Promoting Secondary Orders of Event Pairs in Randomized Scheduling using a Randomized Stride

Mahmoud Abdelrasoul
North Carolina State University, USA
Email: mahmoud.a@acm.org

***Abstract*—Because of the wide use of randomized scheduling in concurrency testing research, it is important to understand randomized scheduling and its limitations. This work analyzes how randomized scheduling discovers concurrency bugs by focusing on the probabilities of the two possible orders of a pair of events. Analysis shows that the disparity between probabilities can be large for programs that encounter a large number of events during execution. Because sets of ordered event pairs define conditions for discovering concurrency bugs, this disparity can make some concurrency bugs highly unlikely. The complementary nature of the two possible orders also indicates a potential trade-off between the probability of discovering frequently-occurring and infrequently-occurring concurrency bugs. To help address this trade-off in a more balanced way, randomized-stride scheduling is proposed, where scheduling granularity for each thread is adjusted using a randomized stride calculated based on thread length. With some assumptions, strides can be calculated to allow covering the least likely event pair orders. Experiments confirm the analysis results and also suggest that randomized-stride scheduling is more effective for discovering concurrency bugs compared to the original randomized scheduling implementation, and compared to other algorithms in recent literature.**

***Index Terms*—Multithreading, software debugging, software quality, parallel programming, scheduling algorithms**

I. Introduction

The use of concurrency in a program introduces the possibility of encountering concurrency bugs, which are bugs whose occurrence depends on how concurrent threads of execution are scheduled relative to each other. Concurrency testing, which is testing that aims at discovering concurrency bugs, has been an active area of research for decades [30]. Past research explored many techniques for discovering concurrency bugs. One of these techniques is randomized scheduling [41], where a thread is selected for execution randomly at each step until program execution is complete.

Research on randomized scheduling remains active. Variations and extensions of randomized scheduling have been studied in recent literature (e.g. [5], [6], [11], [23], [33], [45]). Randomized scheduling also serves as a building block of some other techniques, such as in [22], [28], [43], [51], [53], as a way to explore a program before applying a different algorithm that requires some knowledge of the program. Therefore, it is important to have a good understanding of how randomized scheduling can effectively explore the space of possible thread schedules of a concurrent program.

Despite the wide use of randomized scheduling in concurrency testing research, some aspects of randomized scheduling are not readily explicable. For example, inspecting the results by Thomson et al. [48] suggests that the effectiveness of randomized scheduling for discovering concurrency bugs tends to be worse for programs that encounter a larger number of events during execution. It is not immediately clear from information found in past research why the effectiveness of randomized scheduling would deteriorate for programs that encounter more events.

To help reach a better understanding of how randomized scheduling discovers concurrency bugs, this work focuses on the ability of randomized scheduling to cover the two possible orders of a pair of events. The more-likely order is referred to as the *primary order*, and the less-likely order is referred to as the *secondary order*. With this focus, this work makes the following contributions:

- The work performs a theoretical analysis demonstrating an inherent limitation of the ability of randomized scheduling to discover concurrency bugs in large programs, by showing that the number of secondary orders coverable by randomized scheduling with sufficiently high probability is, at best, inversely proportional to the square root of the total number of events encountered during program execution, given any constant ratio between thread lengths.
- Randomized-stride scheduling is developed as a way to modify randomized scheduling to improve the chances of discovering infrequently-occurring concurrency bugs by improving the probability of covering secondary orders.
- The work sheds light on the potential trade-off between the frequency of discovering frequently-occurring bugs and infrequently-occurring bugs that is suggested by the complementary nature of primary and secondary orders.
- Experiments are performed and confirm the results of the analysis and also show that randomized-stride scheduling is more effective than related techniques described in recent literature. Analysis of the results also shows some limitations in these techniques that are avoided by randomized-stride scheduling.

II. Background

A. Randomized Scheduling

A number of variations of randomized scheduling were proposed in concurrency testing literature. This work specifically considers the variation of randomized scheduling represented in the tool Maple [52] as available on [1]. This randomized

Accepted for publication by IEEE. © 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

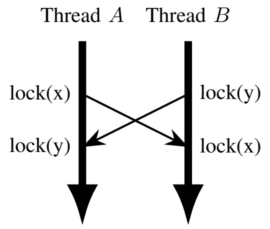

Fig. 1. The necessary conditions for a deadlock represented as a set of ordering edges, as also shown in a similar illustration by Burckhardt et al. [5]

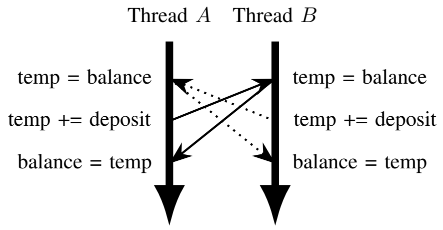

Fig. 2. Two sets of ordering edges representing the conditions needed to expose an atomicity violation bug. One set of ordering edges is shown in continuous lines, while the other is shown in dotted lines. Covering either one of the two sets is sufficient to expose the bug.

scheduling algorithm is also the same as the one that was implemented previously in the tool CalFuzzer [21], which was relied on in other tools (e.g. [22], [28], [36], [43]).

In this implementation of randomized scheduling, the program under test is executed by selecting a thread randomly out of the set of enabled threads, with equal probability of selecting any thread from this set. The selected thread is executed up to the next event, where events are actions by the program that can result in intentional or unintentional communication between threads[1]. Once that event is reached, random thread selection is performed again, selecting any enabled thread, including the most-recently-executed thread if enabled, with equal probability. This process is repeated until the program completes execution or until the program has no enabled threads, which completes one run of the program under randomized scheduling. Testing the program under randomized scheduling requires running the program many times following the same process, keeping random choices in every run independent from those in other runs, with the idea that the randomization can result in exploring variations of thread schedules that uncover concurrency bugs.

As is typical in other work in the literature on randomized scheduling, this work assumes that the inputs to the program are predetermined, for example by existing test cases, and do not vary between runs. For practical purposes, if multiple variations of inputs need to be used during testing, the program can be tested separately for each variation as if each variation defines a distinct program to be tested.

B. *Ordering Edges*

A key concept that is used in this work to reason about the ability to discover concurrency bugs is the concept of an ordered pair of events. The use of ordered pairs of events to describe the conditions necessary for discovering concurrency bugs was introduced by Burckhardt et al. [5]. In that work, ordered pairs of events were referred to as "ordering edges", which is the same term that will be used here. Conceptually, events represent instances of execution of atomic actions performed by the program. Multiple instances of the same action in the same execution must be identified each by a distinct event.

An ordering edge represents that the first event of the ordered pair occurs before the second event in the instruction interleaving that represents the program execution under randomized scheduling. Sets of ordering edges are sufficient to describe ordering constraints that can expose any concurrency bug. In fact, as may also have been implied by Nagarakatte et al. [33], any instruction interleaving representing the execution of a program under randomized scheduling can be completely described by a set of ordering edges sufficient to define a total order of the events in this interleaving. For example, Fig. 1 shows how the necessary conditions for a deadlock can be described by ordering edges. Moreover, as in the example in Fig. 2, some bugs may show when any set out of a number of different sets of ordering edges is satisfied. Other examples of how ordering edges can be used to represent the necessary conditions for bugs are presented by Burckhardt et al. [5] and Cai and Yang [6].

