of synchronization between the write of $t1$ and the read of $t1$ on x. However, CP detection can predict this data race based on the definition of the CP ($\ll_{CP}$) relation as follows:

- $\ll_{CP}$ has a release-acquire edge between critical sections over the same lock that contains conflicting events. Herein, two events by different threads conflict if they access the same shared memory location and one of them is a write.
- $\ll_{CP}$ has a release-acquire edge between critical sections over the same lock that contains CP-ordered events.
- CP is closed under left and right compositions with HB.

IV. Performance Comparison

We compared the performance of five data race detection techniques: FastTrack (FT), Acculock (AC), Multilock-HB (ML), SimpleLock+ (SL+), and CP detection (CP). The evaluation was conducted on the same platform using 21 Java benchmark programs. Time, detection capability, and accuracy were considered.

1. Implementation

The race detection techniques being evaluated are categorized into two types: online (runtime) and offline detection. FT, AC, ML, and SL+ are VC-based online detection techniques. We implemented four online detection techniques in the RoadRunner framework as independent tools. RoadRunner is a dynamic analysis framework for Java programs, which provides event handlers for memory and synchronization operations that are executed during program execution. These event handlers can be overridden to implement user-defined data race detectors. RoadRunner relies on the just-in-time compiler to optimize the instrumentation code and tool dispatches. For the offline detection technique (CP), we borrowed the implementation from [18] for the most efficient implementation. CP was implemented in RVPredict, which is available at *http://fsl.cs.illinois.edu/rvpredict/*. RVPredict first stores the execution traces (including shared data accesses and synchronization events) into a database, and then conducts a predictive analysis based on these traces.

2. Methodology

A. Platform

The evaluation was conducted on a machine with a 3.40 GHz Intel Core i7-3770K (quad core) CPU and 32 GB of RAM running 64-bit Ubuntu 12.4 OS. The SSD free space was 10 GB, which was used for recording the execution traces of CP. We installed H2 database engine 1.4 to manage the execution traces on our SSD.

B. Benchmark Configuration

We conducted out experiments on 21 different benchmark programs, which were classified into two groups. The first group includes small example programs that were used in the previous work [18]: *critical*, *airline*, *account*, *pingpong*, *bbuffer*, *bubblesort*, *bufwriter*, *mergesort*, *raytracer-s*, *montecarlo-s*, and *moldyn-s*. In this first group, *raytracer-s*, *montecarlo-s*, and *moldyn-s* are modifications of *raytracer*, *montecarlo*, and *moldyn* from the parallel Java Grand benchmark suite, respectively, and shrink the internal data size to reduce the program execution time.

The second group includes large-scale real-world applications that were selected from the parallel Java Grand benchmark suite and Dacapo benchmark suite (9.12 bach), which were generally used for comparing the performance of race detection algorithms in previous research [9], [10], [14], [15], [16], [36], [37]. We

Table 1. Benchmark descriptions.

Program	Description
avrora	Simulation tools for programs on a grid of AVR microcontrollers
luindex	Program that uses lucene to index a set of documents (the works of Shakespeare and the King James Bible)
lusearch	Program using lucene to do a text search by keywords over a corpus of data that comprises the works of Shakespeare and the King James Bible
sunflow	Renderer that processes a set of images using a ray tracing algorithm
jython	Python interpreter written in Java
moldyn	Molecular dynamics simulation program
raytracer	3D ray tracing program
montecarlo	Financial simulation program using Monte Carlo techniques

selected programs that are executed on the RoadRunner framework from the Dacapo benchmark suite. Table 1 lists brief descriptions of these large-scale benchmark programs.

We configured *moldyn*, *raytracer*, and *montecarlo* to create four worker threads, used BenchSizeA as a dataset, and used the default settings for all other benchmark programs. Detailed characteristics such as the lines of code, the number of threads, and the number of memory accesses are listed in Tables 2 and 3. Runtime detection techniques track all accesses to the elements of shared arrays only for small programs. For reasons of efficiency, we did not track each array element for large-scale programs.

