Concurrency Testing Using Controlled Schedulers: An Empirical Study

Paul Thomson, Imperial College London
Alastair F. Donaldson, Imperial College London
Adam Betts, Imperial College London

We present an independent empirical study on concurrency testing using controlled schedulers. We have gathered 49 buggy concurrent software benchmarks, drawn from public code bases, which we call SCTBench. We applied a modified version of an existing concurrency testing tool to SCTBench, testing five controlled scheduling techniques: depth-first search, preemption bounding, delay bounding, a controlled random scheduler, and probabilistic concurrency testing (PCT). We attempt to answer several research questions, including: Which technique performs the best, in terms of bug finding ability? How effective are the two main schedule bounding techniques, preemption bounding and delay bounding, at finding bugs? What challenges are associated with applying concurrency testing techniques to existing code? Can we classify certain benchmarks as trivial or non-trivial? Overall, we found that PCT (with parameter d=3) was the most effective technique in terms of bug finding; it found all the bugs found by the other techniques, plus an additional three, and it missed only one bug. Surprisingly, we found that the "naïve" controlled random scheduler, which randomly chooses one thread to execute at each scheduling point, performs well, finding more bugs than preemption bounding and just two fewer bugs than delay bounding. Our findings confirm that delay bounding is superior to preemption bounding and schedule bounding is superior to an unbounded depth-first search. The majority of bugs in SCTBench can be exposed using a small schedule bound (1-2), supporting previous claims, although one benchmark requires 5 preemptions. We found that the need to remove nondeterminism and control all synchronization (as is required for *systematic* concurrency testing) can be nontrivial. There were 8 distinct programs that could not easily be included in out study, such as those that perform network and inter-process communication. We report various properties about the benchmarks tested, such as the fact that the bugs in 18 benchmarks were exposed 50% of the time when using random scheduling. We note that future work should not use the benchmarks that we classify as trivial when presenting new techniques, other than as a minimum baseline. We have made SCTBench and our tools publicly available for reproducibility and use in future work.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

Additional Key Words and Phrases: Concurrency, systematic concurrency testing, stateless model checking, context bounding

ACM Reference Format:
Paul Thomson, Alastair F. Donaldson, and Adam Betts, 2015. Concurrency Testing Using Schedule Bounding: an Empirical Study. *ACM Trans. Parallel Comput.* X, Y, Article Z (May 2015), 38 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

This work was supported by an EPSRC-funded PhD studentship and the EU FP7 STEP project CARP (project number 287767).

DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

As concurrency has increasingly become a mainstream concern for software developers, researchers have investigated numerous methods for finding *concurrency bugs*—software defects (such as crashes, deadlocks, assertion failures, memory safety errors and errors in algorithm implementation) that arise directly or indirectly as a result of concurrent execution. This is motivated by the rise of multicore systems [Sutter and Larus 2005], the ineffectiveness of traditional testing for detecting and reproducing concurrency bugs due to nondeterminism [Křena et al. 2010], and the desire for automatic, precise analysis, which is hard to achieve using static techniques [Bessey et al. 2010].

An important class of techniques employ *controlled schedulers*, where all scheduling nondeterminism is controlled, execution is serialized, and the controlled scheduler determines precisely which thread is executed and for how long. The possible schedules are searched either systematically [Wang et al. 2011; Emmi et al. 2011; Musuvathi et al. 2008; Yang et al. 2008; Godefroid 1997], or through the use of randomization [Burckhardt et al. 2010; Nagarakatte et al. 2012; Thomson et al. 2014]. In contrast, *non-controlled* methods include those that perturb the OS scheduler by inserting calls to `sleep` (possibly with randomization) or other similar functions [Edelstein et al. 2002; Sen 2008; Park et al. 2009; Yu et al. 2012]. Controlled scheduling has the attractive property that the schedule under consideration can be easily recorded, facilitating *replay* of bugs.

Systematic concurrency testing (SCT) [Godefroid 1997; Musuvathi et al. 2008; Wang et al. 2011; Yang et al. 2008; Emmi et al. 2011] (also known as *stateless model checking* [Godefroid 1997]), is a well-known controlled scheduling technique where a multithreaded program is executed repeatedly such that a different schedule is explored on each execution. This process continues until all schedules have been explored, or until a time or schedule limit is reached. The analysis is highly automatic and has no false-positives, and supports reproduction of bugs by forcing the bug-inducing schedule. It has been implemented in a variety of tools, including Verisoft [Godefroid 1997], CHESS [Musuvathi et al. 2008], INSPECT [Yang et al. 2008] and Maple [Yu et al. 2012].

Assuming a nondeterministic scheduler, the number of possible thread interleavings for a concurrent program is exponential in the number of execution steps, so exploring all schedules for large programs using SCT is infeasible. To combat this schedule explosion, *schedule bounding* techniques have been proposed, which reduce the number of thread schedules that are considered with the aim of preserving schedules that are likely to induce bugs. *Preemption bounding* [Musuvathi and Qadeer 2007a] bounds the number of preemptive context switches that are allowed in a schedule. *Delay bounding* [Emmi et al. 2011] bounds the number of times a schedule can deviate from the scheduling decisions of a given deterministic scheduler. During concurrency testing, the bound on preemptions or delays can be increased iteratively, so that all schedules are explored in the limit; the intention is that interesting schedules are explored within a reasonable resource budget. When applied iteratively, these methods are referred to as *iterative preemption bounding* and *iterative delay bounding*.

Schedule bounding has two additional benefits, regardless of bug finding ability. First, it produces simple counterexample traces: a trace with a small number of preemptions is likely to be easier to understand than a trace with many preemptions. This property has been used in trace simplification [Huang and Zhang 2011; Jalbert and Sen 2010]. Secondly, it gives bounded coverage guarantees: if the search explores all schedules with at most c preemptions, then any undiscovered bugs in the program require at least $c+1$ preemptions. A guarantee of this kind provides some indication of

the necessary complexity and probability of occurrence of any bugs that might remain, and work on concurrent software verification employs schedule bounding to improve tractability [Lal and Reps 2009; Cordeiro and Fischer 2011; Atig et al. 2014].

The *probabilistic concurrency testing* (PCT) algorithm [Burckhardt et al. 2010] is another controlled scheduling technique that repeatedly executes a multithreaded program using a randomized priority-based scheduler, where the highest priority thread that is enabled is scheduled. The initial thread priorities are chosen randomly, as are $d-1$ priority change points. Importantly, the depth of each change point (the number of execution steps before the change point causing the currently executed thread to be lowered) is chosen *uniformly* over the length of the execution. Given a program with n threads, at most k execution steps and a bug that requires d ordering constraints to manifest, PCT guarantees that it will find the bug in a single execution with probability $1/nk^{d-1}$ [Burckhardt et al. 2010]. We refer to PCT as being non-systematic (i.e. non-exhaustive), because the same schedule could be explored more than once and schedules are not exhaustively enumerated in some order. However, PCT is still a controlled technique: because all scheduling nondeterminism is controlled, the schedule to be executed is precisely determined by the PCT algorithm and not by the OS scheduler. Thus, buggy schedules can still be recorded and replayed precisely, as with SCT.

The hypothesis that SCT with preemption and delay bounding is likely to be effective is based on empirical evidence suggesting that many interesting concurrency bugs require only a small number of preemptive context switches to manifest [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011]. The evidence that many concurrency bugs only require a small number of ordering constraints [Burckhardt et al. 2010; Lu et al. 2008], is arguably similar. Prior work has also shown that delay bounding improves on preemption bounding, allowing additional bugs to be detected [Emmi et al. 2011]. However, evaluation of systematic techniques has focused on a particular set of C# and C++ programs that target the Microsoft Windows operating system, most of which are not publicly available. Additionally, this work does not explicitly show that schedule bounding provides benefit over a random scheduler for finding bugs.[1] By "random scheduler", we mean a controlled scheduler that randomly chooses a thread to execute after each scheduling point. The PCT algorithm has been shown to find bugs in large applications, such as Mozilla Firefox and Internet Explorer [Burckhardt et al. 2010]. However, a thorough comparison of PCT with systematic techniques and controlled random scheduling has not been conducted.[2]

We believe that these exciting and important claims about the effectiveness of controlled scheduling would benefit from further scrutiny using a wider range of publicly available applications. To this end, we present an independent, reproducible empirical study of controlled scheduling techniques. We have put together SCTBench, a set of 49 publicly available benchmarks, gathered from a combination of stand-alone multithreaded test cases, and test cases drawn from 13 distinct applications and libraries. These are benchmarks that have been used in previous work to evaluate concurrency testing tools (although mostly not in the context of controlled scheduling), with a few additions. The benchmarks are amenable to systematic concurrency testing (and, thus, all controlled scheduling techniques) so that we can compare systematic techniques with non-systematic techniques. Our study uses an extended version of Maple [Yu et al. 2012], an open source concurrency testing tool.

[1] We note that [Musuvathi and Qadeer 2007a] plots the state (partial-order) coverage of preemption bounding against a technique called "random" on a single benchmark, but the details of this and the bug finding ability are not mentioned.

[2] We note that [Burckhardt et al. 2010] compares PCT against the use of random sleeps, but not against controlled random scheduling. PCT is also compared against preemption bounding on two benchmarks.

Our aim was to use SCTBench to empirically compare the following techniques: unbounded SCT, SCT with iterative preemption bounding and iterative delay bounding, PCT, and controlled random scheduling, to answer the following research questions (RQs):

RQ1 Which technique performs the best, in terms of bug finding ability?
RQ2 Does PCT beat the other techniques, as might be expected given previous work [Burckhardt et al. 2010]?
RQ3 How effective is the controlled random scheduler (a naïve, baseline technique) in comparison to the other techniques?
RQ4 Does delay bounding beat preemption bounding, as in prior work [Emmi et al. 2011], and do both schedule bounding techniques beat a straightforward unbounded depth-first search, as in prior work [Musuvathi and Qadeer 2007a; Emmi et al. 2011]?
RQ5 How many bugs can be found with a small number of preemptions/delays, and can we find non-synthetic examples of concurrency bugs that require more than three preemptions (the largest number of preemptions required to expose a bug in previous work [Emmi et al. 2011])?
RQ6 How easy is it to apply controlled scheduling techniques (and, in particular, systematic techniques) to various existing code bases in practice?
RQ7 Can we classify certain benchmarks exhibiting defects as highly trivial or non-trivial, based on the ease or difficulty with which the techniques we study are able to expose defects?

We answer **RQ1–RQ5** quantitatively by investigating the number of bugs found by each technique within a schedule limit, showing how these numbers vary as the schedule limit is increased. We answer **RQ6** qualitatively, based on our experience collecting and modifying benchmarks during the construction of SCTBench. To answer **RQ7**, we identify a number of properties of benchmarks that indicate when bug-finding is trivial, and report on the extent to which SCTBench examples exhibit these trivial properties. We also report on benchmarks that appear to present a challenge for controlled scheduling methods, based on the fact that associated defects were missed by several (or, in one case, all) of the techniques we study.

1.1. Main findings

We summarize the main findings of our study, which relate to the above research questions. The conclusions we draw of course only relate to the 49 benchmarks in SCTBench, but this does include publicly available benchmarks used in prior work to evaluate concurrency testing tools. We forward-reference the Venn diagrams of Figure 3 and the cumulative graphs in Figure 4 and 5, discussed in detail in §6. These diagrams provide an overview of our results in terms of the bug-finding ability of the various techniques we study: iterative preemption bounding (IPB), iterative delay bounding (IDB), depth-first search with no schedule bound (DFS), three parameterized version of probabilistic concurrency testing (PCT $d = n$, for $n \in \{1, 2, 3\}$) and a controlled random scheduler (Rand). For each technique evaluated, a limit of 100,000 schedules per benchmark was used, except for two benchmarks where (as explained in §5) a lower limit was used. We consider a bug to be an assertion failure, deadlock, crash or incorrect output. For the bugs that led to incorrect output, we added assertions in the code to check the output based on the text description of the bug that we found.

