Postprint of article in *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (*ISSTA 2012*), ACM, New York, NY (2012)

CARISMA: a Context-sensitive Approach to Race-condition sample-Instance Selection for Multithreaded Applications

Ke Zhai
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
kzhai@cs.hku.hk

Boni Xu
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
bnxu@cs.hku.hk

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

T. H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

ABSTRACT
Dynamic race detectors can explore multiple thread schedules of a multithreaded program over the same input to detect data races. Although existing sampling-based precise race detectors reduce overheads effectively so that lightweight precise race detection can be performed in testing or post-deployment environments, they are ineffective in detecting races if the sampling rates are low. This paper presents CARISMA to address this problem. CARISMA exploits the insight that along an execution trace, a program may potentially handle many accesses to the memory locations created at the same site for similar purposes. Iterating over multiple execution trials of the same input, CARISMA estimates and distributes the sampling budgets among such location creation sites, and probabilistically collects a fraction of all accesses to the memory locations associated with such sites for subsequent race detection. Our experiment shows that, compared with PACER on the same platform and at the same sampling rate (such as 1%), CARISMA is significantly more effective.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors—Debuggers, run-time environments; D.2.5 [**Software Engineering**]: Testing and Debugging—Testing tools.

General Terms
Reliability, Experimentation, Verification.

Keywords
Concurrency, Data Races, Sampling, Bug Detection

ISSTA'12, July 15–20, 2012, Minneapolis, MN, USA.

1. INTRODUCTION
A multithreaded program may produce a large number of thread interleaving sequences even for the same input. Various concurrency bugs (such as data races and atomicity violations) still exist in heavily-tested real-world programs such as Eclipse [2], MySQL [21], and Firefox [21]. A (*data*) *race condition* [27] refers to multiple accesses from different threads to the same memory location in an unsynchronized way, and at least one of them is a write. Data races may not result in program failures, but their presence often indicates other concurrency errors in the same programs [2][21][23]. As such, it is essential to detect them.

During any execution, accesses to a shared memory location can be in a race. A data race may only occur in some specific but not all thread-scheduled conditions, and the number of threads that would trigger such specific conditions is not known in advance. This adds further challenges to data race detection.

In general, static analyses can provide thorough code coverage and often result in no false negatives, but they still have limitations such as producing false positives and being less scalable than their dynamic counterparts in handling large-scale programs [11][25]. Manually removing false positives is tedious because developers need to examine the code to identify the genuine ones (even though they may apply heuristic classification techniques first) [8]. Many model checking approaches [12][14] have been proposed, but they are still unable to scale well to handle large programs.

Many dynamic race detectors [5][13][17] are precise by monitoring actual program executions and tracking the happen-before relations [20] among threads, shared locations, and locks. However, they still incur significant overheads [9] and cannot scale up to handle large-scale programs that involve many shared locations or many accesses to shared locations in their executions [2][26]. Depending on the tradeoff between detection effectiveness and slowdown overheads in their design, techniques of this kind may produce some or no false positives as well as some or no false negatives. However, even the most efficient precise data race detector still incurs significant overheads (such as 8x slowdown in an experiment with FASTTRACK [9] for Java programs).

A promising approach that explores the above tradeoff is sampling. For instance, LITERACE [23] uses a code-partitioning strategy that keeps two copies of every function *F* of a program — the first one only samples the synchronization events of *F*, and the

second one samples the synchronization as well as the memory access events of *F*. For each function, LITERACE sets up a thread-local burst-sampling strategy [13] to decide which of the two copies should be invoked by the corresponding thread. With such a design, for functions that have been relatively more frequently invoked by a thread, the memory access events in the function will be sampled less often. However, for functions such as one that processes a large array of data for example, LITERACE may sample excessively. On the other hand, PACER [2] uses a global execution-time sampling strategy. It divides an execution into a consecutive series of sampling and non-sampling periods. During a sampling period, all synchronization and memory access events are sampled, whereas in a non-sampling period, the first read and the first write access events of a shared location are sampled only if the location has been sampled in a prior sampling period in the same execution. By so doing, PACER can detect races so long as at least one of the involved concurrent accesses is in a sampling period. Their empirical study [2] shows that the technique can significantly reduce the slowdown overheads.

A common limitation of the above techniques is that although existing sampling-based precise data race detectors such as PACER [2] and LITERACE [23] can effectively reduce overheads so that lightweight precise race detection can be performed efficiently in testing or post-deployment environments, they are ineffective in detecting races when the sampling rates are low. Our insight is that along an execution trace, a program may potentially handle numerous accesses to the memory locations created at the same site for the same purpose. These race detection techniques have not exploited the "similarity" among shared locations (such as manipulating objects in the same class hierarchy in Java, or elements in the same array or linked list). Intuitively, they may perform redundant memory access sampling, which lowers their chance of detecting rare data races.

In this paper, we present CARISMA, which exploits the above-mentioned insight. CARISMA works over a sequence of execution trials. In the first phase, given a particular sampling rate r, CARISMA adaptively estimates the sampling budgets for individual contexts (i.e., memory location creation sites) by carrying forward a set of statistics collected from the first m execution trials to the (m+1)-th trial so that all particular contexts can be evenly sampled as much as possible. In the second phase, CARISMA probabilistically determines whether to sample an entity in a particular context. In our experiment on a suite of nine benchmarks, we find that (1) CARISMA is significantly more effective than PACER; (2) CARISMA incurs 80% less runtime overhead than FASTTRACK running on the same platform at a sampling rate of 1%; and (3) CARISMA retains a detection rate of 10% at a low sampling rate of 0.01%, which is encouraging.

This paper makes the following contributions: (a) proposing the first budget balancing approach, CARISMA, for memory access sampling for data race detection; (b) formulating three levels of sampling granularities for concurrency bug detection; and (c) presenting an experiment that validates CARISMA extensively.

The rest of the paper is organized as follows: Section 2 introduces a motivating example. Section 3 presents CARISMA. Section 4 presents an evaluation of CARISMA. We then review related work in Section 5, and conclude the paper in Section 6.

2. MOTIVATING EXAMPLE

We motivate our work by an example shown in Figures 1 and 2. Figure 1 lists an `Account` class of a banking system with two subtypes: general and savings accounts (created by the methods `createGeneral()` and `createSavings()`, respectively), which are differentiated by the value of `isSavings`. Both subtypes support money deposits, but only savings accounts offer interest. The two operations are implemented by the methods `deposit()` and `creditInterest()`, respectively. To make our presentation more concise, we also use the shorthand `CLS-A` to denote the `Account` class, and `Stm-1` and `Stm-2` to denote the statements in lines 13 and 16 of the code, respectively.

```
 1 public class Account {  // denoted as CLS-A
 2    boolean isSavings;
 3    double balance = 0.0;
 4    Account(boolean b) { isSavings = b; }
 5    boolean IsSavings() { return isSavings; }
 6    synchronized void deposit(double x) {
            // denoted by dep()
 7       balance += x;
 8    }
 9    void creditInterest(double r) {
            // denoted by cre()
10       balance *= (1 + r);  // thread-unsafe
11    }
12    static Account createGeneral() {
13       return new Account(false);
            // denoted by Stm-1
14    }
15    static Account createSavings() {
16       return new Account(true);
            // denoted by Stm-2
17    }
18 }
```

Figure 1. Example of a banking system.

```
 1 public class AccountTest {
 2    Account[] a = new Account[9];
 3    Thread t1 = new Thread() {  // thread t1
 4       public void run() {
 5          for (int i = 0; i < 9; i++)
 6             a[i].deposit(1.0);
 7       }
 8    };
 9    Thread t2 = new Thread() {  // thread t2
10       public void run() {
11          for (int i = 0; i < 9; i++)
12             if (a[i].IsSavings())
13                a[i].creditInterest(0.1);
14       }
15    };
      // The thread is denoted by tmain with type Thd-M
16    public void testCase() {
17       for (int i = 0; i < 8; i++)
18          a[i] = Account.createGeneral();
19       a[8] = Account.createSavings();
20       t1.start(); t2.start();
21       t1.join(); t2.join();
22    }
23 }
```

Figure 2. A test case of the banking system.