To represent an ordering edge that specifies that event U shows before event V in an interleaving that represents the program execution, the notation $U \rightarrow V$ will be used, as was used once in the paper by Burckhardt et al. [5]. Because randomized scheduling, as described earlier, executes instructions from only one thread at a time, any run of the program under randomized scheduling will execute exactly one of the two possible orders of any two events encountered during that run. When an ordering edge is exercised during an execution of the program, the execution of the program will be said to have *covered* the ordering edge.

III. Analysis of Randomized Scheduling

A. *An Idealized Case*

A key part of the analysis in this work focuses on the relative order of two events, A_a and B_b, that happen on two different threads, Thread A and Thread B, in an idealized program. The analysis also refers to a common predecessor event, E_0. E_0 is the farthest event from the beginning of the program that is known by the structure of the program to always precede both A_a and B_b. The subscript a in A_a denotes the number of events in Thread A after E_0 is executed, up to and including A_a. Similarly, b is the number of events in Thread B after E_0 is executed, up to and including B_b. Without loss of generality,

[1]In a single-process multi-threaded program that depends on shared-memory communication, such events include shared memory accesses, synchronization operations, and thread creation and join operations.

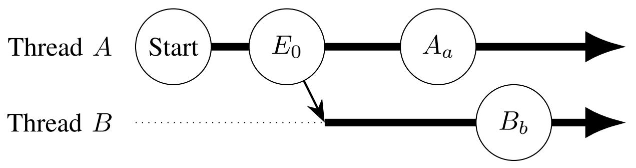

Fig. 3. Illustration of key elements considered in the theoretical analysis

it will be assumed that $a \leq b$. The common predecessor E_0 may be the first event in the program, or can be a later event (e.g. where a thread is instantiated, or where blocking communication happens). Fig. 3 illustrates some of the key elements of the idealized program considered in the theoretical analysis in this work[2].

To enable the analysis, the following simplifying assumptions will be used:

- Neither of the two threads becomes blocked (i.e. the threads will always be enabled) after E_0 is executed.
- The sequence of events executed by each of the two threads individually does not change between runs.

The first assumption simplifies the analysis because it implies that the progress of each of the two threads is unaffected by the progress of any thread other than itself. The assumption also implies that any interleaving of events is feasible if it preserves the order of events after E_0 within Thread A and within Thread B. With this assumption, the model becomes an idealization because it does not account for effects of blocking communication involving either of the two threads after E_0.

The second assumption is needed to allow each event to be assigned an ordinal label that consistently identifies the event across different executions of the program. Therefore, with the second assumption, A_a and B_b refer to the same events in all possible executions of the program. This assumption also makes the model an idealization because it results in the model not accounting for the possibility of variation of code paths executed by a thread depending on communication with other threads. The assumption also implies that the relative order of instructions within a thread does not change between runs, which suggests sequential consistency.

B. Time to Reach an Event

The durations of time that elapse from the end of execution of E_0 to the beginning of execution of A_a and B_b are two random variables T_{A_a} and T_{B_b}, respectively. Time is measured in units of time blocks. A time block is defined as the period of time that starts when the randomized scheduling algorithm selects Thread A for execution, and ends with the next instance when Thread A is selected again for execution. Based on this definition, exactly one event of Thread A is executed in each time block. In essence, the definition of the time block allows describing the progress of Thread B using the most recent event that has been executed from Thread A as a frame of reference. By this definition, T_{A_a} always equals a.

This definition of the time block preserves the ability to determine the relative order of execution of two events, each in a different thread, by comparing the time taken to reach each event. $A_a \rightarrow B_b$ occurs if and only if $T_{A_a} \leq T_{B_b}$, which happens with probability $P(T_{A_a} \leq T_{B_b})$, whereas $B_b \rightarrow A_a$ occurs if and only if $T_{A_a} > T_{B_b}$, which happens with probability $P(T_{A_a} > T_{B_b}) = 1 - P(T_{A_a} \leq T_{B_b})$.

C. Primary and Secondary Orders of Event Pairs

Per the discussion above, the two events A_a and B_b will define two possible ordering edges that are mutually exclusive: $A_a \rightarrow B_b$ and $B_b \rightarrow A_a$. The ordering edge that has the higher probability will be referred to as the *primary order* of the two events, while the ordering edge of the lower probability will be referred to as the *secondary order*. If both orders have the same probability, either order can be selected to be the primary order while the other becomes the secondary order.

Because randomized scheduling treats all threads equally, the only property of A_a and B_b that can affect which order of the two events is the primary order and which is the secondary order is the number of events until each of A_a and B_b are reached, i.e. a and b. Because $a \leq b$, $A_a \rightarrow B_b$ is the primary order of the two events, and $B_b \rightarrow A_a$ is the secondary order.

D. Probability of an Ordering Edge

As described earlier, at each event of the program, the randomized scheduling algorithm makes a decision for selecting the next thread to execute from any of Thread A, Thread B or any other enabled threads, with equal probability for all choices. Selecting events that are not in Thread A or Thread B does not affect the number of time blocks for execution to reach A_a or B_b. Therefore, events in other threads are not relevant for this discussion and can be ignored. Hence, the problem of analyzing randomized scheduling can be correctly reduced to an analysis that assumes that the next event to execute is selected from either Thread A or Thread B with an equal probability of $p = 0.5$ each, regardless of the number of threads in the program.

Because thread selection at each step is independent from previous selections, the amount of time taken until the occurrence of the next event from Thread B, to be referred to as τ_B, is a geometric random variable. Using the formulas in [46], τ_B has a mean of $\mu_{\tau_B} = (1-p)/p = 1$ and variance of $\sigma^2_{\tau_B} = (1-p)/p^2 = 2$. After b events from Thread B, the total amount of execution time from E_0 to B_b (i.e. T_{B_b}) is $\sum_{i=1}^{b} \tau_B$. If b is large, the Central Limit Theorem as stated in [46] implies that T_{B_b} can be approximated by a normally-distributed random variable of mean and variance equal to b times the mean and variance, respectively, of τ_B. Therefore, the mean and variance of T_{B_b} are:

$$\mu_{T_{B_b}} = b \tag{1}$$

$$\sigma^2_{T_{B_b}} = 2b \tag{2}$$

[2]For simplicity of exposition, it will be assumed that the sequence of instructions between E_0 and A_a (and, similarly, between E_0 and B_b) are all on the same thread. However, the analysis can directly be generalized to the case where the sequence of instructions is on multiple threads with the mathematical analysis and the results remaining the same.