RQ1, RQ2: PCT d=3 performed best. With a limit of 100,000 schedules, PCT d=3 found bugs in 48 of the 49 benchmarks—more than any other technique—including all 45 bugs found by IDB, the next best non-PCT technique in terms of number of

bugs found. For lower schedule limits, PCT d=3 still found the most bugs, except for very low schedules limits ($<$10). This concurs with the findings of prior work, in which PCT found bugs faster than IPB [Burckhardt et al. 2010]. However, we note that the three bugs missed by IDB (but found by PCT d=3) are in benchmarks with high thread counts and IDB *was* able to find these bugs within the schedule limit when the thread count was reduced.

RQ3: Controlled random scheduling performed better than IPB and comparably with IDB. Because it is so straightforward, our assumption prior to this study was that use of a controlled random scheduler for bug-finding would not be effective. We initially investigated this method merely because it provides a simple baseline that more sophisticated techniques should surely improve upon (and because this was suggested by a reviewer of the conference version of this article [Thomson et al. 2014]). The effectiveness of controlled random scheduling for bug finding is not addressed in prior work; the papers that introduced preemption bounding [Musuvathi and Qadeer 2007a] and delay bounding [Emmi et al. 2011] only include depth-first search or preemption bounding as a baseline for finding bugs (see Footnote 1, p. 3). Our findings, summarized in Figure 3c, contradict our assumption: with a schedule limit of 100,000, Rand found 43 bugs, more than IPB (38) and DFS (33), and found all but 2 of the bugs found by IDB (45). Furthermore, as shown in Figure 4, the results are similar when lower schedule limits are considered: for schedule limits between 10 and 1000, Rand finds up to 6 more bugs than IDB. This raises two important questions: Does IPB actually aid in bug finding, compared to more naïve approaches? Are the benchmarks used to evaluate concurrency testing tools (captured by SCTBench) representative of real-world concurrency bugs? Our findings indicate that the answer to at least one of these questions must be "no". Nevertheless, as noted above, schedule bounding provides several benefits regardless of bug finding ability, which are not questioned by our findings.

RQ3, RQ7: Researchers should compare against controlled random scheduling. Much prior work that introduced new techniques did not compare against a controlled random scheduler. Many benchmarks contain defects that can be trivially found using a controlled random scheduler. We stress that future work should use the controlled random scheduler as a baseline, to give an accurate representation of the benchmarks used and the improvement obtained by the new technique.

RQ4: IDB beats IPB. Schedule bounding beats DFS. With a schedule limit of 100,000, IDB found all of the 38 bugs that were found by IPB, plus an additional 7 (see Figure 3a). This is in line with experimental claims of prior work [Emmi et al. 2011]. A straightforward depth-first search with no schedule bounding only exposed bugs in 33 benchmarks, all of which were also found by IPB, as well as by IDB. This also validates prior work [Musuvathi and Qadeer 2007a; Emmi et al. 2011]. Results were similar in terms of number of bugs found at various lower schedule limits (see Figure 4).

RQ5: Many bugs could be found using a small schedule bound. With a schedule limit of 100,000, schedule bounding exposed each bug in 45 of the 49 benchmarks, and 44 of these require a preemption bound of 2 or less. Thus, a large majority of the bugs in SCTBench can be found with a small schedule bound. This supports previous claims that in practice many bugs can be exposed using a small number of preemptions or delays [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011]. It also adds weight to the argument that bounded guarantees provided by schedule bounding are useful. However, we note that one bug that was found by schedule bounding requires 3 preemptions and another is reported to require a minimum of 5 preemptions. Also note that certain synthetic benchmarks (such as `reorder_X_bad` and

`twostage_X_bad`) are challenging for schedule bounding when the number of threads parameter, X, is increased; as X is incremented, so is the number of delays required for IDB to find the bug. However, it is not clear whether such a scenario is likely to occur in real multithreaded programs.

RQ6: Controlled testing techniques can be difficult to apply. There were 8 distinct programs (providing 26 potential test cases) that could not easily be included in our study, as they use nondeterministic features or additional synchronization that is not modelled or controlled appropriately by most tools. This includes network communication, multiple processes, signals (other than pthread condition variables) and event libraries. It is sometimes possible to apply non-systematic techniques, such as PCT and random scheduling, to such benchmarks. However, this depends on how the techniques are implemented.

Additionally, program modules were often difficult to test in isolation due to direct dependencies on system functions and other program modules. Thus, creating isolated tests suitable for concurrency testing (or even unit testing) may require significant effort, especially for testers who are not familiar with the software under test.

RQ7: Trivial benchmarks. We argue that certain benchmarks used in prior work are "trivial" (based on properties which we discuss in §6.4 and summarize in Table II) and cannot meaningfully be used to compare the performance of competing techniques. Instead, they provide a minimum baseline for any respectable concurrency testing technique. For example, the bugs in 18 benchmarks were exposed 50% of the time when using random scheduling; in 8 of these cases, the bugs were exposed 100% of the time.

RQ7: Non-trivial benchmarks. We believe most benchmarks from the CHESS, PARSEC and RADBench suites, as well as the `misc.safestack` benchmark (see §4), present a non-trivial challenge for concurrency testing tools. Furthermore, these represent real bugs, not synthetic tests. Future work can use these challenging benchmarks to show the improvement obtained over prior techniques. We also recommend that the research community focus on increasing the corpus of non-trivial concurrency benchmarks that are available for evaluation of analysis tools.

We also summarize several notable findings that do not directly relate to the above research questions:

Data races are common. Many (30) of the benchmarks we tested exhibited data races. Although we did not analyze these data races in detail, to the best of our knowledge they are not regarded as bugs by the relevant benchmark developers. Treating data races as errors would hide the more challenging bugs that the benchmarks capture. Future work that uses these benchmarks must take this into account. For the study, we explore the interleavings arising from sequentially consistent outcomes of data races in order to expose assertion failures, deadlocks, crashes or incorrect outputs.

Some bugs may be missed without additional checks. Some concurrency bugs manifest as out-of-bounds memory accesses, which do not always cause a crash. Tools need to check for these, otherwise bugs may be missed or manifest nondeterministically, even when the required thread schedule is executed. Performing such checks reliably and efficiently is non-trivial.

1.2. SCTBench and reproducibility of our study

To make our study reproducible, we provide the 49 benchmarks (SCTBench), our scripts and the modified version of Maple used in our experiments, online:

http://sites.google.com/site/sctbenchmarks

We believe SCTBench will be valuable for future work on concurrency testing in general and SCT in particular. Each benchmark is directly amenable to SCT and exhibits a concurrency bug. As discussed further in §5, our results are given in terms of number of schedules, not time, which allows them to be easily compared with other work and tools.

1.3. Contribution over conference version of this work

This article is a revised and extended version of a conference paper [Thomson et al. 2014]. The original study of [Thomson et al. 2014] was restricted to systematic concurrency testing with schedule bounding, using the non-systematic random scheduling technique as a point of comparison.

The major novel contribution of the current work is an extension of the study to consider a more advanced non-systematic technique, probabilistic concurrency testing (PCT) [Burckhardt et al. 2010]. We provide background on the PCT method in §2 and explain in §3 how we adapted the Maple tool to include an implementation of PCT suitable for comparison with other controlled techniques. We discuss the incorporation of PCT in our experimental method in §5 and include a detailed discussion of how PCT compares with the other techniques in terms of bug-finding ability in §6. Since the PCT method accepts a parameter d, we tested three different versions of PCT for $d \in \{1, 2, 3\}$ over our benchmarks.

We also increased the schedule limit in our experiments, for all techniques, from 10,000 to 100,000 (with the exception of two benchmarks which we discuss in §5). We additionally consider in §5 how the techniques compare for schedule limits between 1 and 100,000.

2. CONCURRENCY TESTING WITH CONTROLLED SCHEDULERS

Concurrency testing using controlled scheduling works by repeatedly executing a multithreaded program, exploring a particular schedule on each execution. Execution is serialized, so that concurrency is emulated by interleaving the instructions of different threads. Typically, operations that can cause threads to become enabled/disabled (unblocked/blocked) must be modelled or carefully monitored by the concurrency testing implementation in order to update the state of the scheduler. The search space is over schedules. Unlike in stateful model checking, program states are not represented. This is appealing because the state of real software is large and difficult to capture.

In this section, we describe the controlled scheduling techniques used in the study. We start by formalizing the notion of a schedule (§2.1). We then describe systematic concurrency testing (§2.2), preemption bounding (§2.3) and delay bounding (§2.4), discuss upper bounds on schedules for the systematic techniques, and explain how schedule bounding can be applied in an iterative manner (§2.5). We then turn to the two non-systematic controlled techniques studied in the paper, random scheduling (§2.6) and probabilistic concurrency testing (§2.7).

2.1. Schedules

A schedule $\alpha = \langle \alpha(1), \ldots, \alpha(n) \rangle$ is a list of thread identifiers. We use the following shorthands for lists: $\alpha \cdot t = \langle \alpha(1), \ldots, \alpha(n), t \rangle$; $\mathit{last}(\alpha) = \alpha(n)$. The element $\alpha(i)$ refers to the thread that is executing at step i in the execution of the multithreaded program. where step 1 is the first step. For example, the schedule $\langle T0, T0, T1, T0 \rangle$ specifies that, from the initial state, two steps are executed by $T0$, one step by $T1$ and then one step by $T0$. A step corresponds to a particular thread executing a *visible* operation [Godefroid 1997], such as a synchronization operation or shared memory access, followed by a finite sequence of *invisible* operations until immediately before the next visible operation. Considering interleavings involving non-visible operations is unnecessary when

checking safety property violations, such as deadlocks and assertion failures [Godefroid 1997]. The point just before a visible operation, where the scheduler decides which thread to execute next, is called a *scheduling point*. Let $\mathit{enabled}(\alpha)$ denote the set of enabled threads (those that are not blocked, and so can execute) in the state reached by executing α. We say that the state reached by α is a *terminal state* when $\mathit{enabled}(\alpha) = \emptyset$. A schedule that reaches a terminal state when executed is a *terminal schedule*. However, throughout the paper, when discussing the execution of a schedule or the number of schedules explored, we shall use *schedule* to mean *terminal schedule* unless stated otherwise.

2.2. Systematic Concurrency Testing (SCT)

Systematic concurrency testing (SCT) works by repeatedly executing a multithreaded program, exploring a *different* schedule on each execution. It is assumed that the only source of nondeterminism is from the scheduler so that repeated execution of the same schedule always leads to the same program state. Thus, nondeterminism from sources such as user input, files, network packets, etc. must be fixed or modelled. When the algorithm completes, *all* terminal schedules have been explored. However, a time limit or schedule limit is often imposed, since exploring all schedules is usually infeasible.

The schedule space of a deterministic program can conceptually be represented as a prefix-tree, where each node is a schedule and the branches of a node (labelled with thread identifiers) are the enabled threads at the scheduling point. The schedule tree is not known a priori; it is discovered on-the-fly. Systematic concurrency testing is usually implemented by performing a depth-first search of the schedule tree, as this allows the unexplored schedule prefixes to be efficiently stored in a stack data structure. After exploring the first terminal schedule, the search then backtracks to the most recent scheduling point; the next schedule is explored by executing the program from the start, replaying the previous schedule up to the most recent scheduling point, scheduling the next enabled thread and then continuing scheduling threads until termination.