The keyword "`synchronized`" is missing from the declaration of the method `creditInterest()`. Figure 2 shows a test case that reveals a failure. This test case simulates a real-life situation that the number of general accounts is much larger than the number of savings accounts. The test case initializes eight general accounts (`a[0]`–`a[7]`) and one savings account (`a[8]`).

After initialization, the test case starts two threads, t_1 and t_2. Thread t_1 models daily deposit transactions where deposits are made to both types of accounts, and thread t_2 models the interest payment transactions where only savings accounts receive interests. Consider an execution trace for the test case, as illustrated in the first two columns of Table 1. The execution sequence of the methods of the nine objects is shown from top to bottom. The execution first invokes `deposit()` of `a[0]`–`a[4]`, then t_2 calls `creditInterest()` of `a[8]`, and finally t_1 calls `deposit()` of `a[5]`–`a[8]`. In the course of the execution, because no lock protects any memory access in `creditInterest()`, the invoca-

tion of `creditInterest()` by t_2 may result in a data race on the field `a[8].balance` with the concurrent invocation of the method `deposit()` on `a[8]` by t_1.

LITERACE samples either all data accesses in a method instance or none of them. It must sample the first `creditInterest()` instance. *If* it could *evenly* sample 40% of all `deposit()` instances, it would achieve on average a detection rate of 40%.

In reality, however, LITERACE uses an adaptive bursty tracing strategy proposed in [13] to sample each method executed by the same thread. It gradually lowers the probability of sampling the same method. For the sake of illustration, in the example, we set LITERACE to skip 0, 1, 2, …, subsequent accesses of the method after the 1st, 2nd, 3rd, …, accesses to the same method, respectively. Hence, LITERACE samples 4 out of 9 `deposit()` invocations of the execution trace as shown in Table 1. Thus, it samples a total of 5 out of 10 accesses. Because `a[8].deposit()` occurs after 8 invocations of the `deposit()` method in the course of the execution, LITERACE fails to detect the race in the example.

Adjusting its configuration parameters does not *generally* solve the problem, because the number of occurrences of `deposit()` for the general accounts in a thread before the occurrence of `deposit()` for the savings account in the same thread cannot be precisely known in advance. (In addition, we followed Bond et al. [2] to introduce randomness to LITERACE. We find that the probability of sampling problematic accesses can be roughly 40%.)

For the purpose of illustration, we also set **PACER** [2] to sample the whole method in sampling periods and none of its content in non-sampling periods. Column 4 shows the sampling result of PACER (as indicated by hollow dots "○") for a *target* sampling rate of 40%. Because PACER extends its sampling period to include the first read access and the first write access on each memory location in subsequent non-sampling periods if the location has been sampled in a previous sampling period, it detects the data race in Figure 2 as long as the method `a[8].creditInterest()` is in a sampling period. The probability of sampling one method is 0.4. Hence, the probability of detecting the data race is 40% × 100% + 60% × 40% = 64%.

Table 1. Comparison among different sampling techniques

Sample Execution				LITERACE	PACER	CARISMA
Thread t_1	Thread t_2	In C_1?	In C_2?	Sampled?	Sampled? (Sampling Rate)	Sampled? (Sampling Rate)
$dep_0()$		yes		●	○ (0.4)	○ (0.25)
$dep_1()$		yes		●	○ (0.4)	○ (0.25)
$dep_2()$		yes			○ (0.4)	○ (0.25)
$dep_3()$		yes		●	○ (0.4)	○ (0.25)
$dep_4()$		yes			○ (0.4)	○ (0.25)
	$cre_8()$		yes	●	○ (0.4)	●
$dep_5()$		yes			○ (0.4)	○ (0.25)
$dep_6()$		yes		●	○ (0.4)	○ (0.25)
$dep_7()$		yes			○ (0.4)	○ (0.25)
$dep_8()$			yes		○ (0.64)	●
Count:		8	2			
Expected No. of Accesses Sampled:				5	4 .24 (=0.4×9+0.64)	**4** (=2+0.25×8)
Detection Rate:				0%	42.4%	**100%**

Notes:
(1) $dep_i()$ and $cre_i()$ refer to `a[i].deposit()` and `a[i].creditInterest()`, respectively, in Figure 2.
(2) A filled dot ("●") refers to a sampling rate of 100%.
(3) A hollow dot ("○") refers to sampling with a probability of r indicated in the brackets that follow.
(4) The two shaded method invocations lead to a data race in the example.

Unlike LITERACE and PACER, our **CARISMA** approach is context-sensitive and driven by a sequence of execution trials to achieve effective sampling across these trials. We define the *context* of a memory location x as a triple ⟨*threadType*, *dataType*, *allocSite*⟩, referring, respectively, to the type of the thread that allocates x, the data type of x, and the call site at which x is allocated. For simplicity, we use the corresponding statement number for *allocSite*. Hence, in the motivating example, two contexts for the two respective types of accounts are C_1 = ⟨`Thd-M`, `CLS-A`, `Stm-1`⟩ for general accounts and C_2 = ⟨`Thd-M`, `CLS-A`, `Stm-2`⟩ for savings accounts, where `Thd-M` is the type of the thread that executes method `testcase()` in Figure 2.

CARISMA collects the sampling statistics of each context in the course of each execution. Between the m-th and the (m+1)-th trials, CARISMA carries forward the accumulated statistics in the first m execution trials to the (m+1)-th trial. It re-estimates the concentrations of entities (at memory location level or memory access level) under each context, re-estimates a budget (i.e., the total sampling quotas) available to the (m+1)-th trial, and re-distributes the estimated sampling quotas among different contexts for the (m+1)-th trial based on the target sampling rate.

Suppose the following four conditions hold. (i) The target sampling rate for CARISMA is also set to 40%. (ii) The budget for the (m+1)-th trial is estimated to be 4. (iii) The contexts C_1 and C_2 have been discovered in the first m trials. (iv) Based on the first m execution trials, C_1 and C_2 have been estimated to contain, on average, 8 and 2 memory locations, respectively.

CARISMA balances the sampling budget of 4 memory locations among C_1 and C_2 by allocating 2 to each context. Because, on average, C_1 has been estimated to have 8 entities, the effective sampling rate of every entity grouped under C_1 is 25%. For the same reason, CARISMA allocates a sampling rate of 100% to every entity grouped under C_2. The sampling result of CARISMA for the (m+1)-th trial is shown in the rightmost column of Table 1. The race is detected because it has sampled both problematic memory accesses associated with the savings account.

CARISMA checks both contexts of memory locations more evenly than LITERACE and PACER. Moreover, for CARISMA, although the sampling rate for general accounts is lower, the larger estimated population of this kind of account compensates for the reduced rate in the long run. Hence, the overall sampling effort on general accounts will not be significantly compromised.

We further note that if `Stm-1` and `Stm-2` are replaced by two calls to the same static method that creates all account objects, C_1 and C_2 will merge into the same context. By adding the calling context into the definition of a context for the example, one can distinguish between them while our basic idea remains unchanged.

3. CARISMA

This section presents our **C**ontext-sensitive **A**pproach to **R**ace-condition sample-**I**nstance **S**election for **M**ultithreaded **A**pplications (**CARISMA**).

CARISMA operates a sequence of k execution trials with a low target sampling rate r and with every context having a minimal sampling rate r_{min}. An execution trial is an attempt to produce an execution trace. For every execution trial, CARISMA calls the program under test (P) with the same input.

CARISMA consists of two phases. **Phase 1** carries forward the statistics collected from the first m trials to estimate the sampling budgets for individual contexts to be used for the (m+1)-th trial. **Phase 2** actively determines whether a memory location related to a particular context is going to be sampled.

Once a memory access has been sampled, CARISMA sends this memory access to a dynamic data race detector *D* (such as FASTTRACK [9]) for race detection. Note that CARISMA also sends all lock acquisition and release events as well as all other synchronization events to *D*.

CARISMA exploits the concept of probabilistic sampling: If a memory location is not sampled on its first occurrence in a trial, it discards all accesses to the memory location in that trial (to eliminate the analysis conducted by *D* and reduce the overhead). This is feasible because we use a context-based sampling strategy.

3.1 Context

A ***context*** models the environment that allocates a memory location *x*, with the aim of grouping the set of memory locations that a program may handle in the same way into the same context. In this way, by sampling some but not all data accesses associated with a particular context, a technique may sample "similar" data accesses in a more focused manner.