Because, by the definition of the time block as stated earlier, $T_{A_a} = a$, the probability of the secondary order is $P(B_b \to A_a) = P(T_{A_a} > T_{B_b}) = P(T_{B_b} < a)$, which results in:

$$P(B_b \to A_a) \approx \Phi\left(\frac{a - \mu_{T_{B_b}}}{\sigma_{T_{B_b}}}\right)$$

where Φ is the cumulative distribution function of the standard normal distribution. By substituting the mean and the variance of T_{B_b} from Equations (1) and (2):

$$P(B_b \to A_a) \approx \Phi\left(\frac{a-b}{\sqrt{2b}}\right) \tag{3}$$

E. Effect of Thread Length on Ability to Cover Secondary Orders

Equation (3) suggests that, given a probability that can be considered sufficiently high for an ordering edge to be coverable by randomized scheduling, the ratio of secondary orders that can be discovered with a sufficiently high probability diminishes when the magnitude of $a - b$ becomes larger. This subsection describes this conclusion and the reasoning behind it in more precise terms.

Let p be the minimum acceptable probability of an ordering edge. The probability p can be determined in part based on the number of program runs planned under randomized scheduling, as done in the experiments presented later in this work. If the program will be run many times under randomized scheduling, it is reasonable to assume that this minimum acceptable probability is small. Hence, it can be assumed that $p < 0.5$, which implies that all ordering edges that do not meet this minimum acceptable probability represent secondary orders. Based on Equation (3), the following inequality needs to hold true for the ordering edge $B_b \to A_a$ (which is the secondary order of A_a and B_b) to be sufficiently likely:

$$\Phi\left(\frac{a-b}{\sqrt{2b}}\right) \geq p$$

That is:

$$\frac{a-b}{\sqrt{b}} \geq \sqrt{2} \cdot \Phi^{-1}(p)$$

where Φ^{-1} is the inverse of the cumulative distribution function of the standard normal distribution. The right-hand side of this inequality is a constant that depends only on the predetermined desired probability p. The constant is negative, because $p < 0.5$. If we refer to this constant as $-\Lambda$, the inequality can be written as:

$$a \geq b - \Lambda \cdot \sqrt{b} \tag{4}$$

The lengths of threads A and B (i.e. the number of events in threads A and B after E_0) will be referred to as l_A and l_B, respectively. Because $a \leq b$, it follows that $a \in [1, \min(l_A, b)]$ and $b \in [1, l_B]$ where a and b are integers. Equation (4) implies that a must be an integer in the interval $[b - \Lambda \cdot \sqrt{b}, \min(l_A, b)]$ so that $P(B_b \to A_a) \geq p$. The length of this interval is at most $\Lambda \cdot \sqrt{b}$. Because the upper limit of the interval is an integer, the maximum number of distinct possible values of a is at most $\Lambda \cdot \sqrt{b}$.

Proposition 1: Let $l_B = cl_A$, where c is constant, and where l_B and l_A are sufficiently large to make negligible the number of events for which the Central Limit Theorem approximation in Equation (3) is invalid. The ratio of secondary orders coverable by randomized scheduling with probability at least p to the total number of secondary orders is bounded by $O(\sqrt{l_A + l_B})$ (i.e. is at most inversely proportional to the square root of the total number of events that occur after E_0 in threads A and B).

Proof: Let $B_j \to A_i$ be an ordering edge representing a secondary order of the ith event in thread A and the jth event in thread B. As stated earlier, the number of possible values of i so that $P(B_j \to A_i) \geq p$ is at most $\Lambda \cdot \sqrt{j}$. Therefore, given a value for j, the maximum possible number of event pairs for which the secondary order is covered by randomized scheduling with a probability of at least p is $\Lambda \cdot \sqrt{j}$.

Considering that j can be any integer between 1 and l_B inclusive, the number of pairs for which the secondary order has a probability greater than or equal to p is at most $\sum_{j=1}^{l_B} \left(\Lambda\sqrt{j}\right)$. Using the result from [38], it can be found that the most dominant term of this summation is $\frac{2\Lambda}{3} \cdot l_B^{3/2}$, and the value of this summation is in the order of $\Theta(l_B^{3/2})$. As mentioned earlier, this is only a best case that assumes the largest possible range for the possible values of i. Therefore, the number of secondary orders $B_j \to A_i$ that can be covered with probability at least p is $O(l_B^{3/2})$. Given that $l_B = cl_A$ and c is a constant, the number of such secondary orders is $O(l_A^{3/2})$.

Similarly, the number of secondary orders $A_i \to B_j$ that can be covered with probability at least p is also $O(l_A^{3/2})$.

The total number of secondary orders $A_i \to B_j$ and $B_j \to A_i$ is equal to $l_A l_B = cl_A^2$. Therefore, the ratio of the number of secondary orders that have a probability of at least p to the total number of all secondary orders is $[O(l_A^{3/2}) + O(l_A^{3/2})]/cl_A^2 = O(1/\sqrt{l_A})$.

Given that $l_B = cl_A$, it follows that $l_A = (l_A + l_B)/(c+1)$. Because $(c+1)$ is constant, the ratio of the number of secondary orders with a probability of at least p to the total number of all secondary orders is $O(1/\sqrt{l_A + l_B})$.[3] ■

This result shows that the effectiveness of randomized scheduling deteriorates when the program grows larger, because a large proportion of ordering edges becomes unlikely to be covered, leaving a large proportion of the possible sets of ordering edges unexplored. Because discovering ordering edges is necessary for discovering potential bugs for which these ordering edges are part of the necessary conditions, this result may explain the adverse effect that a large number of events in a program can have on the ability of randomized scheduling to discover bugs as discussed in the introduction.

[3]This result can be extended to the case where there are more than two threads where all pairs of threads conform to the assumptions stated earlier on a pairwise basis. It can be shown in this case that the number of secondary orders discoverable with sufficient probability is $O(1/\sqrt{\sum_i l_i})$, assuming the ratio between lengths of each two threads remains constant (i.e. $\forall i, j \ l_i = c_{ij} l_j$, where c_{ij} is constant). This result can be proved by considering each possible pair of threads individually, and aggregating all the results.

IV. Improving Randomized Scheduling

The analysis in the previous section suggests that improving the probability of covering secondary orders can help improve randomized scheduling. However, any potential improvement to the probability of discovering these ordering edges is expected to require a trade-off. Because the primary order of an event pair is mutually exclusive with its corresponding secondary order, any change to randomized scheduling that can increase the probability of covering secondary orders will result in reducing the probability of covering the corresponding primary orders. Therefore, any change that increases the probability of discovering potential bugs for which secondary orders are necessary conditions will inevitably result in decreasing the probability of some potential bugs for which primary orders are necessary conditions. A modification to randomized scheduling that attempts to increase the probability of secondary orders should try to avoid severely reducing the probability of covering primary orders.

The analysis also suggests that coarser scheduling can improve the ability to cover secondary orders that are not sufficiently likely to show when finer-grained scheduling is used. The analysis connected the low probability of discovering secondary orders to the large number of time blocks that are needed to reach an instruction. One way to make randomized scheduling more effective for programs that encounter more events during execution is to reduce the number of time blocks needed to reach events such as A_a and B_b. Considering that the definition of a time block depends on how many times Thread A is selected for execution, the probability of encountering secondary orders can improve if scheduling is made coarser such that multiple events from the same thread are covered each time the thread is selected for execution.