2.3. Preemption Bounding

Preemption bounding [Musuvathi and Qadeer 2007a] is a form of SCT where the number of preemptive context switches in a schedule is bounded. We first define preemptive and non-preemptive context switches.

Context switches. A *context switch* occurs in a schedule when execution switches from one thread to another. Formally, step i in α is a context switch if and only if step i is not the first step and $\alpha(i) \neq \alpha(i-1)$. The context switch is *preemptive* if and only if $\alpha(i-1) \in \mathit{enabled}(\langle \alpha(1), \ldots, \alpha(i-1) \rangle)$. In other words, the thread executing step $i-1$ remained enabled after that step. Otherwise, the context switch is *non-preemptive*.

We define the preemption count PC of a schedule recursively. A schedule of length zero or one has no preemptions. Otherwise:

$$\mathit{PC}(\alpha \cdot t) \;=\; \begin{cases} \mathit{PC}(\alpha) + 1 & \text{if } \mathit{last}(\alpha) \neq t \wedge \mathit{last}(\alpha) \in \mathit{enabled}(\alpha) \\ \mathit{PC}(\alpha) & \text{otherwise} \end{cases}$$

With a preemption bound of c, any schedule α with $\mathit{PC}(\alpha) > c$ will not be explored.

The idea behind preemption bounding is that it greatly reduces the number of schedules, but still allows many bugs to be found [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011]. The intuition is that many bugs only require a *few* preemptions at at the *right places* in order to manifest. In contrast, an unbounded search is unlikely to complete without exceeding the time or schedule limit; in these cases, the search will end prematurely and be biased towards exploring a large num-

```
T0                      T1              T2          T3
a) create(T1,T2,T3)     b) x=1          d) z=1      e) assert(x==y)
                        c) y=1
```

Fig. 1: Simple multithreaded program.

ber of deep preemptions (that occur towards to end of schedules) due to the use of a depth-first search. Many of these may not be useful for exposition of bugs. Using a low preemption bound increases the chance of exploring all schedules within the preemption bound, without exceeding the time or schedule limit, which will include exploring preemptions at various depths.

EXAMPLE 1. *Consider Figure 1, which shows a simple multithreaded program. T0 launches three threads concurrently and is then disabled. All variables are initially zero and threads execute until there are no statements left. We refer to the visible actions of each thread via the statement labels (a, b, c, etc.) and we (temporarily) represent schedules as a list of labels. Note that 'a' cannot be preempted, as there are no other threads to switch to.*

A schedule with zero preemptions is $\langle a, b, c, e, d\rangle$. Note that, for example, e is not a preemption in this particular schedule because T1 has no more statements and so is considered disabled after c. A schedule that causes the assertion to be violated is $\langle a, b, e\rangle$, which has one preemption at operation e. The bug will not be found with a preemption bound of zero, but will be found with any greater bound.

2.4. Delay Bounding

A *delay* conceptually corresponds to blocking the thread that would be chosen by the scheduler at a scheduling point, which forces the next thread to be chosen instead. The blocked thread is then immediately re-enabled. Delay bounding [Emmi et al. 2011] bounds the number of delays in a schedule, given an otherwise deterministic scheduler. Executing a program under the deterministic scheduler (without delaying) results in a single terminal schedule – this is the only terminal schedule that has zero delays.

In the remainder of this paper we assume that delay bounding is applied in the context of a non-preemptive round-robin scheduler that considers threads in thread creation order, starting with the most recently executing thread. We assume this instantiation of delay bounding because it has been used in previous work [Emmi et al. 2011] and is straightforward to explain and implement.

The following is a definition of delay bounding assuming the non-preemptive round-robin scheduler. Assume that each thread id is a non-negative integer, numbered in order of creation; the initial thread has id 0, and the last thread created has id $N-1$. For two thread ids $x, y \in \{0, \ldots, N-1\}$, let *distance*$(x, y)$ be the unique integer $d \in \{0, \ldots, N-1\}$ such that $(x + d) \bmod N = y$. Intuitively, this is the "round-robin distance" from x to y. For example, given four threads $\{0, 1, 2, 3\}$, *distance*$(1, 0)$ is 3. For a schedule α and a thread id t, let *delays*(α, t) yield the number of delays required to schedule thread t at the state reached by α:

$$\textit{delays}(\alpha, t) = |\{x : 0 \leq x < \textit{distance}(\textit{last}(\alpha), t)) \ \wedge \ (\textit{last}(\alpha) + x) \bmod N \in \textit{enabled}(\alpha)\}|$$

This is the number of enabled threads that are skipped when moving from *last*(α) to t. For example, let *last*$(\alpha) = 3$, *enabled*$(\alpha) = \{0, 2, 3, 4\}$ and $N = 5$. Then, *delays*$(\alpha, 2) = 3$ because threads 3, 4 and 0 are skipped (but not thread 1, because it is not enabled).

```
T0                          T1          T2          T3
a) create(T1,T2,T3)         b) x=1      f) x=1      e) assert(x==y)
                            c) y=1      g) y=1
```

Fig. 2: Adversarial delay-bounding example.

We define the delay count *DC* of a schedule recursively. A schedule of length zero or one has no delays. Otherwise:

$$DC(\alpha \cdot t) \;=\; DC(\alpha) + delays(\alpha, t)$$

With a delay bound of c, any schedule α with $DC(\alpha) > c$ will not be explored.

The set of schedules with at most c delays is a subset of the set of schedules with at most c preemptions. Thus, delay bounding reduces the number of schedules by at least as much as preemption bounding.

The intuition behind delay bounding is similar to that of preemption bounding; that is, many bugs can be found with only a few preemptions or delays [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011]. The extra idea behind delay bounding is that it often also does not matter *which* thread is switched to after a preemption; thus, allowing only the next enabled thread without spending additional delays reduces the number of schedules more than preemption bounding, while still allowing many bugs to be found.

EXAMPLE 2. *Consider Figure 1 once more. Assume thread creation order* $\langle T0, T1, T2, T3\rangle$*. The assertion can also fail via:* $\langle a, b, d, e\rangle$*, with one delay/preemption at* d*. However, a preemption bound of one yields 11 terminal schedules, while a delay bound of one yields only 4 (note that an assertion failure is a terminal state).*

Now consider Figure 2, which is a modified version of the program where the statements of T2 have been replaced with the same statements as T1, which we label as f) and g). Now, the assertion cannot fail with a delay bound of one because two delays must occur so that T1 and T2 do not both execute all their statements. For example, $\langle a, b, e\rangle$ *exposes the bug, but executing* e *uses two delays. However, this schedule only has one preemption, so the assertion can still fail under a preemption bound of one.*

Adding an additional n *threads between T1 and T3 (in the creation order) with the same statements as T1 will require* n *additional delays to expose the bug, while still only one preemption will be needed. Empirical evidence [Emmi et al. 2011] suggests that adversarial examples like this are not common in practice. Our results (§6) also support this.*

2.5. Upper Bounds for Terminal Schedules, and Iterative Schedule Bounding

Upper-bounds for the number of terminal schedules produced by the above systematic techniques are described in [Musuvathi and Qadeer 2007a; Emmi et al. 2011]. In summary, assume at most n threads and at most k execution steps in each thread. Of those k, at most b steps block (cause the executing thread to become disabled) and i steps do not block. Complete search (exploring all schedules) is exponential in n and k, and thus infeasible for programs with a large number of execution steps. With a scheduling bound of c, preemption bounding is exponential in c (a small value), n (often, but not necessarily, a small value) and b (usually much smaller than k). Crucially, it is no longer exponential in k. Delay bounding is exponential only in c (a small value). Thus, delay bounding performs well (in terms of number of schedules) even when programs create a large number of threads.

Schedule bounding can also be performed iteratively [Musuvathi and Qadeer 2007a], where all schedules with zero preemptions/delays are executed, followed by those with one preemption/delay, etc. until there are no more schedules or until a time or schedule limit is reached. In the limit, all schedules are explored. Thus, iterative schedule bounding creates a partial-order in which to explore schedules: schedule α will be explored before schedule α' if $PC(\alpha) < PC(\alpha')$, while there is no predefined exploration order between schedules with equal preemption counts (we address how this affects the fairness of our evaluation in §5). The partial-order for iterative delay bounding with respect to *DC* is analogous. Thus, iterative schedule bounding is a heuristic that prioritizes schedules with a low preemption count, aiming to expose buggy schedules before the time or schedule limit is reached.

In this study, we perform preemption bounding and delay bounding iteratively.

2.6. Controlled Random Scheduling

A controlled random scheduler uses randomization to determine the schedule that is explored. At each scheduling point, one thread is randomly chosen from the set of enabled threads using a uniform distribution. This thread is then scheduled until the next scheduling point. Unlike schedule fuzzing, where random sleeps are used to perturb the OS scheduler [Ben-Asher et al. 2006], the random scheduler fully controls scheduling nondeterminism. As with any controlled scheduling technique, the executed schedule can easily be recorded and replayed (because schedule nondeterminism is controlled). However, for controlled random scheduling (and unlike SCT), no information is saved *for subsequent executions*, so it is possible that the same terminal schedule will be explored multiple times. As a result, the search cannot "complete", even for programs with a small number of schedules. Additionally, a random scheduler *can* be used on programs that exhibit nondeterminism *beyond* scheduler nondeterminism, although in this case schedule replay will be unreliable.

2.7. Probabilistic Concurrency Testing

The PCT algorithm [Burckhardt et al. 2010] uses a randomized priority-based scheduler such that the highest priority enabled thread is scheduled at each step. A bounded number of *priority change points* are inserted at random depths in the execution which change the currently executing thread's priority to a low value. Importantly, the random depths of the change points are chosen *uniformly* over the *length* (number of steps) of the execution. This is in contrast to random scheduling, where a random choice is made at every execution step.

More formally, the algorithm is described in [Burckhardt et al. 2010] as follows. Given a program with at most n threads and at most k steps, choose a bound d.

(1) Randomly assign each of the n threads a distinct *initial* priority value from $\{d, d+1, \ldots, d+n\}$. The lower priority values $\{1, \ldots, d-1\}$ are reserved for change points.
(2) Randomly pick integers $k_1, \ldots, k_{d-1}$ from $\{1, \ldots, k\}$. These will be the priority change points.
(3) Schedule threads strictly according to their priorities; that is, never schedule a thread if a higher priority thread is enabled. After executing the k_i-th step ($1 \leq i < d$), change the priority of the thread that executed the step to i.

The work on PCT also introduces the idea of a *bug depth* metric—not to be confused with the depth (number of steps) of a schedule. The *bug depth* is defined as the minimum set of ordering constraints between instructions from different threads that are sufficient to trigger the bug [Burckhardt et al. 2010]. Assuming a bug with depth d, the probability of the PCT algorithm detecting the bug on a single execution is $1/nk^{d-1}$ (inverse exponential in d).

As with random scheduling (and unlike SCT), no information is saved during one execution to inform subsequent executions, so the search cannot "complete" and the technique *can* be used on programs with nondeterminism. Similar to schedule bounding, the intuition behind PCT is that many concurrency bugs typically require certain orderings between only a few instructions in order to manifest [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011; Lu et al. 2008].