For example, to allocate a set of "similar" objects, a programmer may write an iteration loop or a recursive call in the method to achieve the goal of creating these objects. On the other hand, in real-life applications, many memory locations allocated by different types of threads are designed with different access purposes (and different access patterns). For instance, the set of memory locations may be created using different call stacks even though they are created by the same method of a Java class. They should be considered separately in the sampling process.

Hence, we define the context of a memory location *x* by the triple ⟨*threadType*, *dataType*, *allocSite*⟩ as described in Section 2.

In the motivating example, as a thread is an object, CARISMA uses the class signature of the thread class as *threadType*. All memory locations of both contexts C_1 and C_2 are allocated by the same *threadType* `Thd-M` and of the same data type `Account` (`CLS-A`). Their *allocSite*s differ by the statements that allocate them: `Stm-1` for general account and `Stm-2` for savings account.

Whenever a thread creates a memory location *x*, CARISMA assigns a context to *x*. CARISMA then uses the available contexts in the rest of the current execution and the subsequent trials.

Note that CARISMA can also use other data structures, such as calling context [31], instead of simply the allocation statement.

3.2 Phase 1: Budget Estimation and Allocation

This section presents our budget allocation scheme and the method used by CARISMA to carry forward statistics between execution trials.

3.2.1 Sampling Budget Estimation

To estimate the sampling budget for the (*m*+1)-th trial, CARISMA first estimates the expected number of memory locations allocated and the expected number of memory locations under each context in this (*m*+1)-th trial, which are referred as *population size* and *context size*, respectively. We also refer to the total number of memory locations to be targeted for sampling as the *sampling budget* or simply the *budget* (denoted by *B*).

The budget *B* is determined by multiplying the population size with the target sampling rate r. Because r is given, we only need to estimate the population size to obtain *B*. The detail is as follows:

Suppose that *Cid* is a non-empty set of *n* contexts (denoted by $C_1, C_2, \ldots, C_n$) and we have executed the program *m* times (where $1 < m \leq k$). Let P_i be the context size of C_i. Let $r_{i,j}$ denote the sampling rate assigned to C_i in the *j*-th execution trial (where $j < m$) and $s_{i,j}$ denote the number of samples under C_i collected from this *j*-th execution trial.

We generally approximate each $s_{i,j}$ by a random variable with a binomial distribution denoted by $B(P_i, r_{i,j})$ and with an expected value $E[s_{i,j}]$ (which is the mean of the binomial distribution denoted by $r_{i,j}P_i$ [19]). Our estimator for P_i is given by:

$$\hat{P}_i = \frac{\sum_{j=1}^{m} s_{i,j}}{\sum_{j=1}^{m} r_{i,j}} \tag{1}$$

It is an unbiased estimator because:

$$\mathrm{E}\left[\hat{P}_i\right] = \mathrm{E}\left[\frac{\sum_{j=1}^{m} s_{i,j}}{\sum_{j=1}^{m} r_{i,j}}\right] = \frac{\sum_{j=1}^{m} \mathrm{E}\left[s_{i,j}\right]}{\sum_{j=1}^{m} r_{i,j}} = \frac{\sum_{j=1}^{m} r_{i,j} P_i}{\sum_{j=1}^{m} r_{i,j}} = P_i \tag{2}$$

As such, the value of $\sum_{i=1}^{m} \hat{P}_i$ gives an estimation of the population size. We then set $B = r \cdot \sum_{i=1}^{m} \hat{P}_i$, which hat gives an estimate of the sampling budget for the (*m*+1)-th execution trial.

3.2.2 Determination of Context Sampling Rate

After estimating the sampling budget *B* for the (*m*+1)-th trial, CARISMA proceeds to distribute *B* evenly among contexts so that every context have roughly the same number of expected samples. Note that in the same trial, all memory locations related to the same context have the same sampling rate, but those related to different contexts may have different sampling rates.

We use the term *context sampling budget*, denoted by B_i, to refer to the sampling budget allocated to context C_i.

CARISMA formulates the sampling budget allocation process as a single-objective convex optimization problem that minimizes the variances among B_i for all *i*, thus:

$$\begin{gathered} \min \sum_{i=1}^{n} \left(B_i - \frac{B}{n}\right)^2 \\ \text{subject to } B_i \in [r_{min}\hat{P}_i, \hat{P}_i] \text{ and } \sum_{i=1}^{n} B_i = B \end{gathered} \tag{3}$$

Eqn. (3) is a classical quadratic programming problem [3] with the constraints that the context sampling budget B_i is in the range $[r_{min}\hat{P}_i, \hat{P}_i]$ and the sum of all B_i's is *B*. In other words, the budget B_i should not exceed the corresponding estimated context size $\hat{P}_i$ and should retain a threshold level computed by $r_{min}\hat{P}_i$. (Otherwise, a context may be assigned with no budget.) The solution of Eqn. (3) is given by:

$$B_i = \min\left(\max\left(\nu, r_{min}\hat{P}_i\right), \hat{P}_i\right) \tag{4}$$

where $\nu \in \mathbb{R}$ and $\sum_{i=1}^{n} B_i = B$.

We observe that $\sum_{i=1}^{n} B_i$ is a piecewise-linear increasing function of "ν" with breakpoints $r_{min}\hat{P}_i$ and $\hat{P}_i$, so that the water-filling algorithm (which runs in $O(n \lg n)$ time due to sorting, where *n* is the number of contexts) [3] can be used to solve $\sum_{i=1}^{n} B_i = B$ for ν by increasing ν from $\min(r_{min}\hat{P}_i)$ until $\sum_{i=1}^{n} B_i = B$ holds.

As such, CARISMA first computes ν and then B_i for each *i*. To ease our description in Section 3.3.2, we denote above water-filling algorithm to compute ν by *WaterFilling*(*B*, *Cid*, r_{min}), where *Cid* (as defined in Section 3.2.1) is the set of contexts, in which each context C_i (for $i = 1, 2, \ldots, n$) is associated with a $\hat{P}_i$.

CARISMA then uses

$$r_i = \frac{B_i}{\hat{P}_i} \tag{5}$$

to compute each r_i for context C_i to be used in the next execution trial. For ease of presentation, we refer to r_i as the *context sampling rate* for C_i.

For any context C_i that has no sample collected yet (which also applies to the very first execution of any context), CARISMA simply lets r_i be the given target sampling rate r.

3.2.3 Carrying Forward the Statistics

As presented in the last two subsections, CARISMA has computed the context sampling rate for each context (and uses r as default for contexts with no information collected yet). In this subsection, we present how CARISMA carries forward the statistics between consecutive execution trials.

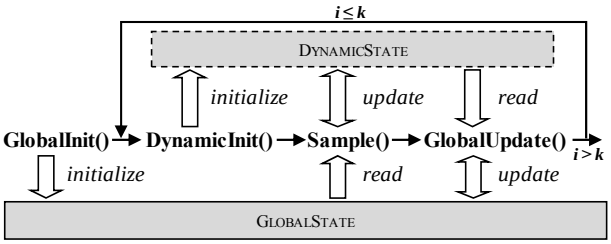

Figure 3. State maintenance of CARISMA in Phase 1

Figure 3 depicts how CARISMA maintains its states. It first initializes its global states (GLOBALSTATE) by calling *GlobalInit*( ). GLOBALSTATE is the data structure that keeps the sampling statistics to be carried forward between execution trials, while DYNAMICSTATE is the data structure that keeps the statistics for the current execution. Right before each execution trial on the program under test, CARISMA calls *DynamicInit*( ) to initialize (i.e., reset) its variable. Table 2 shows the definitions of variables for the data structures. The column "Initial Value" shows how each variable is initialized by *GlobalInit*( ) and *DynamicInit*( ).

Then, CARISMA proceeds to Phase 2, as shown in Figure 3. In the course of the execution of the trial, CARISMA calls *Sample*( ) to collect the statistics and updates DYNAMICSTATE dynamically. After executing the program for the trial, CARISMA updates GLOBALSTATE based on the original values of GLOBALSTATE and the DYNAMICSTATE via the function *GlobalUpdate*( ). If the number of execution trials is still less than *k*, CARISMA will start the next one.