C. Measuring Slowdowns

We measured the running time and slowdowns of the target techniques. The running time of a detection technique is the total detection time, which includes the instrumentation time and execution time of the detection algorithm. A slowdown of a detection technique is the ratio of the instrumented running time to the original running time of the benchmark program. For FT, AC, ML, and SL+, we measured the runtime instrumentation and detection time on the RoadRunner framework. For CP, we summed the instrumentation, logging, and offline detection times. We executed and measured each detection technique 10 times using the same input, and averaged the results.

D. Counting Race Warnings

Similar to the measurement of the slowdown of each detection technique, we conducted 10 executions for each technique using the same input. We counted only the number of warnings for distinct shared variables during the 10 executions.

With the total number of reported data races, we also specified the number of true data races only when the hybrid detectors reported more races than FT. For the 12 small benchmark programs, we verified the true data races by analyzing the source code and execution trace manually because these programs are simple enough for manual analysis. For the eight large-scale programs, we randomly and heuristically inserted noises by calling a Java *sleep ()* method before the accesses to shared variables by threads because we could not find a systematic method or an automatic tool to precisely verify a data race in these large-scale programs.

The insertion of the *sleep ()* method can allow for variety in the thread interleaving sequences of a program. Then, we analyzed the executions using FT to precisely check whether the shared variables cause data races. We considered only shared variables that were reported by the hybrid detectors. When we could not expose a data race to a shared variable, we decided that the warning for the shared variable was not a true positive.

3. Results and Analysis

Table 2 lists the running times and the number (total warnings and true positives) of data races for the five dynamic detection techniques studied (FT, AC, SL+, ML, and CP) on small benchmark programs. We measured the total running time instead of the runtime overhead because CP is an offline detection technique based on the analysis of the execution-trace information recorded in a database. In addition, Table 3 lists the runtime overheads and the number (total warnings and true positives) of data races of the online detection techniques (FT, AC, SL+, and ML) on large-scale benchmark programs. We could not run CP on the large-scale benchmark programs owing to a lack of disk space required to record the execution-trace information generated by CP. A general computer that is used by a developer has difficulty in handling the excessive amount of execution traces, which exceeds dozens of gigabytes using CP because of its significant processing and memory requirements.

A. Overall Results

The CP was shown to be about 8.2 times slower than FastTrack, but the detection capability was the same for all small benchmark programs *array*, *critical*, *airline*, *account*, *pingpong*, *bbuffer*, *bubblesort*, *bufwriter*, *mergesort*, *raytracer-s*, *montecarlo-s*, and *moldyn-s*. Based on this fact, we can infer that the types of potential data races that CP can detect, as shown in Fig. 1, are not general.

All hybrid detectors (SL+, AC, and ML) reported more data races than FastTrack, and SL+ generated the lowest amount

Table 2. Running time and number of data races of five dynamic detection techniques on small benchmark programs.

Program	LoC	Trace			Running time (s)					Data race				
		Thread	Access	Ex-Synch +Volatile	FT	AC	SL+	ML	CP	FT	AC (TP)	SL+ (TP)	ML (TP)	CP
avrora	40	2	11	0	0.3	0.4	0.4	0.4	2.0	0	1(1)	1(1)	1(1)	0
critical	63	3	20	1	0.3	0.3	0.3	0.3	2.2	1	1	1	1	1
airline	83	11	127	0	0.4	0.5	0.5	0.3	2.4	1	1	1	1	1
account	87	3	83	1	0.8	0.8	0.8	0.8	2.3	2	1	2	2	2
pingpong	124	18	116	3	0.3	0.3	0.3	0.3	2.4	2	2	2	2	2
bbuffer	334	4	1K	48	0.4	0.5	0.5	0.6	3.7	2	2	2	2	2
bubblesort	274	26	6.5k	0	0.6	0.6	0.5	1.1	3.2	2	2	2	2	2
bufwriter	199	5	6.5k	4	0.3	0.3	0.3	0.3	2.7	2	4(4)	4(4)	4(4)	2
mergesort	298	5	0.7k	2	0.4	0.4	0.3	0.4	3.2	1	1	0	1	1
raytracer-s	2.0k	4	28.2k	4.5k	0.5	0.6	0.5	0.7	5.3	3	3	2	3	3
montecarlo-s	3.6k	4	6.3M	3	0.5	0.4	0.4	0.4	15.6	0	0	0	0	0
moldyn-s	1.4k	4	259k	29.2k	1.5	2.2	0.9	6.7	6.4	2	2	2	2	2
Total	8.5k		6.6M	33.7k	6.3	7.3	5.7	11.3	51.4	18	21(21)	19(19)	21(21)	18