EXAMPLE 3. *Once again, consider the program in Figure 2. For this program, the number of threads is* $n = 4$ *and the number of steps is* $k = 6$*. One way for the bug to occur is for statement* e *to occur after* b *but before* c*. This is possible with one priority change point, so let* $d = 2$*. Assume the initial random thread priorities chosen are:*

$$\{T0 \mapsto 5, T1 \mapsto 4, T3 \mapsto 3, T2 \mapsto 2\}.$$

Assume the random priority change point chosen is $k_1 = 2$*. Thus, the schedule that will be explored is:* $\langle a, b, e \rangle$*, which causes the assertion to fail. Statement* a *is executed because* $T0$ *has the highest priority.* $T0$ *then becomes disabled, so* $T1$ *becomes the highest priority thread that is enabled and* b *is executed. At this point, step 2 was just executed; thus, the priority change point is triggered and* $T1$*'s priority is lowered to* 1*.* $T3$ *becomes the highest priority thread that is enabled and so* e *is executed.*

3. MAPLE

We chose to use a modified version of the Maple tool [Yu et al. 2012] to conduct our experimental study. Maple is a concurrency testing tool framework for pthread [Lewis and Berg 1998] programs. It uses the dynamic instrumentation library, PIN [Luk et al. 2005], to test binaries without the need for recompilation. One of the modules, *systematic*, is a re-implementation of the CHESS [Musuvathi et al. 2008] algorithm for preemption bounding. The main reason for using Maple, instead of CHESS, is that Maple targets pthread programs. This allows us to test a wide variety of open source multithreaded benchmarks and programs. Previous evaluations [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011] focus on C# programs and C++ programs that target the Microsoft Windows operating system, most of which are not publicly available. In addition, CHESS requires re-linking the program with a test function that can be executed repeatedly; this requires resetting the global state (e.g. resetting the value of global variables) and joining any remaining threads, which can be non-trivial. In contrast, Maple can test native binaries out-of-the-box, by restarting the program for each terminal schedule that is explored. A downside of this approach is that it is slower. Checking for data races is also supported by Maple; as discussed in §5, this is important for identifying visible operations. The public version of CHESS can only interleave memory accesses in native code if the user adds special function calls before each access.[3]

DFS. As explained in §2.2, systematic techniques generally use a depth-first search in order to efficiently store the remaining unexplored schedules using a stack data structure. Maple is no exception; since the stack is deeply ingrained in Maple's data structures and algorithms, we did not attempt to implement other approaches for SCT. The type of depth-first search determines the order in which schedules are explored—recall from §2.2 that the schedule space can be represented as a prefix-tree, where each node is a schedule and the branches of a node (labelled with thread identifiers) are the enabled threads at the scheduling point. In our study, we use a left-recursive depth-first search where child branches (thread identifiers) are ordered in thread creation

[3]See "Why does wchess not support /detectraces?" at http://social.msdn.microsoft.com/Forums/en-us/home?forum=chess

order, starting with the most recently executing thread and wrapping in a round-robin fashion. Thus, the initial execution explores the non-preemptive round-robin schedule; this is the same for all systematic techniques: unbounded depth-first search, iterative preemption bounding and iterative delay bounding. We discuss the impact of depth-first search on our study in §5.

Preemption bounding. Maple already included support for preemption bounding, using the underlying depth-first search approach.

Delay bounding. We modified Maple to add support for delay bounding, following a similar design to preemption bounding. At each scheduling point, Maple conceptually constructs several schedules consisting of the current schedule concatenated with an enabled thread t. If switching to thread t will cause the delay bound to be exceeded (as explained in §2.4), the schedule is not considered.

Controlled random scheduling. Maple already included a controlled random scheduler (although this was not used in prior work [Yu et al. 2012]). As explained in §2.6, at each scheduling point, one thread is randomly chosen from the set of enabled threads using a uniform distribution; that thread is then scheduled for one step.

PCT algorithm. Prior to our modifications, Maple already included a version of PCT implemented using Linux scheduler priorities [Yu et al. 2012]. By changing settings of the Linux scheduler, it is apparently possible to implement strict priorities, as required for PCT. However, in order to ensure that we are using an implementation that is identical to the one described in the original PCT paper [Burckhardt et al. 2010], we re-implemented PCT within the controlled scheduler framework of Maple; as such, our implementation is very similar to the pseudocode from the PCT paper. This also makes the comparison fair, as all techniques are implemented on the same framework (except for the Maple algorithm). Another reason this was necessary was so that we could run the experiments on our cluster (see §6), where it is not possible to change the settings of the Linux scheduler.

Maple algorithm. The Maple tool uses a non-controlled scheduler technique by default, which we refer to as the *Maple algorithm* [Yu et al. 2012]. This algorithm performs several profiling runs, where the scheduler is not influenced, recording patterns of inter-thread dependencies through shared-memory accesses. From the recorded patterns, it predicts possible alternative interleavings that may be feasible, which are referred to as interleaving idioms. It then performs *active* runs, influencing thread scheduling to attempt to force untested interleaving idioms, until none remain or they are all deemed infeasible (using heuristics). Unlike controlled scheduling, Maple does not serialize execution. Though uncontrolled scheduling is generally beyond the scope of this work, we include the Maple algorithm in our study since it is readily available in the tool.

Busy-wait loops. A busy-wait loop (or spin loop) repeatedly checks whether another thread has written to a shared variable before continuing. These must be handled specially in systematic concurrency testing (and often in controlled scheduling) because they result in an infinite schedule where the looping thread is never preempted. To handle this, we manually inserted a call to *yield* in every busy-wait loop. We also modified Maple so that, during systematic testing (but not during random scheduling nor during PCT), a preemption was forced at every yield operation, without increasing the preemption or delay count. This is unsound, as such operations do not guarantee a preemption to another thread in practice and certain bugs may require a yield to *not* be preempted. Prior work provides a sound solution using thread priorities [Musuvathi and Qadeer 2008], as long as yield statements are added appropriately. However, due

Benchmark set	Benchmark types	# used	# skipped
CB	Test cases for real applications	3	17 networked applications.
CHESS	Test cases for several versions of a work stealing queue	4	0
CS	Small test cases and some small programs	29	24 were non-buggy.
Inspect	Small test cases and some small programs	1	28 were non-buggy.
Miscellaneous	Test case for lock-free stack and a debugging library test case	2	0
PARSEC	Parallel workloads	4	29 were non-buggy.
RADBenchmark	Tests cases for real applications	3	5 Chromium browser; 4 networking; 3 (see text).
SPLASH-2	Parallel workloads	3	9 (see text).

Table I: An overview of the benchmark suites used in the study.

to its simplicity and efficiency, and the fact that we are already testing multiple different scheduling algorithms, we used the simpler unsound approach in this study. Furthermore, for benchmarks that use busy-wait loops, the bugs manifest even if yields force a preemption (based on our understanding of the bugs) and all such bugs were indeed found by schedule bounding in our study (except the bug in `misc.safestack`, which was not found by any technique).

Busy-wait loops must also be handled specially in PCT. In the original PCT paper [Burckhardt et al. 2010], the authors state that their implementation uses heuristics to identify threads that are not making progress and lowers their priorities with a small probability. In our implementation, we change the priority of the current thread to the lowest possible priority immediately after it executes a yield operation.

4. BENCHMARK COLLECTION

We have collected a wide range of pthread benchmarks from previous work and other sources. Our benchmarks are amenable to systematic concurrency testing so that we can apply both the systematic and non-systematic techniques to all benchmarks. Thus, all benchmarks are deterministic (apart from scheduler nondeterminism). The controlled testing framework on which all techniques are implemented (except the uncontrolled Maple algorithm—§3) requires all potentially blocking functions to be modelled. As a result, including benchmarks that use network communication, inter-process communication, etc., would require significant changes to the framework, and so these benchmarks were skipped.

Table I summarizes the benchmark suites (with duplicates removed), indicating where it was necessary to skip benchmarks. “Non-buggy” means there were no existing bugs documented and we did not find any during our examination of the benchmark. We now provide details of the benchmark suites (§4.1) and barriers to the application of SCT identified through our benchmark gathering exercise (§4.2).

4.1. Details of benchmark suites

Concurrency Bugs (CB) Benchmarks [Yu and Narayanasamy 2009]. Includes buggy versions of programs such as `aget` (a file downloader) and `pbzip2` (a file compression tool). We modified `aget`, modelling certain network functions to return data from a file and to call its interrupt handler asynchronously. Many benchmarks were skipped due to the use of networking, multiple processes and signals (`apache`, `memcached`, `MySQL`).

CHESS [Musuvathi et al. 2008]. A set of test cases for a work stealing queue, originally implemented for the Cilk multithreaded programming system [Frigo et al. 1998] under Windows. The `WorkStealQueue (WSQ)` benchmark has been used frequently to evaluate concurrency testing tools [Musuvathi and Qadeer 2008; Musuvathi et al. 2008; Musuvathi and Qadeer 2007a,b; Burckhardt et al. 2010; Nagarakatte et al. 2012]. After manually translating the benchmarks to use pthreads and C++11 atomics, we found a bug in two of the tests that caused heap corruption, which always occurred when we ran the tests natively (without Maple). We fixed this bug and SCT revealed another bug that is much rarer, which we use in the study.

Concurrency Software (CS) Benchmarks [Cordeiro and Fischer 2011]. Examples used to evaluate the ESBMC tool [Cordeiro and Fischer 2011], including small multithreaded algorithm test cases (e.g. bank account transfer, circular buffer, dining philosophers, queue, stack), a file system benchmark and a test case for a Bluetooth driver. These tests included unconstrained inputs. None of the bugs are input dependent, so we selected reasonable concrete values. We had to remove or define various ESBMC-specific functions to get the benchmarks to compile.

Inspect Benchmarks [Yang et al. 2008]. Used to evaluate the INSPECT concurrency testing tool. We skipped the `swarm_isort64` benchmark, which did not terminate after five minutes when performing data race detection (see §5). There were no documented bugs, and testing all benchmarks revealed a bug in only one benchmark, `qsort_mt`, which we include in the study.

Miscellaneous. We encountered two individual test cases, which we include in the study. The `safestack` test case, which was posted to the CHESS forums[4] by Dmitry Vyukov, is a lock-free stack designed to work on weak-memory models. The bug exposed by the test case also manifests under sequential consistency, so it should be detectable by existing SCT tools. Vyukov states that the bug requires at least three threads and at least five preemptions. Previous work reported a bug that requires three preemptions [Emmi et al. 2011], which was the first bug found by CHESS that required that many preemptions.

The `ctrace` test case, obtained from the authors of [Kasikci et al. 2012], exposes a bug in the ctrace multithreaded debugging library.

PARSEC 2.0 Benchmarks [Bienia 2011]. A collection of multithreaded programs from many different areas. We used `ferret` (content similarity search) and `streamcluster` (online clustering of an input stream), both of which contain known bugs. We created three versions of `streamcluster`, each containing a distinct bug. One of these is from an older version of the benchmark and another was a previously unknown bug which we discovered during our study (see *Memory safety* in §4.2). We configured the `streamcluster` benchmarks to use non-spinning synchronization and added a check for incorrect output. All benchmarks use the "test" input values (the smallest) with two threads, except for `streamcluster2`, where the bug requires three threads.

RADBenchmark [Jalbert et al. 2011]. Consists of 15 tests that expose bugs in several applications. The 3 benchmarks we use test parts of Mozilla SpiderMonkey (the Firefox JavaScript engine) and the Mozilla Netscape Portable Runtime Thread Package, which are suitable for SCT. We skipped 9 benchmarks due to use of networking and multiple processes. Several tested the Chromium browser; the use of a GUI leads to

[4]See "Bug with a context switch bound 5" at http://social.msdn.microsoft.com/Forums/en-US/home?forum=chess

nondeterminism that cannot be controlled or modelled by any SCT tools we know of. We skipped 3 benchmarks which behave unexpectedly when running under Maple's controlled scheduling framework. We reduced the thread counts and parameter values of stress tests, as is appropriate for controlled scheduling (see *Stress tests* in §4.2). Compared to the original version of this study [Thomson et al. 2014], we compiled the RADBench benchmarks with different compiler flags so that certain provided libraries are statically linked.