For ease of reference, we use the notation GLOBALSTATE.*x* and DYNAMICSTATE.*y* to refer to the variables of GLOBALSTATE and DYNAMICSTATE, respectively. For instance, the variable *Cid* of GLOBALSTATE in Table 2 is denoted by GLOBALSTATE.*Cid*.

In the next subsection, we will present the procedure of *GlobalUpdate*( ) of CARISMA after presenting the procedure *Sample*( ) first, because the strategy used in *GlobalUpdate*( ) depends on the strategy applied in *Sample*( ).

3.3 Phase 2: Execution Monitoring

3.3.1 Overview

In Phase 2, the procedure *Sample*( ) of CARISMA monitors the course of program executions. It forwards all sampled events, synchronization events, and thread management events to the associated race detector.

As we have presented in Section 3.1, whenever a memory location *x* is created, CARISMA checks whether its context C_i exists in GLOBALSTATE, and if not, it creates a new context for *x*. Then, when an access to memory location *x* occurs (regardless of whether it is a read or a write), CARISMA checks whether the access should be forwarded to the race detector *D* according to the context sampling rate r_i associated with context C_i. If this access is decided to be sampled, CARISMA makes such forwarding. Also, for every write access to memory location *x*, CARISMA updates its DYNAMICSTATE to keep track of the statistics of context C_i needed for future execution trials, in which CARISMA applies the budget estimation for Phase 1 (see Section 3.2) that estimates both the population size and the context sizes.

The following subsection presents three sampling strategies of CARISMA. Their descriptions in algorithmic format will be published in a separate report.

3.3.2 Strategies

CARISMA can be configured to use a family of strategies. Three of them are *Coarse Location Sampler* (CLS), *Fine Location Sampler* (FLS), and *Adaptive Access Sampler* (AAS). They differ mainly in terms of the context granularity. Choosing a strategy affects not only this phase but also Phase 1 because they affect how CARISMA estimates the context sizes.

We first use an example to show the difference in sampling granularity among the three strategies: Suppose that (1) r_{min} is 0.3, (2) the context sizes for a given context are 4 memory locations for CLS, and 12 write operations for both FLS and AAS, and (3) the sampling budgets for the context under the CLS, FLS,

Table 2. GLOBALSTATE and DYNAMICSTATE used by CLS, FLS and AAS

$\mathbb{N}_0$ denotes the set of natural numbers including zero. $\mathbb{N}_1$ denotes the set of natural numbers excluding zero. **bool** = {**false**, **true**} is the set of Boolean values. $Instr_0$ is an integer constant that indicates the length of the sampling periods for memory accesses.

	Variable	Type	Initial Value	Description	CLS	FLS	AAS
GLOBALSTATE	*Cid*	set of contexts	∅	Records of all contexts occurring in all prior executions.	✓	✓	✓
	N	$\mathbb{N}_1$	1	Identity of the execution (i.e., the index of the current trial).	✓	✓	✓
	R	$Cid \rightarrow [0, 1]$	$\lambda c.\ r$	Context sampling rate r_i for the context C_i in a particular execution. We simply refer to $R(i)$ as r_i.	✓	✓	✓
	R_{sum}	$Cid \rightarrow \mathbb{R}$	$\lambda c.0$	Mean value of the context sampling rates in all prior executions for the same context.	✓	✓	✓
	S_{sum}	$Cid \rightarrow \mathbb{N}_0$	$\lambda c.0$	Accumulated number of samples obtained for the same context in all prior executions.	✓	✓	✓
	Chk_0	$Cid \rightarrow \mathbb{N}_0$	$\lambda c.0$	Length of non-sampling period of memory accesses for each context.			✓
DYNAMICSTATE	*F*	$Mem \rightarrow$ bool	$\lambda x.$false	Record of whether the memory location has been sampled.	✓	✓	✓
	S	$Cid \rightarrow \mathbb{N}_0$	$\lambda c.0$	Number of samples obtained in the current execution for each context.	✓	✓	✓
	C	$Mem \rightarrow Cid$	$\lambda x.$null	Map from a memory location to a context.		✓	✓
	Chk	$Mem \rightarrow \mathbb{N}_0$	$\lambda x.0$	Count of the length of non-sampling periods of memory accesses.			✓
	Instr	$Mem \rightarrow \mathbb{N}_0$	$\lambda x.Instr_0$	Count of the length of burst sampling periods of memory accesses.			✓

and AAS strategies are 1 location, 3 accesses, and 3 accesses, respectively. According to Eqn. (5), their context sampling rates are 0.25, 0.25, and 0.25, respectively. Because $0.25 < r_{min}$, both CLS and FLS set the context sampling rate to be r_{min}; whereas AAS sets the context sampling rate back to 0.3 at the memory location level and activates sampling at the memory access level with a rate of 0.833 (= 0.25 ÷ 0.3). Table 3 summarizes the sampling parameters computed by CARISMA in Phase 1 and Phase 2. The details of each strategy are discussed in the following paragraphs.

Table 3. Comparison of three stratgeis of CARISMA: CLS, FLS and AAS. We assume $r_{\min} = 0.3$ and there are three memory accesses to each memory location.

	Estimation	CLS	FLS	AAS
Phase 1	Context size	4 memory locations	12 write accesses	12 write accesses
	Context sampling budget	1 location	3 accesses	3 accesses
	Computed sampling rate	25%	25%	25%
Phase 2	Context sampling rate (memory location level)	30%	30%	30%
	Context sampling rate (memory access level)	N/A	N/A	83%
	Effective sampling rate	30%	30%	25%

***Coarse Location Sampler* (CLS).** The CLS strategy estimates the context size and monitors memory accesses at the memory location level for both phases. The motivating example in Section 2 illustrates this strategy. CLS works as follows:

When a memory location x is created in the course of an execution, CLS creates a context C_i for it. It adds this context to GLOBALSTATE.*Cid* if C_i does not exist in the set *Cid*.

CLS exploits the probabilistic sampling to conserve memory consumption: CLS generates a random number α in the range of [0, 1]. Because each context is associated with a context sampling rate r_i, CLS checks whether the condition $r_i > \alpha$ holds. If so, it sets DYNAMICSTATE.F(x) to *true* and increments DYNAMICSTATE.S(C_i) by 1, indicating that one more location, which is location x, is sampled for the current trial. If the condition $r_i > \alpha$ does not hold, CLS sets DYNAMICSTATE.F(x) to *false*, indicating that location x will not be sampled for the current trial.

The key design in the above procedure is that, rather than computing whether each particular write access and each particular read access should be sampled probabilistically, CLS simplifies the decision procedure by probabilistically determining whether all memory accesses of the same location should be sampled.

Hence, when a write access or a read access event in the course of the current execution is monitored, CLS simply checks whether DYNAMICSTATE.F(x) has been set to *true*. If so, CLS forwards the access event to the race detector D; otherwise, it simply skips the event.

We have explained in Section 3.2.3 that the procedure of *GlobalUpdate*( ) depends on the strategies used in *Sample*( ). For CLS, *GlobalUpdate*( ) works as follows:

We recall from the above paragraphs of CLS that DYNAMICSTATE.F( ) and DYNAMIC-STATE.S( ) have maintained the list of sampled memory locations at the end of an execution trial. In *GlobalUpdate*( ), for each context C_i, CLS first increments GLOBALSTATE.$R_{sum}(C_i)$ by r_i and GLOBALSTATE.$S_{sum}(C_i)$ by DYNAMICSTATE.S(C_i). With these two values, by Eqn. (1), CLS computes $\hat{P}_i$ = GLOBALSTATE.$S_{sum}(C_i)$ ÷ GLOBALSTATE.$R_{sum}(C_i)$. Then, CSL computes a new budget B by Eqn. (2), v by *WaterFilling*(B, *Cid*, r_{min}), and a new context sampling rate r_i by Eqn. (5) for each context C_i. It finally adds 1 to GLOBALSTATE.N to indicate that Phase 1 of the next trial is ready to start.

We have explained above how CLS handles memory location creation as well as memory read and write access events, and how it carries forward the information between two consecutive executions. We now present the second sampling strategy of CARISMA.