To make scheduling coarser, an integer s, the *stride*, will be selected with uniform probability between 1 and s_{max}, inclusive, at every point where randomized scheduling selects a thread for execution. The thread that is selected for execution at that point will be executed exclusively until s events have been encountered or until the thread becomes blocked or completes execution. The value s_{max} will be referred to as the *maximum stride*. If s_{max} is set to 1, the scheduling algorithm becomes identical to the original randomized scheduling algorithm analyzed earlier. If s_{max} is greater than 1, the scheduling algorithm will be referred to as *randomized-stride scheduling*.

For selecting s_{max}, although a larger value is desirable because it increases the chance of encountering unlikely ordering edges, a larger s_{max} also has a disadvantage because it reduces the number of thread switches that are expected to happen during a run of the program, and therefore makes it less likely to discover bugs whose discovery requires covering a combination of several ordering edges. Therefore, it is desirable to use the smallest possible s_{max} that does not make any secondary orders too unlikely to be covered.

When considering two threads A and B in a program being run under randomized scheduling, the most unlikely ordering edge originating from thread A is the ordering edge $A_l \rightarrow B_1$ where A_l is the last event in thread A and B_1 is the first event in thread B. This ordering edge is unlikely because it requires that scheduling decisions never result in selecting thread B until thread A completes execution. A longer stride improves the probability of discovering this ordering edge, so the scheduling algorithm should choose a stride that is long enough to make $P\left(A_l \rightarrow B_1\right)$ just as high as necessary. This probability can be calculated as follows:

$$P(A_l \rightarrow B_1) = \sum_{i=1}^{\infty} 0.5^i P\left(\left(\sum_{j=1}^{i} U_d\left(1, s_{max}\right)\right) \geq l\right)$$

where $U_d(1, s_{max})$ is the discrete uniform distribution for which the possible values are the integers between 1 and s_{max}, inclusive. This expression can be approximated by using the continuous uniform distribution, which results in:

$$P(A_l \rightarrow B_1) = \sum_{i=1}^{\infty} 0.5^i P\left(\left(\sum_{j=1}^{i} U_c\left(0, s_{max}\right)\right) + \zeta \geq l\right)$$

where $U_c(0, s_{max})$ is the continuous uniform distribution with the range $[1, s_{max}]$, and $\zeta = O(i)$. Dividing both sides by s_{max} yields:

$$P(A_l \rightarrow B_1) = \sum_{i=1}^{\infty} 0.5^i P\left(\left(\sum_{j=1}^{i} U_c\left(0, 1\right)\right) + \frac{\zeta}{s_{max}} \geq \frac{l}{s_{max}}\right)$$

To enable calculating s_{max} based on $P(A_l \rightarrow B_1)$ using this formula, some approximation can be helpful. Approximations taken towards calculating s_{max} should try to avoid making the probability less than the targeted probability, to help avoid making the less-likely ordering edges too unlikely to be encountered during testing.

The term ζ/s_{max} can be dropped without making the probability of the left-hand side lower than the target. Moreover, this term will have a negligible value for values of i that are small compared to s_{max}, which is expected to be large for large programs, and the multiplier 0.5^i will make the corresponding term of the outer summation negligibly small for large values of i. This results in:

$$P(A_l \rightarrow B_1) \approx \sum_{i=1}^{\infty} 0.5^i P\left(\left(\sum_{j=1}^{i} U_c\left(0, 1\right)\right) \geq \frac{l}{s_{max}}\right) \quad (5)$$

This approximation allows calculating s_{max} on a per-thread basis, based only on the length of the thread and the desired probability of covering the least likely ordering edge. Moreover, given a desired probability, the approximation makes the ratio of the length of the thread to the ideal maximum stride, i.e. l/s_{max}, constant for all threads in the program under test.

Using Equation (5), it is possible to estimate l/s_{max} based on $P(A_l \rightarrow B_1)$ with the help of Monte-Carlo sampling. Fig. 4 shows the relation between l/s_{max} and $P(A_l \rightarrow B_1)$ estimated this way. For the experiments in this work, only two values of the probability needed to be used, so l/s_{max} was estimated directly based on Monte-Carlo sampling. However, Fig. 4 suggests that the relation between l/s_{max} and $P(A_l \rightarrow B_1)$ can potentially be approximated by an exponential relation.

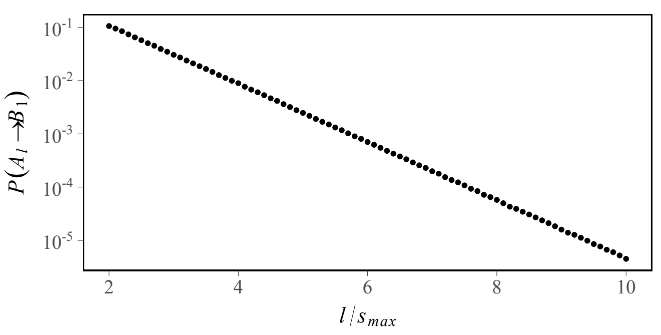

Fig. 4. The probability of covering the ordering edge $A_l \rightarrow B_1$ versus the ratio of thread length to the maximum stride length, based on the approximation in Equation (5)

V. Experiment

A. Experiment Setup

To test the prediction that coarser scheduling can help improve the ability to discover concurrency bugs that are infrequently-occurring under the original randomized scheduling, an experiment was performed using 41 out of the 52 test programs available from the work by Thomson et al. [48] as the test subjects. The programs are available for download from the web along with a virtual machine that is set up to run the programs [1]. 9 of the 11 excluded programs were failing all the time and were reported by Thomson et al. to fail all the time, suggesting that the bugs in these programs are not concurrency bugs or at least are not interesting for the purposes of this work. The two other programs that were excluded are parsec.ferret and parsec.streamcluster2. When trying to reproduce the results by Thomson et al., the results of these two programs were inconsistent with what was reported. The test subject parsec.ferret failed all the time, while parsec.streamcluster2 never failed. Inspecting the code of parsec.streamcluster2 did not reveal a concurrency bug.

For the programs that were used in the experiment, there were also variations of the results from those reported by Thomson et al. Nearly all of these variations can be expected because of the randomness intrinsic to randomized scheduling. For a few programs, the differences appeared to be larger than what can be explained by randomness. Therefore, the results of running the original randomized scheduling in this experiment are reported in https://randomizedstride.wordpress.com/.

The programs used in this work included real-world programs with real-world bugs, and also artificial programs. The programs were drawn by Thomson et al. from various sources. As can be seen in the next section, the bugs in the programs also present a wide range of frequencies of occurrence under randomized scheduling, and the programs also cover a wide range of the number of events encountered during program execution.

To explore the effect of changing s_{max}, three variations of randomized-stride scheduling were used. The first variation used $s_{max} = 2$, which is the smallest possible s_{max} that results in coarser scheduling than the original randomized scheduling. The second and third variations used a value of s_{max} that is calculated based on the thread length as described in Section IV. The second variation used a ratio l/s_{max} that enables discovering the least-likely ordering edge with a confidence of 95% in at least one run. The third variation used a ratio l/s_{max} that enables discovering any two independent ordering edges in the same run with a confidence of 95%.