SPLASH-2 [Woo et al. 1995]. Three of these benchmarks have been used in previous work [Park et al. 2009; Burckhardt et al. 2010]. SPLASH-2 requires a set of macros to be provided; the bugs are caused by a set that fail to include the "wait for threads to terminate" macro. Thus, all the bugs are similar. For this reason, we just use the three benchmarks from previous work, even though the macros are likely to cause issues in the other benchmarks. We added assertions to check that all threads have terminated as expected. We reduced the values of input parameters, such as the number of particles in `barnes` and the size of the matrix in `lu`, so the tests could run without exhausting memory. Reducing parameters as much as possible is appropriate for controlled scheduling (see *Stress tests* in §4.2); we discuss this further in §6.

4.2. Effort Required to Apply Controlled Schedulers

Recall that we restrict our benchmarks to those where we can apply SCT, so that we can apply all techniques to all benchmarks and because the controlled scheduler framework we use does not model certain external calls, such as those that perform network communication. We encountered a range of issues when trying to apply controlled scheduling to the benchmarks. These are general limitations of either SCT or concurrency testing, not of our method specifically.

Environment modelling. When applying SCT, system calls that interact with the environment, and hence can give nondeterministic results, must be modelled or fixed to return deterministic values. Similarly, depending on the framework being used, functions that can cause threads to become enabled or disabled must be handled specially, as they affect scheduling decisions. This includes the forking of additional processes, which requires both modelling and engineering effort to make the testing tool work across different processes. For the above reasons, a large number of benchmarks in the CB and RADBenchmark suites had to be skipped because they involve testing servers, using several processes and network communication. Modelling network communication and testing multiple processes are both non-trivial tasks. We believe the difficultly of controlling nondeterminism and synchronization is a key issue in applying controlled schedulers to existing code bases. However, note that non-systematic techniques can handle programs with nondeterminism and unmodelled synchronization, depending on how the techniques are implemented; for example, the random scheduler does require deterministic programs (although bug replay will be unreliable) and blocking synchronization functions can be detected approximately (on-the-fly) using heuristics [Nagarakatte et al. 2012].

Isolated concurrency testing. An alternative approach to modelling nondeterminism is to create isolated tests, similar to unit testing, but with multiple threads. Unfortunately, we found that many programs are not designed in a way that makes this easy. An example is the Apache httpd web server. The server module that we inspected had many dependencies on other parts of the server and called system functions directly, making it difficult to create an isolated test case. Developers test the server as a whole; network packets are sent to the server by a script running in a separate process. Note that it is also difficult to apply (sequential) unit testing such software.

Many applications in the CB benchmarks use global variables and function-static variables that are scattered throughout several source files. These would need to be handled carefully with SCT tools that require a repeatable function to test, such as CHESS, in which the state must be reset when the function returns. This is not a problem for Maple, which restarts the test program for every schedule explored.

Stress tests. Some of the benchmarks we obtained were stress tests, such as those in RADBench. These benchmarks create a large number of threads that undertake a significant amount of computation to increase the chance of exploring an unlikely interleaving under the OS scheduler. Increasing the amount of work is often achieved by increasing the size of the inputs or making threads execute work in a loop. In the context of controlled scheduling, this is extremely inefficient and unnecessary; instead, the number of threads and other parameters should be reduced as much as possible, as the controlled scheduler ensures that unique interleavings will be explored. Artificially increasing the thread count and parameters to make the benchmark "harder" is not representative of how one should use controlled scheduling for bug-finding; thus, we chose the minimum thread counts and parameters when converting stress tests and CPU performance benchmarks (such as the PARSEC benchmarks). In practice, one may also increase the thread count and other parameters iteratively, in case there exist bugs that depend on higher thread counts or parameter values. However, prior work suggests that most concurrency bugs only require certain orderings between a small number of threads (typically two) [Lu et al. 2008]. The only instance where we had to increase the thread count above the minimum for the bug to manifest was for `streamcluster2` from the PARSEC benchmark suite (we changed the "number of threads" parameter, t, from two to three, although note that the benchmark actually creates $2t$ threads from the main thread).

Memory safety. We found that certain concurrency bugs manifest as out-of-bounds memory accesses, which do not always cause a crash. We implemented an out-of-bounds memory access detector on top of Maple, which allowed us to detect a previously unknown bug in the PARSEC `streamcluster3` benchmark. Unfortunately, detecting out-of-bound memory accesses is a non-trivial problem and our implementation had many false-positives where memory allocation was missed or where libraries access bookkeeping information that lies outside of `malloc`ed regions. Furthermore, the extra instrumentation code caused a slow-down of up to 8x; Maple's existing information on allocated memory was not designed to be speed-efficient. We disabled the out-of-bound access detector in our experiments, but we note that a production quality concurrency testing tool would require an efficient method for detecting out-of-bound accesses to automatically identify this important class of bug. We manually added assertions to detect the (previously unknown) out-of-bounds access in `streamcluster3` and the (previously known) out-of-bounds access in `fsbench_bad` in the CS benchmarks. Out-of-bounds accesses to synchronization objects, such as mutexes, are still automatically detected; this was used to detect the bug in `pbzip2` from the CS benchmarks.

Data races. We found that 30 of the 49 benchmarks contained data races. There are many compelling arguments against the tolerance of data races [Boehm 2011], and according to the C++11 standard, if it is possible for a program execution to lead to a data race, the behavior of the program for this execution is undefined. Nevertheless, at the level of program binaries, data races do not result in undefined behavior and many data races are not regarded as bugs by software developers. Treating data races as errors would be too easy for benchmarking purposes, as they hide the more challenging bugs that the benchmarks capture. A particular pattern we noticed was that data races often occur on flags used in ad-hoc busy-wait synchronization, where one thread

keeps reading a variable until the value changes. At the C++ level, the "benign" races could be rectified through the use of C++11 relaxed atomics, the "busy-waits" could be formalized using C++11 acquire/release atomics, and synchronization operations could be added to eliminate the buggy cases. However, telling the difference between benign and buggy data races is non-trivial in practice [Kasikci et al. 2012; Narayanasamy et al. 2007]. We explain how we treat data races in our study in §5.

Output checking. The bugs in the benchmarks `CB.aget` and `parsec.streamcluster2`, lead to incorrect output, as documented in the bug descriptions. Thus, we added extra code to read the output file and trigger an assertion failure when incorrect; the output checking code for the `CB.aget` was provided as a separate program, which we added to the benchmark. Several of the PARSEC and SPLASH benchmarks do not verify their output, greatly limiting their utility as test cases.

5. EXPERIMENTAL METHOD

Our experimental evaluation aims to compare unbounded depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB), controlled random scheduling (Rand) and probabilistic concurrency testing (PCT). We also test the default Maple algorithm (MapleAlg). Bugs are deadlocks, crashes, assertion failures and incorrect output. Each benchmark contains a single concurrency bug.

For each controlled scheduling technique, we use a limit of 100,000 schedules to enable a full experimental run over our large set of benchmarks to complete on a cluster within one month. There are two exceptions: for the `chess.IWSQWS` and `chess.SWSQ` benchmarks (see Table III and IV) we use a schedule limit of 10,000 as in the conference version of this work [Thomson et al. 2014]; for these longer-running benchmarks, evaluation with the higher schedule limit exceeded our one month time restriction.

We chose to use a schedule limit instead of a time limit because there are many factors and potential optimization opportunities that can affect the time needed for a benchmark to complete; we believe that the time variance for the different controlled techniques (for execution of a single schedule of a given benchmark) is negligible, assuming reasonably optimized implementations. Furthermore, the cluster we have access to shares its machines with other jobs, making accurate time measurement difficult. In contrast, the number of schedules explored cannot be improved upon, without changing key aspects of the search algorithms themselves. By measuring the number of schedules, our results can potentially be compared with other algorithms and future work that use different implementations with different overheads.

Each benchmark goes through the following phases:

Data Race Detection Phase. In the context of systematic concurrency testing, it is sound to only consider scheduling points before each synchronization operation, such as locking a mutex, as long as execution aborts with an error as soon as a data race is detected [Musuvathi et al. 2008]. Thus, if there are data races, an error will be reported; if there are no data races and the search completes, then the program is free from safety property violations (such as assertion failures and deadlocks). This greatly reduces the number of schedules that need to be considered. However, treating data races as errors is not practical for this study due to the large number of data races in the benchmarks (see §4.2), which would make bug-finding trivial and arguably not meaningful.

As in previous work [Yu et al. 2012], we circumvent this issue by performing dynamic data race detection to identify a subset of load and store instructions that are known to participate in data races. We treat these instructions as visible operations during concurrency testing by inserting scheduling points before them. For each benchmark, we execute Maple in its data race detection mode ten times, without controlling the

schedule. Each racy instruction (stored as an offset in the binary) is treated as a visible operation in subsequent phases. We also tried detecting data races during concurrency testing, but this caused an additional slow-down of up to 8x, as Maple's data race detector is not optimized for this scenario.

Thus, the techniques explore the sequentially consistent outcomes of a subset of the possible data races for a concurrent program. Bugs found by this method are real (there are no false-positives), but bugs that depend on relaxed memory effects or data races not identified dynamically will be missed. We do not believe these missed bugs threaten the validity of our comparison, since the same information about data races is used by all of the techniques (excluding the Maple algorithm); the set of racy instructions could be considered as part of the benchmark.

An alternative to under-approximation would be to use static analysis to over-approximate the set of racy instructions. We did not try this, but speculate that imprecision of static analysis would lead to many instructions being promoted to visible operations, causing schedule explosion.

Note that the data races detected and used in our experiments are likely to be different from those in the original study [Thomson et al. 2014] because the data race detection phase is not deterministic.

Depth-First Search (DFS) Phase. We next perform SCT using a depth-first search, with no schedule bounding and a limit of 100,000 schedules.

Iterative Preemption Bounding (IPB) Phase. We next perform SCT on the benchmark using iterative preemption bounding. By repeatedly executing the program, restarting after each execution, we first explore all terminal schedules that have zero preemptions, followed by all schedules that precisely one preemption, etc., until either the limit of 100,000 schedules is reached, all schedules have been explored or a bug is found. If a bug is found, the search does not terminate immediately; the remaining schedules within the current preemption bound are explored (for our set of benchmarks, it was always possible to complete this exploration without exceeding the schedule limit). This allows us to check whether non-buggy schedules could exceed the schedule limit when an underlying search strategy other than depth-first search is used (see §3).

In practice, all SCT tools that we are aware of do not perform iterative preemption bounding in this manner. Instead, with a preemption bound of c, it is necessary to explore all schedules with c *or fewer* preemptions due to the use of a depth-first search. Thus, iterative preemption bounding will explore all schedules with 0 preemptions, followed by all schedules with 0–1 preemptions (redundantly re-exploring schedules with 0 preemptions), followed by all schedules with 0–2 preemptions (redundantly re-exploring schedules with 0–1 preemptions), etc. In our study, we simulate an optimized version of preemption bounding that does *not* redundantly re-explore schedules with fewer than c preemptions. We achieve this simply by ignoring previously explored, and thus redundant, schedules when processing our log files. We chose to do this because it might be possible to implement such an algorithm in practice and we did not want to unfairly penalize the technique due to the specific implementation that we used.

Iterative Delay Bounding (IDB) Phase. This phase is identical to the previous, except delay bounding is used instead of preemption bounding.

Random scheduler (Rand) Phase. We run each benchmark 100,000 times using Maple's controlled random scheduler mode. This allows us to compare the other techniques against a naïve controlled scheduler. Recall that the random scheduler may re-explore schedules.