Ex-Synch: explicit synchronization, volatile: volatile-variable access, TP: true positive

Table 3. Runtime slowdowns and number of data races of four online detection techniques on large-scale benchmark programs.

Program	Trace			Slowdown				Data race			
	Threads	Access (M)	Ex-Synch +Volatile (K)	FT	AC	SL+	ML	FT	AC (TP)	SL+ (TP)	ML (TP)
avrora	7	889.4	577.0	5.5	7.2	6.6	855.7	3	4(3)	4(3)	4(3)
luindex	2	273.5	1.7	8.6	13.9	12.1	20.4	1	1	1	1
lusearch	10	497.8	1,153.3	9.0	12.6	11.2	549.8	0	2(0)	2(0)	2(0)
sunflow	17	3,396.3	0.0	43.3	71.9	66.3	120.2	5	31(31)	31(31)	31(31)
jython	2	638.8	4,594.1	7.7	9.8	10.9	1,228.4	21	22(21)	22(21)	22(21)
raytracer	4	1,521.3	0.0	95.0	135.8	90.7	281.8	1	1	0	1
montecarlo	4	166.9	0.0	9.1	10.6	9.3	12.8	1	1	1	1
moldyn	4	598.4	1.2	57.5	85.8	68.5	159.8	0	0	0	0
Total		7982.3	6,327.3					32	62(58)	61(57)	62(58)
Average				29.5	43.5	34.5	403.6				

Ex-Synch: explicit synchronization, volatile: volatile-variable access, TP: true positive

of overhead for all benchmark programs. The overhead of SL+ was 1.2 times that of FastTrack on our large-scale benchmark programs, but SL+ missed one true data race that ML reported on *raytracer*. ML provides the highest detection capability without the possibility of false positives that may be introduced by AC, and detected 29 more true data races than FT on our benchmark programs. However, the runtime overhead of ML is 13.7 times that of FT on large-scale benchmark programs, and reaches 159.5 times on *jython*.

B. Running Time and Runtime Overhead

Figure 3 illustrates the running time of the five dynamic detection techniques, which are listed in Table 2. As shown in Fig. 2, the CP is about 9.3-times slower than FastTrack for the 12 small benchmark programs *array*, *critical*, *airline*, *account*, *pingpong*, *bbuffer*, *bubblesort*, *bufwriter*, *mergesort*, *raytracer-s*, *montecarlo-s*, and *moldyn-s*. In particular, CP requires a long detection time for long-running programs that generate many access events (*montecarlo-s*), as shown in Fig. 3.

With the exception of CP, the online detection techniques show a similar running time on the 12 small benchmark programs. ML shows a particularly high runtime overhead for *bubblesort* and *montecarlo-s*, which have many threads, and generates many explicit synchronization and volatile-variable

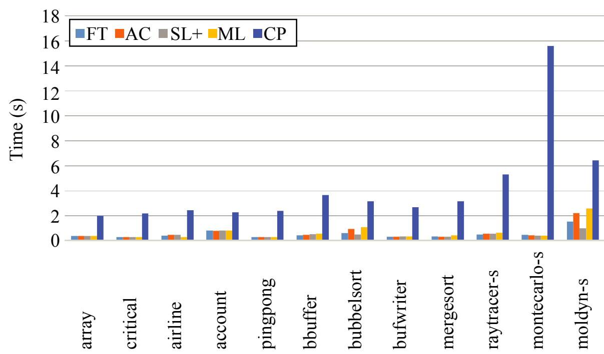

Fig. 2. Running time of five dynamic detection techniques on small benchmark programs.