The program was executed for a total of 10,000 times under each of the three variations of randomized-stride scheduling. For the second and third variations, the first 10 runs were performed using $s_{max} = 1$. The maximum length of each thread in the 10 runs was used as the estimate of the thread length. Then, the program was executed 9,990 times using the maximum stride lengths calculated based on the estimated thread lengths. The value of l/s_{max} that can achieve the desired confidence in 9,990 runs of randomized scheduling is approximately 6.6 for the second variation of randomized-stride scheduling, and approximately 3.4 for the third variation. Therefore, during the experiment, the maximum stride for the second and third variations of randomized-stride scheduling was set as the ceiling of $l/6.6$ and $l/3.4$, respectively. For the purpose of counting events in any thread, the only events that were counted were those that implied potential intentional or unintentional communication between threads as described in Section II-A and that happened while other threads were enabled.

B. Baselines for Comparison

For comparison, the same test subjects were executed under the original randomized scheduling algorithm, and also under the randomized scheduling algorithms PCT that is described by Burckhardt et al. [5] and RPro that is described by Cai and Yang [6]. In addition, the results by Thomson et al. [48] were used to extend the comparison to some algorithms that do not depend on randomization, namely iterative-preemption bounding, iterative-delay bounding, depth-first search and the Maple heuristic.

For the original randomized scheduling algorithm, the program was simply run 10,000 times under the algorithm. The seed value for pseudorandom number generation in the original implementation was modified to be based on a finer-grained clock to help avoid unintended redundancy between runs.

For the PCT algorithm, the algorithm depends on a "depth" parameter, d, and targets bugs of depth d. When the depth of the targeted bugs is unknown, the ideal depth to use for the algorithm is also unknown. However, based on a formula by Burckhardt et al. [5], larger values of d result in an exponentially smaller guaranteed probability for discovering bugs. Therefore, similar to how the length-to-maximum-stride ratio was determined for randomized-stride scheduling, the formula in [5] was used to calculate the maximum depth that allows the guaranteed probability of discovering a bug to be sufficiently high. Specifically, the depth was calculated such that it allows a confidence of at least 95% that a bug with this depth will be encountered in at least one of the program runs under PCT. Similar to randomized-stride scheduling, the first 10 runs used the original randomized scheduling algorithm to

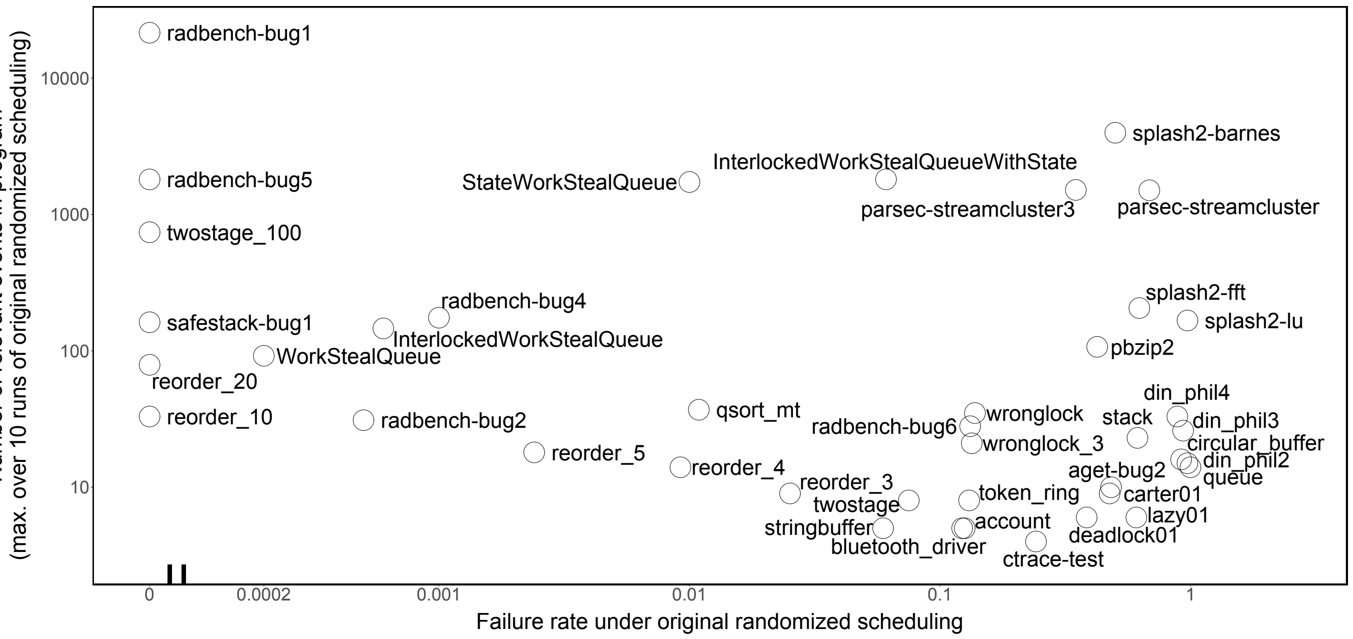

Fig. 5. The results of running the experiment subjects 10,000 times under the original randomized scheduling. The annotation on each point indicates the name of the test subject that it represents. The results in this figure are used in Fig. 6 as a baseline for comparison of other algorithms.

estimate the number of events in the program, then the PCT algorithm with the parameters based on this estimate was used for 9,990 runs.

For large programs, the depth parameter of the PCT algorithm calculated as described above becomes very small. The only variation of PCT in the literature that allows reaching a similar probability guarantee at a greater depth is the RPro algorithm by Cai and Yang [6]. Although the RPro algorithm was developed to target deadlock bugs, the algorithm does not exclude other kinds of concurrency bugs, so it was relevant for use in this experiment.

The RPro algorithm uses a parameter r, the radius of the bug, in addition to the depth parameter d. Two variations of the algorithm were used: one with radius $r = 10$ and one with radius $r = 50$, which are the same radii used by Cai and Yang. As in Cai and Yang's work, the two variations will be referred to as "RPro$_{10}$" and "RPro$_{50}$". To calculate the depth to use for the RPro algorithm, the same approach that was used for PCT was also used for RPro.

The PCT and RPro algorithms were implemented as described in their respective papers, with minor changes to handle special cases presented by the experiment subjects and to deal with ambiguity in the descriptions of the algorithms. Ambiguities were resolved in a way that should not adversely affect the ability of these algorithms to discover concurrency bugs. The details of the changes to the algorithms are described in https://randomizedstride.wordpress.com/.

VI. Experiment Results

As discussed in Section IV, reasonable increases of the maximum stride are expected to improve the ability to discover many infrequently-occurring concurrency bugs, and should also enable discovering some concurrency bugs that were not discovered by the original randomized scheduling. At the same time, because increasing the probability of discovering secondary orders will necessarily decrease the probability of discovering the corresponding primary orders, the frequency of discovering some concurrency bugs can decrease, especially for those bugs that are encountered frequently with the original randomized scheduling.