Probabilistic Concurrency Testing (PCT) Phase. Recall that PCT requires parameters n (maximum number of threads), k (maximum execution steps) and d (the "bug depth", which controls the number of priority change points that will be chosen). In order to experiment with PCT using varying values for d, it was necessary to obtain reasonable estimates for n and k. We obtained these estimates for each benchmark as follows. First, we used results related to SCTBench obtained in prior work to provide initial estimates for n and k—see Table 3, column "# threads" and "# max scheduling points" in [Thomson et al. 2014]. Using these initial estimates we executed 1000 schedules of the benchmark using PCT with d=3. We chose d=3 as we believed that this would increase the amount of interleaving, potentially increasing the chance of observing different execution lengths. During these executions we recorded the maximum observed number of threads and the maximum observed number of steps; we start counting steps from when the initial thread first launches a second thread. We used these values for n and k, respectively, in our experiments[5].

Unlike the other bounded techniques, there is no obvious way to perform iterative PCT. In order to provide a thorough evaluation of PCT, we experimented with each d in $\{1, 2, 3\}$, using PCT to run each benchmark for 100,000 executions for each value of d. We present each version of PCT (parameterized with a value for d) as a separate technique.

Maple Algorithm (MapleAlg) Phase. We test each benchmark using the Maple algorithm. This algorithm terminates based on its own heuristics; we enforced a time limit of 24 hours per benchmark, although execution only took this long due to livelocks.

Notes on depth-first search and partial-order reduction. As discussed in §3, the systematic methods we evaluate are built on top of Maple's default depth-first search strategy. Although depth-first search is just one possible search strategy, and different strategies could give different results, we argue that this is not important in our study. First, if the depth-first search biases the search for certain benchmarks, then both schedule bounding algorithms are likely to benefit or suffer equally from this. Second, iterative schedule bounding explores *all* schedules with c preemptions/delays before *any* schedule with $c + 1$ preemptions/delays. This means that when the first schedule with $c + 1$ preemptions/delays is considered, exactly the same set of schedules, regardless of search strategy, will have been explored so far. Thus, if a bug is revealed at bound c then, by exploring *all* schedules with bound c (as described above), we can determine the worst case number of schedules that might have to be explored to find a bug, accounting for an adversarial search strategy.

Partial-order reduction (POR) [Godefroid 1996] is a commonly used technique in concurrency testing [Musuvathi et al. 2008; Musuvathi and Qadeer 2007b; Flanagan and Godefroid 2005; Godefroid 1996]. We do not attempt to study the various POR techniques in this work. This is because (a) our principle aim was to validate the findings of prior works on controlled scheduling, most of which do not incorporate full partial-order reduction (and indeed, the relationship between POR and schedule bounding is complex [Coons et al. 2013; Musuvathi and Qadeer 2007b; Holzmann and Florian 2011]), and (b) each partial-order reduction technique considered would approximately double the number of technique configurations amenable to experimental comparison. Nevertheless, we believe that a study incorporating partial-order reduction methods would be valuable future work. In the context of our experimental setup, this would

[5]We note that, in hindsight, this may be an unrealistic approach to obtaining the parameters. A better approach would be as follows: (1) Choose any values for n and k. (2) Execute PCT for e.g. 1,000 schedules, recording the maximum observed number of threads and steps. (3) Update n and k based on what was observed. (4) Repeat the process to refine the values for n and k.

involve implementing dynamic partial-order reduction (the state-of-the-art) [Flanagan and Godefroid 2005] in Maple.

Notes on randomization. The controlled random scheduler and PCT techniques both use a random number generator. Given one of these techniques, a seed (used to initialize the random number generator) and a benchmark, the (single) schedule executed by the technique for the benchmark is deterministic. Unlike with the systematic techniques, which operate with respect to an underlying DFS algorithm with iteratively increasing preemption or delay bounds, for the random techniques there is no implied order between schedules: two different seeds result in two independent schedules that can be tested in parallel.

Our method for testing the controlled random scheduler and PCT techniques was as follows. We used a fixed initial seed to generate a single list of 100,000 seeds using a random number generator; we used these same seeds for all benchmarks and for all randomized techniques to produce 100,000 schedules in each case.

For a given benchmark, we can use the number of buggy schedules out of 100,000 (i.e. the proportion of buggy schedules) to compare the random-based techniques; this is dependent on the initial seed but, as the schedule limit is increased, we would expect this to become stable. We can also use the number of schedules before the bug is found. However, this is very dependent on the initial seed and a technique may "get lucky" for some benchmarks. Thus, we can instead consider the "average number of schedules needed to expose a bug", calculated using: 100,000 / "number of buggy schedules"; this shows how many schedules are likely to be needed *on average* before a bug is found. As the schedule limit is increased, we would expect this number to become stable and, thus, be independent of the initial seed.

6. EXPERIMENTAL RESULTS

We conducted our experiments on a Linux cluster, with Red Hat Enterprise Linux Server release 6.4, an x86_64 architecture and gcc 4.7.2. Our modified version of Maple is based on the last commit from 2012.[6] The benchmarks, scripts and the modified version of Maple used in our experiments can be obtained from http://sites.google.com/site/sctbenchmarks.

Throughout this section, we use **RQ1–RQ7** to indicate that an observation relates to one of the research questions posed in §1. When we refer to x buggy schedules, we mean the x schedules executed by a particular technique that found the bug in a given benchmark. When we refer to x bugs being found by a technique, we mean that the technique found a bug in x of the benchmarks.

For **RQ6**, we refer the reader to §4.2, where we discuss the difficultly of applying controlled scheduling to the benchmarks.

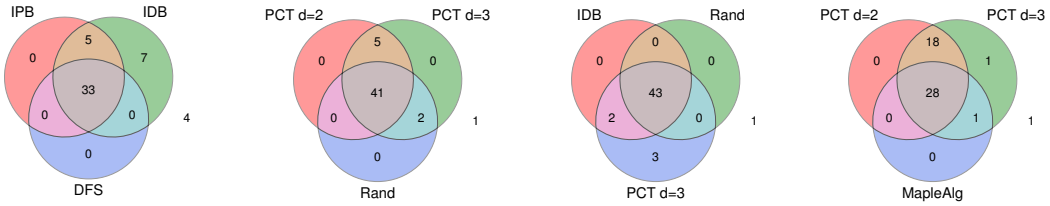
6.1. Venn diagrams

The Venn diagrams in Figure 3 give a concise summary of the bug-finding ability of the techniques in terms of number of bugs found in SCTBench within the schedule limit.

Figure 3a summarizes the bugs found by the systematic techniques. In relation to **RQ4**, the figure shows that IPB was superior to DFS, finding all 33 bugs found by DFS, plus an additional 5. The figure also shows, also in relation to **RQ4**, that IDB found all 38 bugs found by IPB, plus an additional 7. The bugs in 4 benchmarks were missed by all systematic techniques; we discuss this further below.

Figure 3b shows the bugs found by the non-systematic techniques, PCT and Rand. We show the results for PCT with d=2 and d=3 because PCT found the most bugs when

[6] http://github.com/jieyu/maple commit at Sept 24, 2012



(a) Systematic. (b) Non-systematic. (c) Systematic vs. non-systematic. (d) MapleAlg.

Fig. 3: Venn diagrams showing number of benchmarks in which the bugs were found with the various techniques.

using these values for d. The results show that PCT d=3 performed the best in terms of number of bugs found within the schedule limit, finding 48 bugs, including all those found by the other techniques (see Figure 3c and 3d also). Thus, in answer to **RQ1** and **RQ2**, the results show that PCT d=3 is the most capable technique at finding bugs in SCTBench; this concurs with findings of prior work in which PCT found bugs faster than IPB [Burckhardt et al. 2010].

Figure 3c shows the bugs found by the superior schedule bounding technique (IDB), the random scheduler (Rand) and PCT with d=3 (the most successful configuration of PCT). Note that the bugs in 43 benchmarks were found by both IDB and Rand, and IDB found just 2 additional bugs that were missed by Rand. Although not shown in these diagrams, Rand also found *all* the bugs found by IPB, plus an additional 5. Thus, in answer to **RQ3**, Rand performed better than IPB in terms of number of bugs found and was not far behind IDB. Furthermore, Rand found the bugs in fewer schedules than IDB for 21 of the benchmarks. A similar observation can be made about IPB and Rand. Thus, Rand was often faster at finding bugs than schedule bounding. We discuss the surprising results for Rand below.

Figure 3d shows the bugs found by MapleAlg vs. PCT d=2 and PCT d=3. Maple found 29 of the 49 bugs (all of which were also found by PCT d=3) and missed 19 bugs that were found by PCT d=3.

The bug in `misc.safestack` was missed by *all* techniques; we discuss this in more detail below.

6.2. Cumulative plots

The graphs in Figures 4 and 5 give an alternative summary of the techniques.

Figure 4 is a *cumulative* plot showing the number of bugs found (y-axis) after x schedules (x-axis) for each technique over all the benchmarks. Each line represents a technique and is labelled by the name of the technique and the number of bugs found by the technique within the schedule limit. If a given technique has a point at coordinate (x, y) then there were y benchmarks for which the technique was able to expose a bug using x schedules or fewer, i.e. for which "number of schedules to first bug" is less than or equal to x. This plot show the the numbers of bugs that would be found by the techniques using schedule limits lower than 100,000. For example, with our schedule limit of 100,000, IDB and Rand found 45 and 43 bugs, respectively; with a schedule limit of 1,000, they would have found 40 and 42 bugs, respectively.

As explained in §5, the Rand and PCT results are specific to the random seeds used during our experiments. Thus, in Figure 5, we present results using the average number of schedules needed to expose a bug, which is given by: 100,000 / "number of buggy schedules". Figure 5 is similar to Figure 4, but includes only PCT d=3 and Rand (oth-

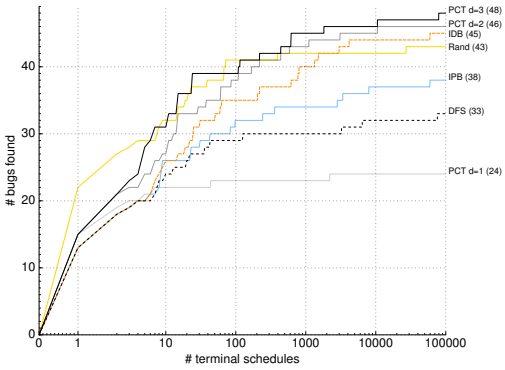


Fig. 4: Cumulative plot, showing, for each controlled scheduling technique, the number of bugs found after x schedules over all the benchmarks. The plot is intended to be viewed in color.

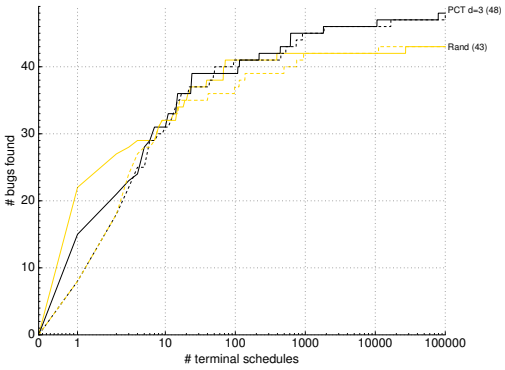


Fig. 5: For PCT d=3 and Rand, compares the number of bugs found after x schedules as in Figure 4 (solid lines) with the *average* behavior of the techniques (dashed lines). The plot is intended to be viewed in color.

erwise, the graph is overcrowded). The additional dashed lines show the *average* behavior of the techniques.