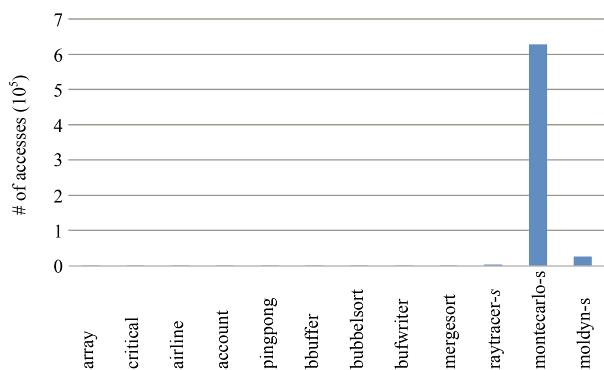

Fig. 3. Number of shared memory accesses.

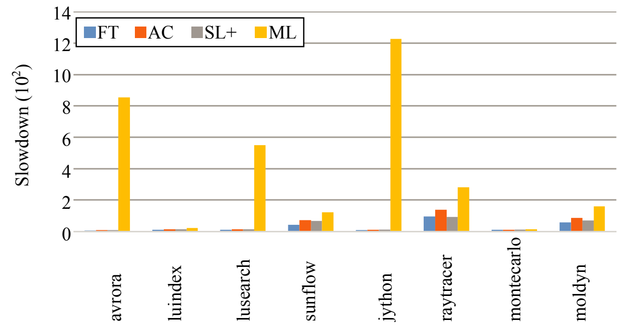

Fig. 4. Runtime overhead of online dynamic detection techniques on large-scale benchmark programs.

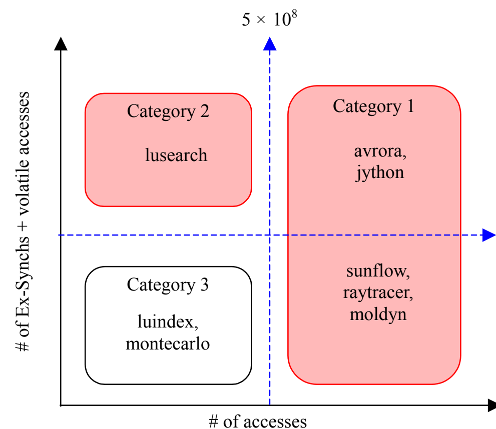

Fig. 5. Classification of large-scale benchmark programs.

access events.

SL+ shows the shortest total running time among all four techniques. The total running time of SL+ is 0.91 times that of FT. In particular, SL+ provides a shorter running time on programs that have many threads, or that generate a lot of explicit and volatile-variable access events, such as *bubblesort*, *raytracer-s*, and *moldyn-s*.

To analyze the performance correlation between the types of events (memory access and synchronization events) and the online detection techniques, we measured the runtime overhead of these techniques on large-scale benchmark programs. Because of the very high overhead of ML on *jython*, we limited the length of history that ML must maintain for each thread to 3,000.

Figure 4 illustrates the runtime overhead of the online detection techniques listed in Table 3. To easily distinguish the program groups that incur high detection overhead, we classified the benchmark programs into three categories, as shown in Fig. 5. Categories 1 is the program group of long-running programs that generate many shared memory accesses. Category 2 generates many synchronizations (explicit synchronizations and volatile-variable access) but generates a smaller number of shared memory accesses than Category 1. Category 3 includes programs that generate a relatively small number of shared memory accesses and synchronizations.

As shown in Fig. 4, ML generates the highest runtime overhead for all of the large-scale benchmark programs. In particular, ML generates very high overhead (532.6 times slower than the original program on average) for Category 1 and Category 2. The overhead of ML on *jython* is more than 1,228.4 times, which is 159.5 times that of FT. FT generates the lowest amount of runtime overhead, and SL+ and AC are 1.2 times and 1.5 times slower than FT, respectively. In addition, FT, SL+, and AC generate slightly more overhead for Category 1 programs, which are long-running programs.