***Fine Location Sampler* (FLS).** The FLS strategy is the same as the CLS strategy except that it estimates the context size at the memory access level (in Phase 1) but performs sampling at the memory location level (in Phase 2). Because a memory location may have multiple accesses, FLS is finer in granularity than CLS. For brevity, we mainly present the similarities and differences between CLS and FLS.

Like CLS, at the time that memory location x is created in the course of an execution, FLS creates a context C_i for x and also checks whether the condition $r_i > \alpha$ holds. If so, FLS sets DYNAMICSTATE.F(x) to *true*. However, FLS does not change DYNAMICSTATE.S(C_i) as what CLS does; rather, FLS associates x with context C_i by assigning C_i to DYNAMICSTATE.$C(x)$. If the condition does not hold, FLS works in the same way as CLS.

When a write access or a read access event on a memory location x is monitored in the course of the same execution, FLS works in the same way as CLS, except that if FLS forwards a write access of x to the race detector D, FLS also increments DYNAMICSTATE.S(C_i) by 1, where C_i is the context kept at DYNAMICSTATE.$C(x)$. The *GlobalUpdate*( ) of FLS is the same as that of CLS.

We note that DYNAMICSTATE.S(C_i) of FLS is incremented at the access level and restricted to count the number of write accesses forwarded to the race detector. This is motivated by the following two reasons: (1) Each data race must involve at least one write access, and may not involve any read access; and (2) the authors of [9] reported that write accesses only take up a small proportion of all monitored operations (which is 14.5% in their experiment), whereas read accesses take up a larger ratio (82.3% in their experiment). Hence, recording write accesses only can potentially alleviate CARISMA's overhead.

***Adaptive Access Sampler* (AAS).** The AAS strategy targets to both estimate the context size and do sampling at the memory access level. Its sampling strategy is more complex due to sampling at the memory access level. AAS applies the burst-sampling strategy, and following [13], it uses four parameters: DYNAMICSTATE.*Skip*(x), DYNAMICSTATE.*Smpl*(x), GLOBALSTATE.$Skip_0(C_i)$, and GLOBALSTATE.$Smpl_0$. (We drop their prefixes in following paragraphs to save page space.) We note that the first two parameters are at the location level, the third is at the context level, and the fourth is a constant.

When a memory location x is created in the course of an execution, AAS behaves in the same way as FLS, except that if AAS forwards the creation event to the race detector D, AAS additionally computes $Skip(x) = \lfloor rand(\,) \times (Skip_0(c) + 1) \rfloor$, where *rand*( ) generates a random number in the range of [0, 1]. We use this variable $Skip(x)$ to decide on the time to activate sampling at the access level for the same location x. We choose the activation condition $Skip(x) = 0$, and before this condition is reached, each memory access to x is skipped, as explained as follows:

When a read access event on a memory location x is monitored in the course of the current execution, AAS forwards the event to the race detector D if both DYNAMICSTATE.F(x) is *true* and *Skip*(x) = 0; otherwise, it skips the event.

For a write access event of x, there are three cases to consider: (a) If DYNAMICSTATE.F(x) is *false*, AAS skips the event. (b) If the event is not skipped by the first case and *Skip*(x) > 0, AAS decrements *Skip*(x) by 1, and skips the event. (c) If the event is still not skipped by the above two cases, AAS works identically to FLS to forward the write access of x to the race detector D, increments DYNAMICSTATE.S(C_i) by 1, where C_i is the context kept at DYNAMICSTATE.C(x), and decrements *Smpl*(x) by 1. If this updated *Smpl*(x) is 0, AAS resets *Skip*(x) to $Skip_0(C_i)$ and *Smpl*(x) to $Smpl_0$.

In essence, the AAS strategy samples a location if *Skip*(x) = 0. The number of samples under the condition *Skip*(x) = 0 (which depends on the sampling period for the location) is controlled by *Smpl*(x). This sampling period of consecutive accesses is further controlled by the two parameters $Skip_0(C_i)$ and $Smpl_0$.

GlobalUpdate( ) of AAS is the same as that of FLS except that it computes v and r_i by substituting r_{min} by 0 in Eqn. (4). Moreover, AAS computes $Skip_0(C_i)$ for each context C_i as follows: If $r_i \geq r_{min}$, AAS sets $Skip_0(C_i) = 0$ so that AAS can sample all the accesses of a location at this rate because the context is assigned with a sampling rate larger than r_{min}. If $r_i < r_{min}$, AAS first sets r_i back to r_{min}; then, it computes $Smpl_0 \times (r_{min}/r_i - 1)$ and assigns the computed value to $Skip_0(C_i)$.

In brief, if the computed sampling rate for the context is too small, AAS sets to sample a portion of the memory accesses of the memory locations whose sampling rates are kept at r_{min}. The purpose of such assignment is to ensure an adequate number of memory locations to be sampled. Following [13], we set the $Smpl_0$ to be 10 in our experiments.

4. EXPERIMENTATION

4.1 Implementation

CARISMA was implemented as a tool on top of the ROADRUNNER platform [10], a published Java instrumentation framework. We enhanced this instrumentation framework by further observing the instantiation of objects (as required for the sampling strategies). When an object is instantiated, CARISMA records the type of the current thread and the relevant statement. (We use the statement that initiates the object as the *allocSite* of the context.) CARISMA uses the type of the enclosing object class together with the name of the field to represent the data type of the field. When an object field is accessed for the first time, CARISMA computes the context from its type, the thread, and the statement recorded in its enclosing object, and stores the context as a 64-bit integer in the field's `ShadowVar` [10]. For static fields, we use the type of the thread that executes the static initializer together with the data types of the fields as their context and ignore the allocation statement because it is not specified in the source code.

In our implementation, we avoided using hash tables and stored the context information directly in the objects themselves. Meanwhile, we avoided using thread-shared data structures and utilities such as random number generators, and made them thread-local so that there was no need to synchronize them.

4.2 Evaluation Settings

This section presents the evaluation settings for CARISMA.

Configuration. We used a Dell PowerEdge 1950 server running on Solaris UNIX to conduct the experiment. The server had 2 Xeon 5355 (2.66 GHz, 4-core) processors with 8 GB memory.

Benchmarks. We used nine popular Java benchmarks in our experiment, as shown Table 4. Each program has at least one data race detectable by FASTTRACK [9][10] in 100 executions. They include raytracer, a ray-tracing program [16]; tsp, a traveling salesman problem solver [9]; mtrt, another ray-tracing program from the SPEC JVM98 benchmarks suite [32]; jbb, the SPEC JBB2000 business object simulator [32]; hedc, a web crawler from ETH [9]; weblech, another open-sourced web crawler [29]; cache4j, an open-source tool for caching java objects [29]; xalan, an XML transformer; and hsqldb, a relational database engine from Dacapo benchmarks [6]. (We also tried to use eclipse but it could not work correctly with ROADRUNNER, with or without CARISMA on our server.) We used the test harnesses provided by the benchmarks. We configured all benchmarks to run with eight threads, except cache4j, xalan, and hsqldb for which the numbers of threads were not configurable.

Table 4. Descriptive statistics of benchmarks

Name	Size (LOC)	# of threads	# of locations	# of accesses	# of races	# of contexts
mtrt	3,755	8	23 M	152 M	1	474
raytracer	1,532	8	224 M	9,741 M	1	193
tsp	720	8	180 k	120 M	1	40
hedc	7,817	8	17 k	79 k	4	145
jbb	29,058	11	70 M	305 M	2	1,652
weblech	1,952	8	0.6 k	6 k	2	127
cache4j	3,930	2	46 k	711 k	2	230
xalan	345,088	9	22 M	105 M	19	2,242
hsqldb	253,044	81	13 M	302 M	18	1,288

Table 4 presents the statistics of the benchmarks. The data shown in columns 3–5 were obtained from the original ROADRUNNER platform [10] averaged over 100 executions. (We followed the execution times reported in [28] for computationally intensive programs like mtrt and raytracer.) Column 3 shows the maximum number of live threads at any time in our experiment. The column "# of locations" refers to the number of memory slots (one slot from each object field) in the heap used by a program that ROADRUNNER reported. In Java, a memory slot stores the value of a primitive data type (such as `int`, `float`, and `double`) or an object reference. Column 6 shows the number of distinct data races that were reported by the FASTTRACK tool in the experiment.