To enable visualizing the results of the experiment using the original randomized scheduling as a baseline, Fig. 5 shows the results of running the original randomized scheduling on each test subject. Each test subject corresponds to one point in the figure, located according to how frequently the bug in the test subject was found under the original randomized scheduling and how many events were encountered during program execution. Hence, test subjects represented by points towards the left side of Fig. 5 are those with bugs that were discovered rarely or not discovered at all by the original randomized scheduling. For example, the topmost leftmost point in the figure represents radbench-bug1, which is the program with the largest number of events and also one of the programs where the bug was not discovered by the original randomized scheduling.

Using Fig. 5 as a baseline, the left-hand side panels of Fig. 6 illustrate how the frequency of discovering bugs using the three variations of randomized-stride scheduling changes compared to the frequency of discovering the same bugs using the original randomized scheduling. As expected from the analysis in previous sections, the three variations of randomized-stride scheduling all show significant improvement of the ability to discover infrequently-occurring concurrency bugs relative to the original randomized scheduling, as indicated by

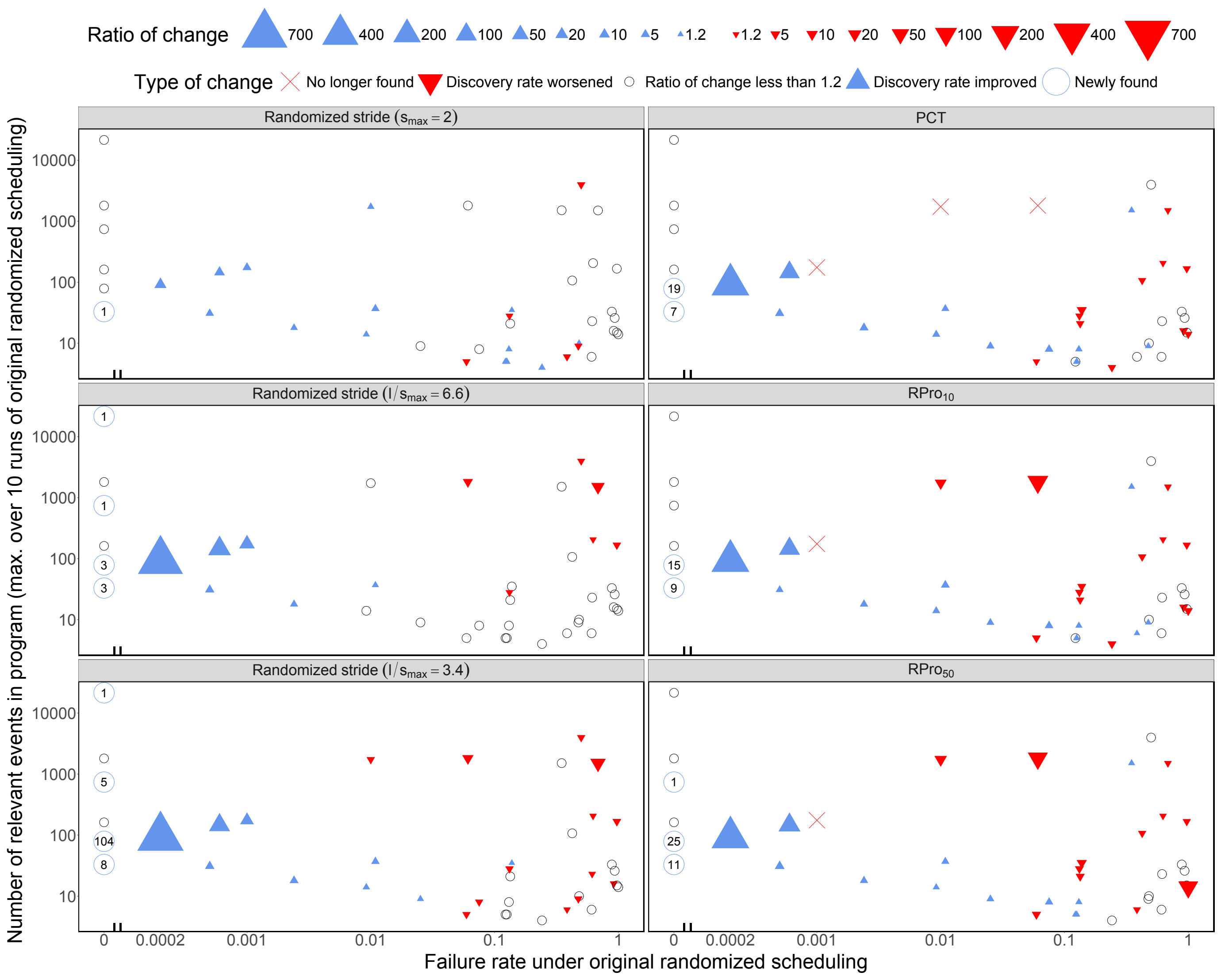

Fig. 6. An illustration comparing the rate of discovering bugs when testing the experiment subjects using randomized-stride scheduling and other techniques, using the results of original randomized scheduling in Fig. 5 as a baseline. Points in each panel have the same layout as in Fig. 5. Numbers inside circles for "Newly found" bugs state the number of times a bug was found for bugs that were not discovered by the original randomized scheduling.

points at the left of each panel. They also show a decrease in the frequency of discovering frequently-occurring concurrency bugs, corresponding to points at the right of each panel, but these bugs are still discovered fairly frequently. This pattern of increase and decrease in frequency of bug discovery shows even for the smallest increase in maximum stride ($s_{max} = 2$), and is more amplified when the maximum stride increases. Moreover, the two variations of randomized-stride scheduling where the maximum stride is calculated based on the length of the thread show better ability to discover more of the bugs that were undiscovered by the original randomized scheduling.

Inspecting the code around the additional bugs that were found by randomized-stride scheduling reveals that the newly-discovered bugs were indeed discovered because of the boost in the probability of discovering less-probable ordering edges. The four additional bugs that were discovered by randomized-stride scheduling follow two different patterns. In the case of reorder_10, reorder_20 and twostage_100, the occurrence of the bugs requires, in part, that the main thread of the program completes the creation of a large number of threads (10, 11 and 100, respectively) before any of the created threads reaches a point in execution that will make the bug undiscoverable. A thread schedule that can allow this condition is highly unlikely under the original randomized scheduling, but a lot more likely under randomized-stride scheduling because the maximum stride associated with the main thread is sufficiently large to give it a chance to make sufficient progress before other threads make too much progress. The other additional bug discovered by randomized-stride scheduling, radbench-bug1, shows a different pattern. It is a much simpler bug, but its discovery requires, in part, that two threads of drastically different lengths reach points close to the end of their execution at approximately the same time. Because the maximum stride is proportional to the thread length, randomized-stride scheduling enables such a schedule. On the other hand, with the original randomized scheduling, a large difference in thread lengths will result in the shorter thread practically always reaching the end of its execution a lot earlier than the longer thread, which will prevent the bug from showing.

To compare the overall effectiveness of the algorithms used in the experiment, Fig. 7 compares the number of bugs missed by the algorithms that were used in this work and also by the algorithms that were used in Thomson et al.'s work. As can be seen from the figure, the variations of randomized-stride scheduling with a thread-length-based maximum stride are the most effective compared to all techniques that were considered in this work and in Thomson et al.'s work.