As a result, FT shows the smallest amount of overhead, and SL+ was shown to be the fastest among the four other dynamic detection techniques studied.

C. Detection Capability

The detection capability of CP detection is the same as that for FastTrack in all of the small benchmark programs. On the small benchmark programs, all of the hybrid detection techniques (AC, SL+, and ML) additionally detected two

```
1: package org.sunflow.core.accel;
2: public class KDTree implements AccelerationStructure {
3:     ..
4:     public void intersect(Ray r, IntersectionState state) {
5:         ..
6:         sleep(40);/*delaying thread executions*/
7:         primitiveList.intersectPrimitive(r, primitives [offset], state);
8:         ..
9:     }
10:    ..
11:}
```

Fig. 6. Delay insertion in *sunflow*.

potential data races in *bufwriter*, which were missed by FT. In addition, SL+ missed two potential data races in *mergesort* and *raytracer-s*, which were detected by AC and ML. We verified that all these potential data races are true data races that can be exposed in other thread interleaving sequences.

For the large-scale benchmark programs, the hybrid detectors reported more data races than FastTrack. Among the hybrid detectors, AC and ML reported the same data races, and SL+ missed one in *raytracer*, which is identical to that missed by SL+ in *raytracer-s*. Although AC detected the same data races as ML on our benchmark programs, AC has the possibility of additional false positives, which were not reported by SL+ and ML, as described in Section II.

Except for *raytracer*, hybrid detectors reported 30 more data races than FT on large-scale benchmark programs. Among these data races, we verified that 26 data races in *sunflow* are real data races, which can be exposed by delaying thread accesses to a shared variable during program execution using the Java *sleep()* method, as shown in Fig. 6.

Figure 6 shows the part of the *sunflow* source code that we used to verify the data races that are reported from hybrid data races. We detected 31 data races using FT by inserting "*sleep (40)*" into delaying thread access to a shared memory location to which *primitiveList* is pointing.

V. Conclusion

We presented the performance comparison results of five recent dynamic data race detection techniques: FastTrack, Acculock, SimpleLock+, Multilock-HB, and CP detection. We conducted experiments on the same platform using 12 small and 8 large-scale benchmark programs. The comparison results show that CP detection has the highest amount of overhead among the five detection techniques, although the detection capability of CP detection does not surpass that of FastTrack on our benchmarks. SimpleLock+ generated the lowest amount of runtime overhead among the hybrid detection techniques (Acculock, SimpleLock+, and Multilock-HB) on all benchmarks, which is 1.2 times that of FastTrack on average. Unlike Acculock, SimpleLock+ does not introduce additional false positives into Multilock-HB, but misses 3.6% of the true data races reported by Mutilock_HB. Therefore, SimpleLock+ can be a good option for the frequent detection of data races during the development process, and MultiLock-HB can be used for the late stages of development for a more thorough check. We believe that our performance comparison in various aspects of the program characteristics can provide useful information for further research to improve the efficiency of current data race detection techniques.

References

[1] N.G. Leveson and C.S. Turner, "An Investigation of the Therac-25 Accidents," *Comput.*, vol. 26, no. 7, July 1993, pp. 18–41.

[2] K. Poulsen, *Software Bug Contributed to Blackout*, Accessed Dec. 30, 2015. http://www.securityfocus.com/news/8016

[3] R.H.B. Netzer and B.P. Miller, "What Are Race Conditions?: Some Issues and Formalizations," *Lett. Programming Languages Syst.*, vol. 1, no. 1, Mar. 1992, pp. 74–88.

[4] E. Pozniansky and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs," *Symp. Principles Practice Parallel Programming*, San Diego, CA, USA, June 11–13, 2003, pp. 179–190.

[5] S. Savage et al., "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *Symp. Operating Syst. Principles*, Saint-Malo, France, Oct. 5–8, 1997, pp. 27–37.

[6] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-Race Detection," *Conf. Programming Language Des. Implementation*, Dublin, Ireland, June 15, 2009, pp. 34–143.