Techniques. We evaluated the three strategies (CLS, FLS, and AAS) of CARISMA and compared them with PACER [2] (denoted by PCR), which is the best state-of-the-art sampling technique for data race detection. We configured CARISMA to use FASTTRACK, shipped with ROADRUNNER, as the associated race detector.

We implemented PACER on top of ROADRUNNER by adding the sampling infrastructure into FASTTRACK and modifying the vector operations [2] using the state-of-the-art epoch approach proposed by FASTTRACK. Because ROADRUNNER did not control garbage collection, we used a timer and synchronizations to switch between sampling and non-sampling periods. When the signal from timer arrives, a global flag f_G was set to indicate that a sampling decision was going to be made. Our version of PACER checked f_G every time before it processed the critical operations. If f_G was not set, it proceeded with the processing and set a thread-local flag f_t, which was unset when the processing was done; otherwise, it waited until

f_G was unset. On the other hand, after setting f_G, the callback procedure would wait until the f_t flags of all threads were unset.

Executions. The range of sampling rates we used in our experiments is [0.0001, 1], and we set the minimum sampling rate r_{min}= 0.00001 to make sure that the sample budget should cover at least 0.001% of the context population. Strictly following the experimental setup presented in [2], we determine the number of executions by making the probability of detecting a race greater than 0.95 for a given sampling rate, and limit the number of executions to be between 50 and 500. The number of execution is given by the formula:

$$nTrials_r = \min(\max(\lceil \log_{1-r} 0.05 \rceil, 50), 500)$$

Table 5 shows the mean effective sampling rates (plus or minus one standard deviation) realized by CARISMA, which are close to the specified target sampling rate r. We observe that, overall speaking, the sampling rates for benchmarks with more memory locations and accesses tend to be closer to r and vary less.

4.3 Effectiveness of Data Race Detection

This section evaluates the effectiveness of CARISMA.

CARISMA works over a sequence of execution trials. Figure 4 shows the mean detection rates (over 100 repetitions for each of the first 40 trials) achieved by both AAS and PACER on xalan (the largest subject in the benchmark) at the target sampling rate r of 0.01. The *x*-axis represents the index of the execution (starting from zero) and the *y*-axis represents the average detection rate for a given execution indexed by the *x*-axis. (It is *not* the aggregate probability for the previous executions). In the first few trials, AAS is less effective than PACER, but the mean effectiveness of AAS increases over execution trials, and finally reaches a level that is much higher than that of PACER.

Figure 5 shows a summary of the detection rates over all benchmarks and Figure 6 shows the mean detection rates of PACER and each strategy of CARISMA on all benchmarks. In each plot, the *x*-axis is the target sampling rate and the *y*-axis is the race detection effectiveness. Each point is the *unweighted* average over all races. We adopt from [10] the definition of detection rate of a strategy/technique as the ratio of (1) the mean number of detected races per execution at target sampling rate r to (2) the mean number of races detected with r = 100%.Among our three strategies, both AAS and FLS achieve much higher detections rates than that of CLS, and the detection rate of AAS is slightly higher than that of FLS on all benchmarks. Both AAS and FLS use the memory access as the granularity for the estimation of statistics, whereas CLS uses the memory location as the granularity. We observe that a finer granularity tends to characterize the differences among contexts more effectively.

In six out of all benchmarks, CARISMA can detect more than 10% data races using a target sampling rate of 0.01%, except for hedc, weblech and cache4j, which are smaller in scale than other benchmarks in terms of the number of memory locations and the number of accesses in the course of an execution. We have investigated the rationale. For weblech, the detection rate of either CLS or FLS drops quickly as the sampling rate decreases, because there are so few memory locations to be sampled; and hence the expected numbers of memory locations sampled in some contexts were less than one. AAS adaptively samples memory accesses and its line for weblech has a gentler slope than that of CLS or FLS. Similar findings are observed on hedc and cache4j.

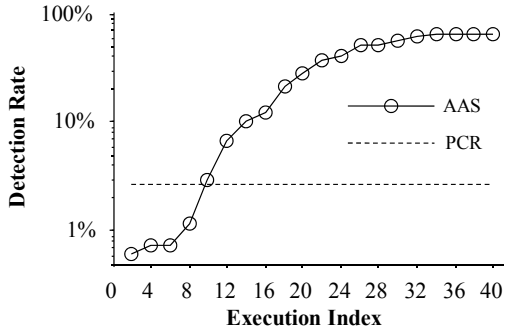

Figure 4. Change of detection rates over first 40 executions of xalan at r=1%.

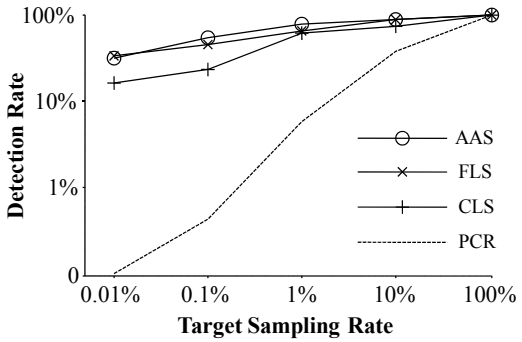

Figure 5. Summary of detection rates.

We find that PACER's mean detection rate of is only slightly better than constant with respect to r. This is because racy memory locations are usually accessed multiple times, and if any one of them is sampled, the race can be detected by PACER. On average, CARISMA shows a significantly higher detection rate than PACER.

Table 5. Mean effective sampling rates (± one standard deviation) as the percentage of memory accesses sampled

	AAS				FLS				CLS			
r	**0.01%**	**0.1%**	**1%**	**10%**	**0.01%**	**0.1%**	**1%**	**10%**	**0.01%**	**0.1%**	**1%**	**10%**
mtrt	0.009±0.003	0.10±0.01	1.1±0.2	10.2±1.1	0.011±0.004	0.09±0.03	1.0±0.3	10.0±1.1	0.009±0.002	0.09±0.04	0.8±0.2	10.9±1.0
raytracer	0.011±0.003	0.09±0.02	1.0±0.2	10.6±0.7	0.009±0.011	0.10±0.02	0.8±0.2	10.0±0.8	0.010±0.003	0.10±0.02	1.0±0.0	10.1±0.7
tsp	0.013±0.005	0.11±0.02	0.9±0.2	9.4±1.6	0.009±0.002	0.11±0.02	1.1±0.2	9.0±1.0	0.009±0.006	0.09±0.05	0.9±0.1	10.5±1.5
hedc	0.011±0.006	0.09±0.05	0.8±0.4	11.3±1.5	0.007±0.008	0.05±0.12	0.9±0.2	9.5±2.4	0.011±0.007	0.11±0.11	0.7±0.3	12.1±2.2
jbb	0.009±0.004	0.10±0.03	1.0±0.3	10.5±0.7	0.011±0.007	0.11±0.07	1.5±0.6	10.0±1.5	0.014±0.004	0.10±0.02	1.3±0.4	10.5±1.8
weblech	0.006±0.007	0.12±0.07	0.7±0.5	8.1±2.4	0.007±0.012	0.11±0.11	0.9±0.7	11.0±1.5	0.010±0.007	0.13±0.06	0.8±0.2	10.0±1.6
cache4j	0.012±0.004	0. 08±0.03	0.9±0.2	9.6±1.2	0.012±0.004	0. 08±0.02	1.2±0.2	9.5±1.0	0.007±0.006	0.13±0.03	0.9±0.2	10.0±1.0
xalan	0.010±0.003	0.12±0.04	1.0±0.3	11.9±0.6	0.013±0.003	0.09±0.06	1.2±0.2	9.4±1.2	0.011±0.005	0.09±0.03	0.9±0.1	10.0±1.4
hsqldb	0.011±0.004	0.09±0.03	1.1±0.4	9.2±1.3	0.006±0.004	0.10±0.04	1.4±0.4	10.2±1.5	0.010±0.003	0.11±0.04	1.0±0.4	9.8±1.2
Average:	*0.010±0.004*	*0.10±0.03*	*0.9±0.3*	*10.1±1.1*	*0.009±0.006*	*0.09±0.05*	*1.1±0.3*	*9.8±1.3*	*0.010±0.005*	*0.10±0.04*	*0.9±0.2*	*10.4±1.4*