Closer inspection suggests that one key reason why PCT and RPro missed some bugs that were discovered by randomized-stride scheduling is the limitation on the depth parameter, which was necessary to keep the guarantees presented by these algorithms as high as targeted. Because the maximum depth depends on the number of events in program execution, the maximum depth used with a program with many events needs to be small to allow achieving these guarantees. RPro allows the depth to be larger, and the results show that it successfully discovers some of the bugs missed by PCT. However, it still does not discover some of the bugs discovered by randomized-stride scheduling. For example, the experiment subject radbench-bug4 is still missed by RPro because of the depth limitation. The bug in radbench-bug4 has a depth of 3, but the maximum depth that could be used both for the PCT and RPro algorithms for this program while satisfying the targeted guarantee was 2. Compared to PCT and RPro, randomized-stride scheduling does not present the same limitation on the depth of discoverable bugs, which allows randomized-stride scheduling to discover bugs that are not discovered by PCT and RPro.

Fig. 6 provides more information about how PCT and RPro compare to randomized-stride scheduling. As can be seen in the figure, although randomized-stride scheduling shows improvement of the rate of discovering infrequently-occurring bugs at the expense of the rate of discovering frequently-occurring bugs, the other algorithms do not show this pattern of improvement with the same consistency. Moreover, some of the bugs that were missed or encountered very infrequently by PCT and RPro were bugs that were encountered relatively frequently under the original randomized scheduling algorithm. This suggests that one advantage of randomized-stride scheduling is that it results in a more balanced trade-off between the rate of discovering frequently-occurring bugs and infrequently-occurring bugs.

The results also showed that the original randomized scheduling algorithm discovered exactly the same set of bugs in this experiment as the set of bugs discovered by iterative delay bounding in the experiment by Thomson et al. [48], in addition to one bug discovered only by the original randomized scheduling. Based on the model discussed in earlier sections,

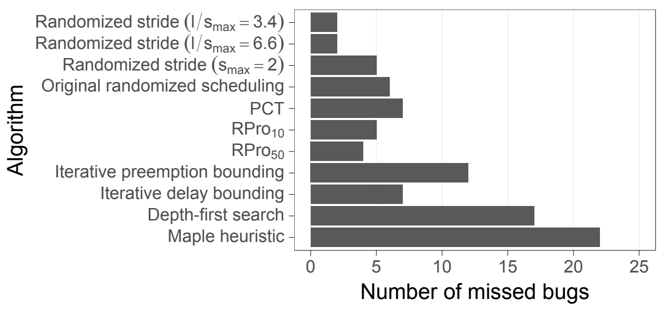

Fig. 7. Number of bugs missed by various algorithms, out of 41 bugs. The data for iterative preemption bounding, iterative delay bounding, depth-first search and Maple is from [48].

this similarity between the two algorithms can actually be expected. Because iterative delay bounding deviates from round-robin scheduling only systematically and in a very limited way, its ability to explore secondary orders is very limited. It is also expected to have a more limited ability to discover secondary orders than the original randomized scheduling because the systematic nature of iterative delay bounding limits it to schedules that are closer to round-robin than the original randomized scheduling.

Additional details of the experiment results are available on https://randomizedstride.wordpress.com/, where the results of running each of the randomized scheduling algorithms on each of the programs are listed.

VII. Threats to Validity

The work presented here is composed of two main parts: A theoretical part and an empirical part. Because the first part used an idealization to enable analyzing the problem, a threat to its validity is whether its results are generally applicable. Mitigating this concern to a large extent is that the results of the empirical work agreed with the general outcomes of the theoretical work, suggesting that these outcomes are of practical significance. For the empirical part of the work, a threat to validity is that the programs and the bugs that were used in this work may not necessarily represent concurrent programs and bugs in general. One point that helps mitigate this threat to validity is that the programs used for the experiment included programs of various sizes and programs where the bugs occurred at different rates, and included programs with both real-world bugs and artificial bugs that were drawn by Thomson et al. [48] from multiple different sources.

VIII. Related Work

The area of concurrency testing has been an active area of research for decades [30]. Techniques for discovering concurrency bugs that are described in research are various and include static analysis [10], [34], [35], symbolic execution [2], [17], [26], [44], sequentialization [12], [37], systematic exploration of thread schedules [9], [16], [18], [19], [31], [32], [49], coverage-based exploration of thread schedules [4], [25], [50], and other techniques and heuristics. Other work has also been done in past research to help with the process of concurrency testing, such as concurrency coverage criteria (e.g. [29], [47]), concurrency mutation operators and mutation analysis (e.g. [3], [13], [14], [15], [27]), and ways to generate test cases or create combinations of test cases that can help expose concurrency bugs (e.g. [39], [40], [53]).

Some concurrency testing techniques depend mainly on randomization to uncover buggy thread schedules. These techniques can be divided into two general categories according to how much control they exercise over thread scheduling: In one category, scheduling decisions are influenced by injecting noise or delays, which can be injected completely randomly or at deliberately selected points according to a heuristic. A notable tool that falls into this category is ConTest [7], [8]. Other work that explored this area recently includes work by Krena et al. [24] and Fiedor et al. [11], who worked on comparing noise-generation heuristics, and Hruba et al. [20], who explored using a genetic algorithm to select the parameters that control noise injection.

In the other category, which has been the focus of this work, scheduling decisions are controlled more closely, either by setting thread priorities that are used by the scheduler of the underlying operating system or runtime to schedule the threads exactly as required by the testing algorithm, such as what is used by PCT [5] and its parallelized variation PPCT [33], or by implementing a thread scheduler that decides which thread to schedule at each relevant event, such as what is done by Maple [52] as available on [1]. Some of the work on this variation of randomized scheduling includes work that aimed at providing guarantees for the probability of discovering bugs with certain properties, including bugs with a maximum specified depth as described by Burckhardt et al. [5] and Nagarakatte et al. [33], and bugs of a specified maximum depth and maximum radius, as in the work of Cai and Yang [6]. Other work aimed at combining randomized scheduling with other techniques, such as partial-order reduction as done by Sen [42].

Randomized scheduling also serves as the basis of some other concurrency testing techniques. For example, it is used as a way to explore the program before applying a different technique that requires some knowledge of the program execution, and sometimes it is used to drive program execution subject to some constraints. Examples of such techniques include [22], [28], [36], [43], [51], [53]. Other work [23], [45] explored using randomized scheduling as a basis for saturation-based testing for discovering concurrency bugs.

IX. Conclusion

This work demonstrated a number of key points. First, the work showed that the ability of randomized scheduling to cover ordering edges is very limited and is impacted negatively when the number of events in the execution of the program under test increases. The work also showed that the ability of randomized scheduling to discover rarely-occurring concurrency bugs in larger programs can improve when the scheduling is made coarser, and discussed the potential trade-off between discovering frequently-occurring and infrequently-occurring bugs. Finally, an approach for making scheduling coarser was described, and experimental results suggested that the approach has advantages over the original randomized scheduling algorithm and over other techniques described in recent literature. The work introduced the concept of a *stride*, limited by a *maximum stride* that can be used to control the coarseness of scheduling and to enable achieving a target probability of discovering the most unlikely ordering edge. The results of the work also confirmed that, to discover some bugs more frequently under randomized scheduling, other bugs are expected to be discovered less frequently, and showed that randomized-stride scheduling achieves a better balance regarding this trade-off.