Observe that, in Figure 4, the ordering of the techniques by number of bugs found remains fairly consistent for schedule limits above 1,000, the exception being IDB and Rand, with IDB overtaking Rand in terms of bug-finding ability at 2990 schedules. In Figure 5, the same is true when considering the average behavior of the techniques. Thus, the number of bugs found by the techniques within our schedule limit is, for the most part, an accurate reflection of the bug finding ability of the techniques on our benchmarks.

Our results show that PCT d=3 almost invariably finds more bugs than the other techniques, unless the schedule limit is extremely low. Thus, our findings for **RQ1** and **RQ2** apply for a range of schedule limits. One exception is that Rand overtook PCT d=3 at 100 schedules, but in the average case (Figure 5), PCT d=3 is still consistently above Rand when the schedule limit is 20 or higher. Regarding **RQ4**, the findings that IDB found more bugs than IPB and that IPB found more bugs than DFS both hold for schedule limits of 50 or higher; the difference in bugs found between these techniques increased with the schedule limit. Similarly, for **RQ3**, Rand beat IPB for *all* schedule limits up to and including 100,000, showing that this finding is not simply due to our choice of schedule limit. In the average case, Rand beat IPB for for all schedule limits of 10 or greater, indicating that for non-trivial limits this finding is independent of our choice of initial random seed. Rand was also ahead of IDB in terms of bugs found between schedule limits 1–1,000, giving further evidence for **RQ3** that Rand performed well. In fact, Rand found 27 bugs in the first 2 schedules and was ahead of all other techniques by at least 6 bugs; Figure 5 shows that this is not the case on average, but after 10 schedules, both Rand and averaged Rand found 32 bugs, which is the same as PCT d=3 (and more than all other techniques). The fact that Rand finds many of the bugs so quickly is evidence of the trivial nature of some of the benchmarks (**RQ7**), which we discuss in §6.4.

Regarding the average behavior of PCT d=3 and Rand (Figure 5), both techniques still performed well and our main conclusions do not change. We can see that Rand

was slightly "lucky" between 10–1,000 schedules compared to the average case and was slightly "unlucky" at finding the bug after 10,000 schedules.

6.3. Results tables

The full set of experimental data gathered for our benchmarks is shown in Tables III and IV. As explained in §5, we focus on the number of schedules explored rather than time taken for analysis. The execution time of a single benchmark varied between 1-10 seconds depending on the benchmark. The longest time taken to perform ten data race detection runs for a single benchmark was five minutes, but data race detection was significantly faster in most cases. Data race detection could be made more efficient using an optimized, state-of-the-art method. Because data race analysis results are shared between all techniques (except MapleAlg), the time for data race analysis is not relevant when comparing these methods.

For each benchmark, # *max threads* and # *max enabled threads* show the total number of threads launched and the maximum number of threads simultaneously enabled at any scheduling point, respectively. The # *max steps* column shows the maximum number of scheduling points (visible operations) k observed from when the initial thread first launches a second thread. As explained in §5, these numbers were obtained by running 1000 executions of PCT on the benchmarks.

Results for systematic techniques. In Table III, the smallest preemption or delay bound required to find the bug for a benchmark, or the bound reached (but not fully explored) if the schedule limit was hit, is indicated by *bound*; # *schedules to first bug* shows the number of schedules that were explored up to and including the detection of a bug for the first time; # *schedules* shows the total number of schedules that were explored; # *new schedules* shows how many of these schedules have exactly *bound* preemptions (for IPB) or delays (for IDB); # *buggy schedules* shows how many of the total schedules explored exhibited the bug. As explained in §5, when a bug is found during IPB or IDB, we continue to explore all buggy and non-buggy schedules within the preemption or delay bound; the schedule limit was never exceeded while doing this. An *L* entry denotes 100,000 (the schedule limit discussed in §5). When no bugs were found, the bug-related columns contain ✗. We indicate by *% buggy*, the percentage of schedules that were buggy out of the total number of schedules explored during DFS. We prefix the percentage with a '*' when the schedule limit was reached, in which case the percentage applies to all explored schedules, not the total number of possible schedules.

Results for non-systematic techniques. For the Rand and PCT techniques in Table IV, the # *schedules* column is omitted, as it is always 100,000 (although, as explained in §5, the `chess.IWSQWS` and `chess.SWSQ` benchmarks use a lower schedule limit of 10,000). This is because these techniques do not maintain a history of explored schedules and thus there is no notion of the search terminating. The # *schedules to first bug* column shows the number of schedules that were explored up to and including the detection of a bug for the first time. The # *buggy schedules* column shows how many of the 100,000 schedules exhibited a bug. For each value of d that we used for PCT and for each benchmark, we estimate the worst case (smallest) number of buggy schedules that we should find given a bug of depth d, parameters n and k from the benchmark, and our schedule limit of 100,000. This estimate is shown under *est. worst case* # *buggy* in Table IV, and is calculated by computing the worst-case probability that an execution using PCT will expose a depth-d bug (using the formula $1/nk^{d-1}$ discussed in §2.7) and multiplying this probability by 100,000 (the schedule limit). Of course, the estimate for each d is only relevant if the bug associated with a benchmark can in fact manifest with depth d.

Property	# benchmarks
Bug was found with a delay bound of 0	13
Total number of schedules $<$ 100,000	18
$>$ 50% of random terminal schedules were buggy	18
Every random terminal schedule was buggy	8

Table II: Benchmarks where bug-finding is arguably trivial.

For the Maple algorithm, we report whether the bug was found (the *found?* column in Table IV), the total number of (not necessarily distinct) schedules explored, as chosen by the algorithm's heuristics, and the total time in seconds for the algorithm to complete. Benchmarks 32, 33 and 34 caused Maple to livelock, so the 24 hour time limit was exceeded. We indicate this with '-'.

6.4. Benchmark Properties

The # *max threads* and # *max steps* columns from the results tables can be used to estimate the total number of schedules, which may shed light on the complexity of a given benchmark. With at most n enabled threads and at most k steps, there are at most n^k terminal schedules. On the other hand, if most of the schedules are buggy then the number of schedules is not necessarily a good indication of bug complexity. For example, `CS.din_phil2_sat` has a relatively high number of schedules, but since 87% of them are buggy (see the DFS results in Table III), this bug is trivial to find. Of course, the majority of benchmarks cannot be explored exhaustively, and estimating the percentage of buggy schedules from the partial DFS results is problematic because DFS is biased towards exploring deep context switches.

To answer **RQ7**, we present Table II which provides some further insight into the complexity of the benchmarks, using properties derived from Tables III and IV. Bugs found with a delay bound of zero (13 cases) will always be found on the initial schedule for IPB, IDB and DFS, as they all initially execute the same schedule. Any technique based on this same depth-first schedule will also find the bug immediately. It could be argued that this schedule is effective at finding bugs, or that the bugs in question are trivial, since the schedule includes minimal interleaving (there are no preemptions). Benchmarks with fewer than 100,000 schedules total (as measured by unbounded DFS, which is exhaustive) will always be exhaustively explored (and so the bug will be found) by all systematic techniques (18 cases). Techniques can still be compared on how quickly they find the bugs in such benchmarks. Note that the two `chess` benchmarks that were explored using a schedule limit of 10,000 do not have fewer than 100,000 schedules. Bugs that were exposed more than 50% of the time when using the random scheduler could arguably be classified as "easy-to-find" (18 cases). Among these, bugs that were exposed 100% of the time when using the random scheduler (8 cases) are almost certainly trivial to detect; indeed, Tables III and IV show that all of these benchmarks were buggy for all schedules explored by *all* techniques. For 5 of these benchmarks, DFS was exhaustive, showing that these bugs are not even schedule-dependent. Note that the `CS.din_phil7_sat` benchmark contains fewer schedules than the smaller versions of this benchmark and has 100% buggy schedules according to DFS. This is because `CS.din_phil7_sat` contains an additional, unintentional bug introduced by the original authors of the benchmark; when we converted the benchmark to use (non-recursive) pthread mutexes, the bug causes additional deadlocks. We did not fix this additional bug and instead used the benchmark as it was found.

Regarding **RQ7**: in our view the relatively trivial nature of some of the bugs exhibited by our benchmarks has not been made clear in prior work that studies these examples (prior to the conference version of this article [Thomson et al. 2014]). The

controlled random scheduler can detect many of the bugs with a high probability. We regard these easy-to-find bugs as having value only in providing a minimum baseline for any respectable concurrency testing technique. Failure to detect these bugs would constitute a major flaw in a technique; detecting them does not constitute a major achievement.

6.5. Techniques In Detail

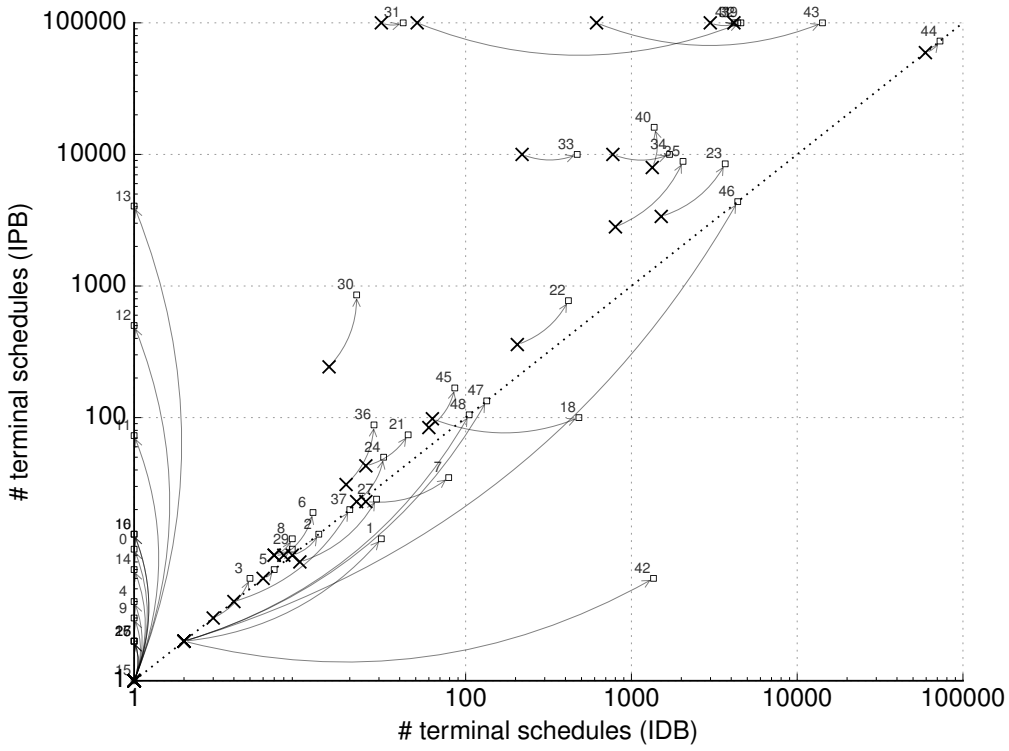


Fig. 6: Comparison of IPB (y-axis) and IDB (x-axis), showing the number of schedules to the first bug (cross) connected to the total number of schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

IPB vs. IDB. Figure 6 compares IPB and IDB by plotting data from the following columns in Table III: # *schedules to first bug* (as a cross) and # *schedules* (as a square). All benchmarks are shown for which at least one of the techniques found a bug. A benchmark is depicted as a line connecting a cross and a square. Each square is labelled with its benchmark *id* from Table III. Where the bug was not found by one of the techniques, this is indicated with a cross at 100,000 (the schedule limit discussed in §5). However, as described in §5, benchmarks 33 and 34 used a schedule limit of 10,000 and so the crosses for these benchmarks on the line $y =$10,000 indicate that IPB hit the schedule limit without finding the bug. The cross indicates which technique was faster at finding the bug; crosses below/above the diagonal indicate that IPB/IDB was faster. The square indicates how many schedules exist with a bound less than or equal to the