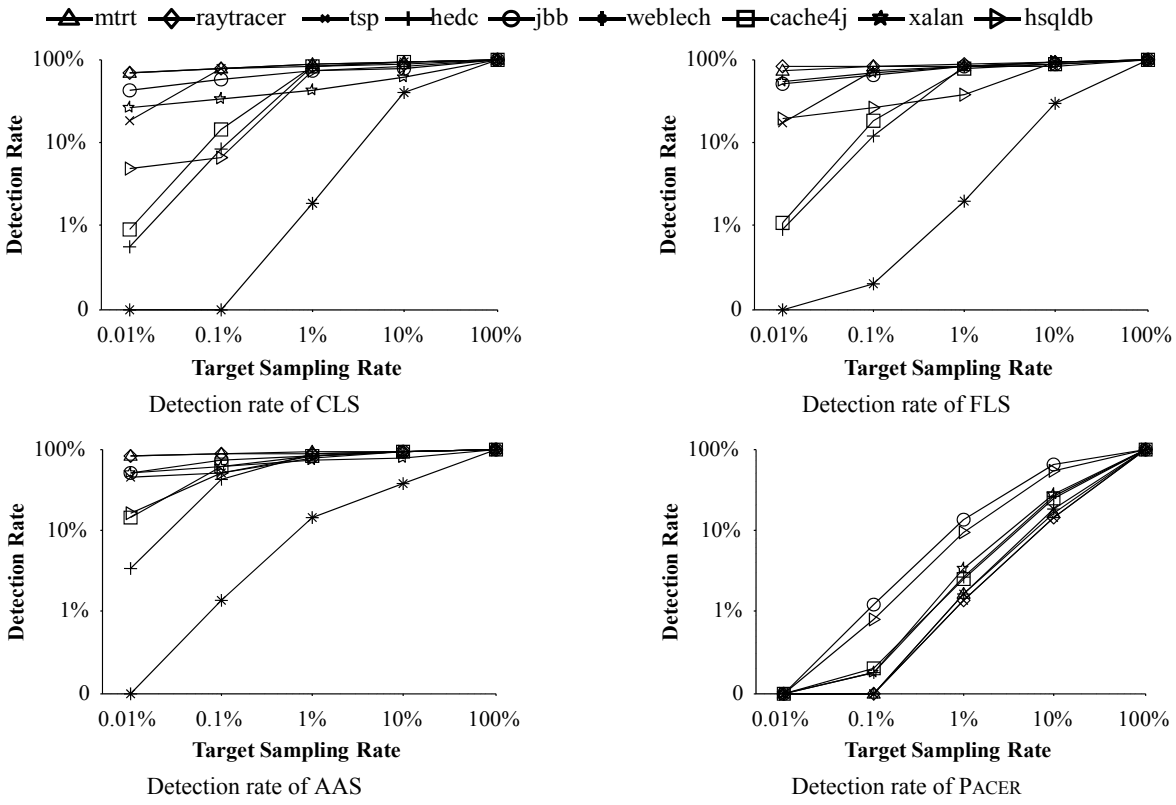

Figure 6. Detection rates of various techniques.

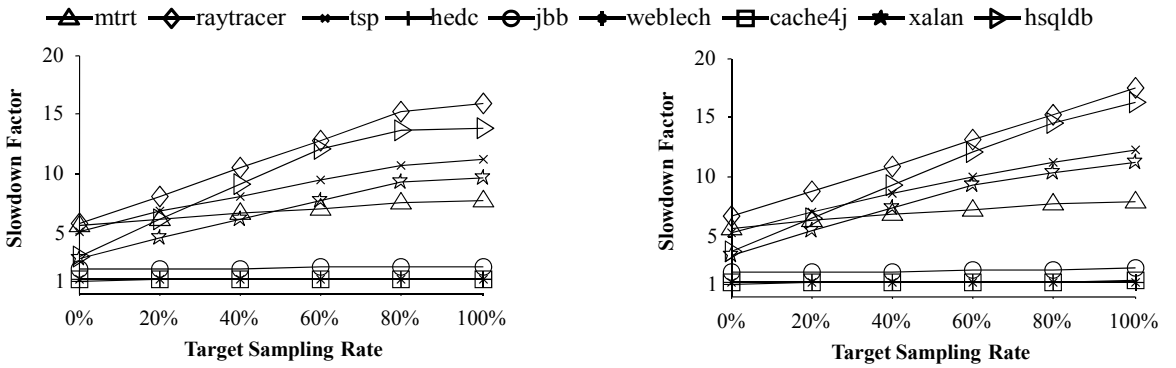

Figure 7. Slowdown vs. sampling rates for CLS.

Figure 8. Slowdown vs. sampling rates for FLS.

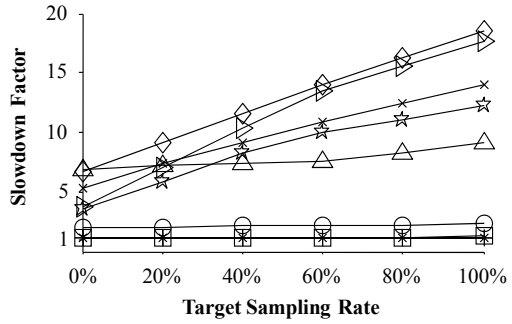

Figure 9. Slowdown vs. sampling rates for AAS. at r=0-100%

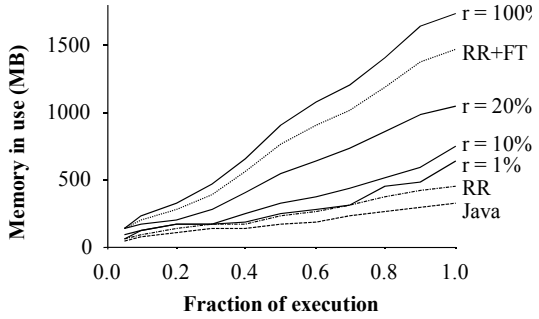

Figure 10. Memory used over one execution of xalan.

4.4 Time and Space Overheads of CARISMA

This section evaluates the time and space overheads of CARISMA.

Time overhead. Figures 7 to 9 show the slowdown incurred by the three strategies of CARISMA on each benchmark for r ranging over [0, 100%]. All figures show that the slowdowns incurred by CARISMA are roughly linear with respect to r, although the slowdown factors on different benchmarks vary a lot. We note that the slowdown factor includes the execution time of the program, the overhead incurred by the ROADRUNNER platform, the overhead incurred by additional monitoring of memory allocations, the overhead incurred by the sampling algorithms, and the overhead of FASTTRACK. Working at a target sampling rate of 1%, CARISMA incurs slowdowns by a factor of 5x on five benchmarks (mtrt, raytracer, tsp, xalan, and jbb), which are either computation-bound or relatively large in sizes, and can detect races with a relatively high probability (60%). If working at a sampling rate of 100%, CARISMA is functionally equivalent to FASTTRACK [9], which has no sampling effort. CARISMA slows down the five programs by a factor of 13.0x, compared with 8.5x by FASTTRACK. For the other programs, CARISMA incurs reasonable overheads.

Note that the mean slowdown by the original ROADRUNNER platform on compute-bound programs is 4.1x. As we have implemented CARISMA on top of ROADRUNNER, there are many internal operations and events in ROADRUNNER that cannot be eliminated by sampling. For this reason, we believe that implementing CARISMA within a Java virtual machine can improve the performance further. In fact, at a sampling rate of 1%, CARISMA can reduce 80% of the overhead incurred (from 8.5x – 4.1x = 4.4x to 5x – 4.1x = 0.9x) by the FASTTRACK tool, which is significant.

Space overhead. By design, AAS consumes the most memory among the three strategies. Figure 10 shows the memory space overhead incurred by AAS on xalan. (Due to page limit, we do not present the results on other smaller programs or the results of CLS and FLS.) The *x*-axis shows the time fraction of one execution (as we normalize the execution time to 1). For each target sampling rate shown, we take the mean overhead over all executions. The line *Java* is the memory used by executing the program directly. The line *RR* is the memory used by executing the program with the original ROADRUNNER platform. The line *RR+FT* is the memory used by executing the program with FASTTRACK on top of the original ROADRUNNER. At a sampling rate of 100%, AAS uses more memory than FASTTRACK because AAS collects dynamic information during an execution. At a sampling rate of 20%, AAS uses significantly less memory than FASTTRACK. The result shows that CARISMA can scale well with the sampling rate in terms of memory space used, and with a low sampling rate of r = 1%, its space overhead appears to be very low.