Although the results show that randomized-stride scheduling was significantly more effective than PCT and RPro, PCT and RPro provide mathematical guarantees that are not provided by the original randomized scheduling nor by randomized-stride scheduling. This result suggests that the guarantees provided by PCT and RPro come at a cost in terms of the general ability to discover bugs. The results do not preclude the possibility that PCT or RPro could discover more of the bugs if higher values of the depth parameter are used. However, making the depth parameter much higher will result in a vanishingly low guaranteed probability of finding bugs. No published work that the author is aware of answers the question of what the best way is to select the depth parameter for PCT and RPro when the depths of the targeted bugs are unknown. Exploring this question in the future may help find ways to make PCT and RPro achieve better results than what is presented in this work.

The results also suggest that there is potential for improvement over the variations of randomized-stride scheduling that are described in this work, considering that randomized-stride scheduling missed 2 out of the 41 bugs that were used in this experiment. Moreover, although the results of randomized-stride scheduling showed no degradation of effectiveness at discovering the most infrequently occurring bugs compared to the original randomized scheduling, it is conceivable to have bugs that are infrequently-occurring under the original randomized scheduling and that become even more infrequent under randomized-stride scheduling. This is possible, for example, if the reason a bug is infrequently-occurring is that it requires covering a large number of ordering edges, all of which represent primary orders.

Acknowledgment

The author would like to acknowledge Prof. G. Jin at North Carolina State University for conversations about this work and for suggestions of wording and suggestions regarding presentation, and Prof. T. Menzies at North Carolina State University for conversations about the research direction.

The author is affiliated with Microsoft Corporation and with North Carolina State University. This work is not meant to represent the points of view of Microsoft Corporation, North Carolina State University, or any entity or person other than the author.

References

[1] SCT Benchmarks. https://sites.google.com/site/sctbenchmarks/. Accessed: 2015-11-02.

[2] S. Anand, C. Păsăreanu, and W. Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer Berlin / Heidelberg, 2007.

[3] J. Bradbury, J. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (J2SE 5.0). In *Second Workshop on Mutation Analysis, 2006*, page 11, 2006.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 206–212, New York, NY, USA, 2005. ACM.

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[6] Y. Cai and Z. Yang. Radius Aware Probabilistic Testing of Deadlocks with Guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 356–367, New York, NY, USA, 2016. ACM.

[7] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[9] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422, New York, NY, USA, 2011. ACM.

[10] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[11] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in noise-based testing of concurrent software. *Software Testing, Verification and Reliability*, 25(3):272–309, May 2015.

[12] B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–713, Nov. 2013.

[13] M. Gligoric, V. Jagannath, Q. Luo, and D. Marinov. Efficient mutation testing of multithreaded code. *Software Testing, Verification and Reliability*, 23(5):375–403, 2013.

[14] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code. In *2010 Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 55–64, Apr. 2010.

[15] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective Mutation Testing for Concurrent Code. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[16] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[17] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental Symbolic Execution of Concurrent Software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 531–542, New York, NY, USA, 2016. ACM.

[18] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[19] G. J. Holzmann. Cloud-Based Verification of Concurrent Software. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, number 9583 in Lecture Notes in Computer Science, pages 311–327. Springer Berlin Heidelberg, Jan. 2016.

[20] V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In G. Fraser and J. T. d. Souza, editors, *Search Based Software Engineering*, number 7515 in Lecture Notes in Computer Science, pages 152–167. Springer Berlin Heidelberg, 2012.

[21] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, number 5643 in Lecture Notes in Computer Science, pages 675–681. Springer Berlin Heidelberg, Jan. 2009.

[22] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 110–120, New York, NY, USA, 2009. ACM.

[23] B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software. In S. Khurshid and K. Sen, editors, *Runtime Verification*, number 7186 in Lecture Notes in Computer Science, pages 177–192. Springer Berlin Heidelberg, Jan. 2012.

[24] B. Křena, Z. Letko, and T. Vojnar. Noise Injection Heuristics for Concurrency Testing. In Z. Kotásek, J. Bouda, I. Černá, L. Sekanina, T. Vojnar, and D. Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, number 7119 in Lecture Notes in Computer Science, pages 123–135. Springer Berlin Heidelberg, Jan. 2012.

[25] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '10, pages 48–58, New York, NY, USA, 2010. ACM.

[26] D. Kroening and M. Tautschnig. CBMC – C Bounded Model Checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 8413 in Lecture Notes in Computer Science, pages 389–391. Springer Berlin Heidelberg, Apr. 2014.

[27] M. Kusano and C. Wang. CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 722–725, Nov. 2013.

[28] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, NY, USA, 2010. ACM.

[29] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, pages 533–536, New York, NY, USA, 2007. ACM.

[30] S. M. Melo, S. R. S. Souza, R. A. Silva, and P. S. L. Souza. Concurrent Software Testing in Practice: A Catalog of Tools. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST 2015, pages 31–40, New York, NY, USA, 2015. ACM.

[31] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[33] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 543–554, New York, NY, USA, 2012. ACM.

[34] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[35] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[36] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT Inter-*

national Symposium on Foundations of software engineering, SIGSOFT '08/FSE-16, pages 135–145, New York, NY, USA, 2008. ACM.

[37] S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

[38] S. Ramanujan. On the sum of the square roots of the first n natural numbers. *Journal of the Indian Mathematical Society*, VII:173–175, 1915. http://ramanujan.sirinudi.org/Volumes/published/ram09.html. Accessed: 2016-12-17.

[39] M. Samak and M. K. Ramanathan. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 131–142, New York, NY, USA, 2015. ACM.

[40] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing Racy Tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 175–185, New York, NY, USA, 2015. ACM.

[41] Scott D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142–157, Dec. 2002.

[42] K. Sen. Effective Random Testing of Concurrent Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 323–332, New York, NY, USA, 2007. ACM.

[43] K. Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[44] K. Sen and G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, volume 4144, pages 419–423. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[45] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based Testing of Concurrent Programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 53–62, New York, NY, USA, 2009. ACM.

[46] E. Suhir. *Applied Probability for Engineers and Scientists*. McGraw-Hill, New York, 1997.

[47] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, Mar. 1992.

[48] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency Testing Using Schedule Bounding: An Empirical Study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 15–28, New York, NY, USA, 2014. ACM.

[49] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[50] C. Wang, M. Said, and A. Gupta. Coverage Guided Systematic Concurrency Testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[51] J. Xue and X. Chang. Race-driven Active Random Testing of Null-pointer Dereferences. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*, Internetware '16, pages 63–70, New York, NY, USA, 2016. ACM.

[52] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 485–502, New York, NY, USA, 2012. ACM.

[53] T. Yu. TACO: Test Suite Augmentation for Concurrent Programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 918–921, New York, NY, USA, 2015. ACM.