[7] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions Via Adaptive Tracking," *Symp. Operating Syst. Principles*, Brighton, UK, Oct, 23–26, 2005, pp. 221–234.

[8] E. Pozniansky and A. Schuster, "MultiRace: Efficient On the Fly Data Race Detection in Multithreaded C++ Programs," *Concurrency Comput.: Practice Experience*, vol. 19, no. 3, Mar. 2007, pp. 327–340.

[9] C. Flanagan and S.N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," *Conf. Programming Language Des. Implementation*, Dublin, Ireland, June 15–21, 2009, pp. 121–133.

[10] M.D. Bond, K.E. Coons, and K.S. McKinley, "PACER: Proportional Detection of Data Races," *Conf. Programming Language Des. Implementation*, Toronto, Canada, June 5–10, 2010, pp. 255–268.

[11] K. Zhai et al., "CARISMA: A Context-Sensitive Approach to Race-Condition Sample-Instance Selection for Multithreaded Applications," *Int. Symp. Softw. Testing Anal.*, Minneapolis, MN, USA, July 15–20, 2012, pp. 221–231.

[12] Y.W. *Song* and Y.H. Lee, "Efficient Data Race Detection for

https://doi.org/10.4218/etrij.17.0115.1027

C/C++ Programs Using Dynamic Granularity," *Int. Parallel Distrib. Process. Symp.*, Phoenix, AZ, USA, May 19–23, 2014, pp. 679–688.

[13] R. O'Callahan and J.D. Choi, "Hybrid Dynamic Data Race Detection," *Symp. Principles Practice Parallel Programming*, San Diego, CA, USA, June 11–13, 2003, pp. 167–178.

[14] X. Xie and J. Xue, "Acculock: Accurate and Efficient Detection of Data Races," *Symp. Code Generation Optimization*, Chamonix, France, Apr. 2–6, 2011, pp. 201–212.

[15] X. Xie, J. Xue, and J. Zhang, "Acculock: Accurate and Efficient Detection of Data Races," *Softw. Practice Experience*, vol. 43, no. 5, May 2013, pp. 543–576.

[16] M.S. Yu and D.H. Bae, "SimpleLock+: Fast and Accurate Hybrid Data Race Detection," *Comput. J.*, vol. 59, no. 6, 2016, pp. 793–809.

[17] Y. Smaragdakis et al., "Sound Predictive Race Detection in Polynomial Time," *Symp. Principles Programming Languages*, Philadelphia, PA, USA, Jan. 25–27, 2012, pp. 387–400.

[18] J. Huang, P.O. Meredith, and G. Rosu, "Maximal Sound Predictive Race Detection with Control Flow Abstraction," *Conf. Programming Language Des. Implementation*, Edinburgh, Ireland, June 9–11, 2014, pp. 337–348.

[19] SPEC JVM98 benchmarks, Accessed Dec. 30, 2015. http://www.spec.org/osg/jvm98/

[20] L.A. Smith, J.M. Bull, and J. Obdrizalek, "A Parallel Java Grande Benchmark Suite," *Conf. Supercomput.*, Denver, CO, USA, Nov. 10–16, 2001, p. 8.

[21] C. Flanagan and S.N. Freund, "The RoadRunner Dynamic Analysis Framework for Concurrent Programs," Workshop *Program Anal. Softw. Tools Eng.*, Toronto, Canada, June 5–6, 2010, pp. 1–8.

[22] S.M. Blackburn et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," *Conf. Object-Oriented Programming Syst., Languages, Appli.*, Portland, OR, USA, Oct. 22–26, 2006, pp. 169–190.

[23] M. Said et al., "Generating Data Race Witnesses by an SMT-Based Analysis," *Int. Conf. NASA Formal Methods Symp.*, Pasadena, CA, USA, Apr. 18–20, 2011, pp. 313–327.

[24] C. Flanagan and S.N. Freund, "Type-Based Race Detection for Java," *Conf. Programming Language Des. Implementation*, Vancouver, Canada, June 18–21, 2000, pp. 219–232.