4.5 Threats to validity

The threats to validity of the experiment can be further mitigated by using more benchmarks with and without races, and by evaluating more aspects of the techniques, such as the use of different numbers of threads per run, different types of contexts, and different notions of correctness criteria (e.g., detection rate of atomicity violations). We have referred to evaluation methodologies reported in previous work and assured the quality of our evaluations. We have measured the mean performance of different techniques in terms of effectiveness as well as time and memory consumption. Measuring the results using other metrics such as variances in performance may produce different comparison results.

5. RELATED WORK

This section relates our work to that of others.

5.1 Sampling for Bug Detection and Testing

The work most relevant to CARISMA is LITERACE [23] and PACER [2]. LITERACE [23] uses a code-partitioning strategy and adaptive burst sampling to reduce sampling overheads. Bond et al. [2] have compared LITERACE and PACER, and states that PACER can be better than LITERACE. This paper has extensively compared CARISMA with PACER. Cooperative Crug Isolation (CCI) [18] samples inserted predicates according to their characteristics to infer the locations of concurrency bugs. Unlike CARISMA, these three techniques used neither cross-execution budget estimation nor inter-context budget balancing.

RACEZ [30] uses a hardware performance monitor to sample memory accesses and an (imprecise) offline lockset-based analyzer to detect races from the collected data. Although the overhead can be lower, hardware support cannot completely replace software tools for two reasons: (a) hardware is less flexible and more difficult to be adapted and applied to other situations, and (b) hardware and software techniques are generally complementary. The notion of even spreading of test efforts is also explored in adaptive random testing. See, for example, Jiang et al. [17].

5.2 Context Sensitivity

Context is a broad concept. We first review related work on its use for concurrency bug detection. Communication contexts [22] are proposed to record the ordering of communication events (memory reads and writes), so that communication edges can be effectively distinguished from one another. BREADCRUMBS [1] uses probabilistic calling contexts, which are encoded to unique identifiers and tagged to dynamic events. Such an identifier can be decoded offline to give the detailed circumstances of the events in order to facilitate debugging. MAGICFUZZER [5] uses a lightweight indexing algorithm to compute object abstraction to confirm deadlocks. CARISMA uses contexts to partition memory creation environments. Smaragdakis et al. [31] discussed the use of calling context for point-to analysis in object-oriented languages. We note that our definition of call context is unrestricted to any specific type, and calling context can also be use with CARISMA.

5.3 Data Race Detection

CARISMA is designed to work with dynamic data race detection tools, most of which are based on *locksets* [27], *happens-before relations* [20][24], or their hybrids. Lockset-based techniques [7][8][27] verify whether program executions follow *locking disciplines* [27]. They are fast and interleaving-insensitive, but imprecise. Happens-before based detectors [4][9][26][29] can precisely detect data races, and FASTTRACK [9] has shown a promise that its performance can match that of lockset-based techniques. LOFT [4] optimizes the vector clock operations on synchronization events. Hybrid techniques such as GOLDILOCKS [7] have been proposed, which often use locksets to reduce overheads and happens-before-based tracking to achieve precision. CARISMA has been tested to work well with FASTTRACK. CARISMA requires no technique-specific information. We expect that with slight adaptations, CARISMA can also work well with other dynamic race detection tools like GOLDILOCKS [7].

6. CONCLUSION

Data race is a common problem in multithreaded programs. Each data race is related to one memory location. In this paper, we have proposed CARISMA, a novel context-sensitive sampling approach to dynamic bug detection. CARISMA innovatively estimates and balances the sample budgets among contextual groups over a sequence of execution trials. It can be configured at various granularity levels. The experimental results show that CARISMA finds data races with high probability and low overheads. We also foresee that CARISMA has the potential to support program testing in general and integrate with other concurrency bug analysis tools. In the future, we will generalize the approach to deal with different types of bugs (including assertion violation, atomicity violations, and atomic-set serializability violations) and will develop new methods to further refine the approach.

7. ACKNOWLEDGEMENTS

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project numbers 111410 and 717811). All correspondence related to this paper should be addressed to Dr. W. K. Chan.

8. REFERENCES

[1] Bond, M. D., Baker, G. Z., and Guyer, S. Z. 2010. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *PLDI 2010*, 13–24.

[2] Bond, M. D., Coons, K. E., and McKinley, K. S. 2010. PACER: proportional detection of data races. In *PLDI 2010*, 255–268.

[3] Boyd, S. and Vandenberghe, L. 2004. *Convex Optimization*, Cambridge University Press.

[4] Cai, Y. and Chan, W. K. 2011. LOFT: redundant synchronization event removal for data race detection. In *ISSRE 2011*, 160–169.

[5] Cai, Y. and Chan, W. K. 2012. MagicFuzzer: scalable deadlock detection for large-scale applications. In *ICSE 2012*.

[6] The DaCapo Benchmark Suite. Available at http://dacapobench.org/.

[7] Elmas, T., Qadeer, S., and Tasiran, S. 2007. Goldilocks: a race and transaction-aware Java runtime. In *PLDI 2007*, 245–255.

[8] Engler, D. and Ashcraft, K. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP 2003*, 237–252.

[9] Flanagan, C. and Freund, S. N. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI 2009*, 121–133.

[10] Flanagan, C. and Freund, S. N. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE 2010*, 1–8.

[11] Flanagan, C., Freund, S. N., Lifshin, M., and Qadeer, S. 2008. Types for atomicity: static checking and inference for Java. *ACM TOPLAS* 30 (4), 20:1–20:53.

[12] Flanagan, C. and Godefroid, P. 2005. Dynamic partial-order reduction for model checking software. In *POPL 2005*, 110–121.

[13] Hauswirth, M. and Chilimbi, T. M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI*, 156–164.

[14] Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Race checking by context inference. In *PLDI 2004*, 1–13.

[15] Jannesari, A., Bao, K., Pankratius, V., and Tichy, W. F. 2009. Helgrind+: an efficient dynamic race detector. In *IPDPS 2009*, 1–13.

[16] The Java Grande Forum Benchmark Suite. Available at http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[17] Jiang, B., Zhang, Z., Chan, W. K., and Tse, T. H. 2009. Adaptive random test case prioritization. In *ASE 2009*, 233–244.

[18] Jin, G., Thakur, A., Liblit, B., and Lu, S. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA 2010*, 241–255.

[19] Johnson, N. L., Kemp, A. W., and Kotz, S. 2005. *Univariate Discrete Distributions*, John Wiley.

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

[21] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y. 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP 2007*, 103–116.

[22] Lucia, B. and Ceze, L. 2009. Finding concurrency bugs with context-aware communication graphs. In *MICRO 42*, 553–563.

[23] Marino, D., Musuvathi, M., and Narayanasamy, S. 2009. LiteRace: effective sampling for lightweight data-race detection. In *PLDI 2009*, 134–143.

[24] Mattern, F. 1989. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms*, 215–226.

[25] Naik, M., Aiken, A., and Whaley, J. 2006. Effective static race detection for Java. In *PLDI 2006*, 308–319.

[26] Pozniansky, E. and Schuster, A. 2003. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP 2003*, 179–190.

[27] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM TOCS* 15 (4), 391–411.

[28] Sen, K. 2007. Effective random testing of concurrent programs. In *ASE 2007*, 323–332.

[29] Sen, K. 2008. Race directed random testing of concurrent programs. In *PLDI 2008*, 11–21.

[30] Sheng, T., Vachharajani, N., Eranian, S., Hundt, R., Chen, W., and Zheng, W. 2011. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *ICSE 2011*, 401–410.

[31] Smaragdakis, Y., Bravenboer, M., and Lhoták, O. 2011. Pick your contexts well: understanding object-sensitivity. In *POPL 2011*, 17–30.

[32] *SPEC Benchmarks*, Standard Performance Evaluation Corporation. Available at http://www.spec.org/.