[25] C. Flanagan and M. Abadi, "Object Types Against Races," *Conf. Concurrency Theory*, Eindhoven, Netherlands, Aug. 24–27, 1999, pp. 288–303.

[26] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," C*onf. Object-Oriented Programming, Syst., Languages, Appli.*, Seattle, WA, USA, Nov. 4–8, 2002, pp. 211–230.

[27] C. Boyapati and M. Rinard, "A Parameterized Type System for Race-Free Java Programs," C*onf. Object-Oriented Programming, Syst. Languages, Appli*, Tampa Bay, FL, USA, Oct. 14–18, 2001, pp. 56–69.

[28] R.L. Bocchino et al., "A Type and Effect System for Deterministic Parallel Java," C*onf. Object Oriented Programming Syst. Languages Appli.*, Orlando, FL, USA, Oct. 25–29, 2009, pp. 97–116.

[29] A. Greenhouse and J. Boyland, "An Object-Oriented Effects System," *European Conf. Object-Oriented Programming*, Lisbon, Portugal, June 14–18, 1999, pp. 205–229.

[30] J. Boyland, "The Interdependence of Effects and Uniqueness," *Workshop Formal Techs. Java Programs*, Budapest, Hungary, June 18, 2001.

[31] C. Von Praun and T.R. Gross, "Static Conflict Analysis for Multi-threaded Object-Oriented Programs," *Conf. Programming Language Des. Implementation*, San Diego, CA, USA, June 8–11, 2003, pp. 115–128.

[32] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," *Conf. Programming Language Des. Implementation*, Ottawa, Canada, June 10–16, 2006, pp. 308–319.

[33] M. Naik and A. Aiken, "Conditional Must Not Aliasing for Static Race Detection," *Symp. Principles Programming Languages*, Nice, France, Jan. 17–19, 2007, pp. 327–338.

[34] C. Radoi and D. Dig, "Practical Static Race Detection for Java Parallel Loops," *Int. Symp. Softw. Testing Anal.*, Lugano, Switzerland, July 15–20, 2013, pp. 178–190.

[35] Y. Lin and D. Dig, "A Study and Toolkit of CHECK-THEN-ACT Idioms of Java Concurrent Collections," *Softw. Testing, Verification Rel.*, vol. 25, no. 4, June 2015, pp. 397–425.

[36] J. Wilcox et al., "Array Shadow State Compression for Precise Dynamic Race Detection," *Conf. Automated Softw. Eng.*, Lincoln, NE, USA, Nov. 9–13, 2015, pp. 155−165.

[37] B.P. Wood, L. Ceze, and D. Grossman, "Low-Level Detection of Language-Level Data Races with LARD," *Conf. Archit. Support Programming Languages Operating Syst.*, Salt Lake, Canada, Mar. 1–5, 2014, pp. 671–686.

Misun Yu is a senior researcher at the Embedded Software Research Department at ETRI, Daejeon, Rep. of Korea. She received her MS degree from the Department of Computer Science and Engineering at Pohang University of Science and Technology, Rep. of Korea. Her main research interests include concurrent program analysis, software testing, and cyber-physical systems.

Seung-Min Park is a principal member of the engineering staff in Embedded Software Research Department at the ETRI. He received his MS degree from Hongik University, Seoul, Rep. of Korea in 1983. His research interests include embedded software, cyber-physical Systems, autonomic computing, and live-virtual-constructive technologies.

Ingeol Chun is the director of the CPS research section at ETRI and an adjunct professor at the University of Science & Technology, Daejeon, Rep. of Korea. He received his PhD and MS degrees in Electrical and Computer Engineering from Sungkyunkwan University, Suwon, Rep. of Korea. His research interests are cyber-physical systems, smart factories, autonomic computing systems, embedded systems, and software engineering.

Doo-Hwan Bae is a professor in the School of Computing at the Korea Advanced Institute of Science and Technology, Daejeon, Rep. of Korea. He received his PhD from the Department of Computer Science at the University of Florida Gainesville, USA. He currently leads many projects funded by the Korean government and industry. His research interests include software safety, software testing, quality-driven software development, embedded software design, and mining software repositories.