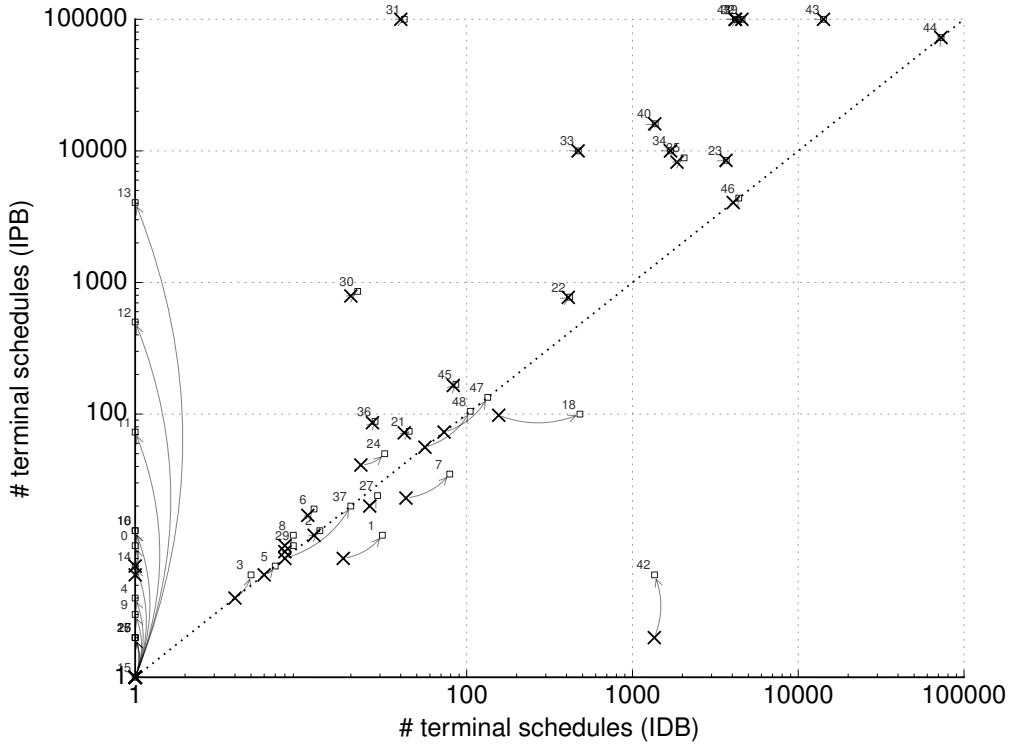


Fig. 7: Comparison of IPB (y-axis) and IDB (x-axis), showing total number of non-buggy schedules (cross) connected to the total number of schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

bound that found the bug. For example, when exploring benchmark 30 with IPB, the first buggy schedule is found after 243 schedules. This schedule involves one preemption, so the search continues until all 856 schedules with at most one preemption have been explored (bound at which the bug was found). Since the search terminated before reaching the schedule limit, we know that the bug would be found within the first 856 schedules even if we were using an underlying search strategy other than depth-first search. Notice that a number of benchmarks appear at $(x, 100{,}000)$, with $x <$100,000: this is where IPB failed to find a bug and IDB succeeded (except for benchmarks 33 and 34, as explained above).

The bug-finding ability of the techniques in Figure 6 is tied to the underlying depth-first search. It is possible that this might cause one of the techniques to "get lucky" and find a bug quickly, while another search order could lead to many additional non-buggy schedules being considered before a bug is found. To avoid this implementation-dependent bias, in Figure 7 we consider the *worst-case* bug-finding ability. For each benchmark, a cross plots, for IDB and IPB, the total number of *non-buggy* schedules within the bound that exposed the bug. This corresponds to the difference between the # *schedules* and # *buggy schedules* columns presented in Table III, and represents the worst-case number of schedules that might have to be explored to find a bug, given an unlucky choice of search ordering. The squares are the same as in Figure 6.

Overall, IDB finds all bugs found by IPB, plus an additional seven. Regarding **RQ4**: in Figure 6, most crosses fall on or above the diagonal, showing that IDB was as fast or faster than IPB in terms of number of schedules to the first bug. The same is mostly true for the squares, showing that IDB generally leads to a smaller total number of schedules than IPB (up to the bound at which the bug was found). In the worst case (Figure 7), some crosses fall under the line, but most are still very close, or represent a small number of schedules (less than 100) where the difference between the techniques is negligible. An outlier is benchmark 42 where, *in the worst case*, IPB requires 3 schedules to find the bug, while IDB requires 1356 schedules. Table III shows that the bug does not require any preemptions, but requires at least one delay; this difference greatly increases the number of schedules for IDB. We believe this can be explained as follows. First, there must be a small number of blocking operations, leading to a very small number of schedules with a preemption bound of zero. Second, the bug in question requires that when two particular threads are started and reach a particular barrier, the "master" thread (the thread that was created before the other) does *not* leave the barrier first. With zero preemptions, the non-master thread can be chosen at the first blocking operation (as any enabled thread can be chosen). With zero delays, only the master thread can be chosen, as one delay is required to skip over the master thread. Thus, this is an example where IDB performs worse than IPB. Nevertheless, IDB is still able to find the bug within the schedule limit.

The `CS.reorder_X_bad` benchmark (where X is the number of threads launched – see Table III) is the adversarial delay bounding example given in Figure 2 in §2.4; the smallest delay bound required for the bug to manifest is incremented as the thread count is incremented. However, IDB still performs better than IPB, as the number of schedules in IPB increases exponentially with the thread count. Furthermore, this is a synthetic benchmark for which the bug is found quickly by both techniques with a low thread count.

Effectiveness of controlled random scheduling. In answer to **RQ3**, we have shown above that Rand is surprisingly effective, finding more bugs than IPB and almost as many as IDB. The cumulative plots in Figure 4 and 5 show that these findings apply on average and for various schedule limits. A possible intuition for this is as follows. If a bug can be exposed with just one preemption, say, then there may be a number of scheduling points at which the preemption can occur so that the bug can be exposed. Furthermore, there may be a number of "unexpected" operations in other threads that will cause the bug to trigger (e.g. writing to a variable that the preempted thread is about to access). Any schedule where (a) the preemption occurs in a suitable place, and (b) additional preemptions do not prevent the bug from occurring, will also expose the bug. There may be many such schedules and thus a good chance of exposing the bug through random scheduling. More generally, if a bug can be exposed with a small delay or preemption count, there may be a high probability that a randomly selected schedule will expose the bug. A counter-example is the bug in the `parsec.ferret` benchmark, which is missed by Rand but found by IDB. The bug requires a thread to be preempted early in the execution and not rescheduled until other threads have completed their tasks. Since Rand is very likely to reschedule the thread, it is not effective at finding this bug. For IDB, only one delay is required, but, as seen in Table III, only one buggy schedule was found; thus, the delay must occur at a specific scheduling point for the bug to manifest.

The CHESS benchmarks test several versions of a work stealing queue. They were used for evaluation in the introduction of preemption bounding [Musuvathi and Qadeer 2007a] and thus were used to show the effectiveness of preemption bounding as a bug finding technique. Depth-first search fails to find the bug in `chess.WSQ`, while

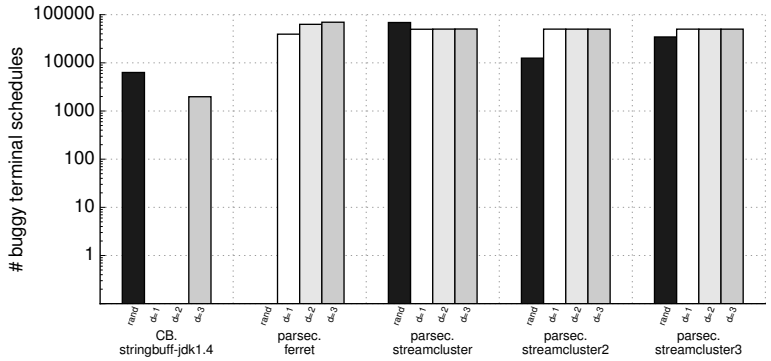


Fig. 8: Shows, for the `stringbuff-jdk1.4` and `parsec` benchmarks, the number of buggy schedules explored by Rand, and PCT for each value of $d \in \{1, 2, 3\}$. Each technique explored 100,000 schedules.

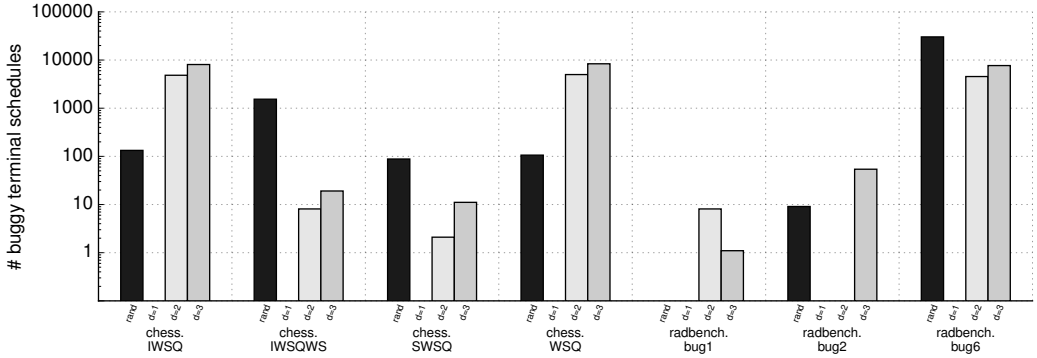


Fig. 9: Shows, for the `chess` and `radbench` benchmarks, the number of buggy schedules explored by Rand, and PCT for each value of $d \in \{1, 2, 3\}$. Each technique explored 100,000 schedules (except for `chess.IWSQWS` and `chess.SWSQ`, which use a schedule limit of 10,000).

IPB succeeds (as in prior work). The remaining CHESS benchmarks are more complex (lock-free) versions of `chess.WSQ`, which were also used in prior work. IPB and DFS fail to find the bugs in these benchmarks, while IDB is successful (which is relevant to **RQ4**). However, Rand is able to find all the bugs in these benchmarks (like IDB) *and* it also finds them in fewer schedules than IDB and IPB (which is highly relevant to **RQ3**). The prior work that introduced these techniques did not compare against a random scheduler in terms of bug finding ability.

Effectiveness of Probabilistic Concurrency Testing. Figure 8 and Figure 9 compare the effectiveness of Rand and PCT for each value of $d \in \{1, 2, 3\}$ at finding bugs for a subset of benchmarks; the subset is not representative of all the benchmarks—we focus on benchmarks for which the probabilistic results are notable and worthy of discussion. The bars show the number of buggy terminal schedules exposed by the techniques within 100,000 schedules (except for `chess.IWSQWS` and `chess.SWSQ`, which use a schedule limit of 10,000). The graphs uses a log scale for the y-axis. Regarding **RQ2**

and **RQ3**, it is interesting to see that Rand is often similar and sometimes better than PCT in terms of number of buggy schedules found. As explained above, we conjecture that in these cases, there are probably many places at which preemptions can occur to allow the bug to manifest and many opportunities for an unexpected operation in a different thread to occur after the preemption. Nevertheless, PCT with d=3 finds all the bugs that Rand finds, plus an additional four, as shown in Figure 3b.

Recall that the PCT algorithm inserts $d - 1$ priority change points (see §2.7). Looking at `CB.stringbuffer-jdk1.4` in Figure 8, we can see that this bug was found by both Rand and PCT d=3. Looking at Table III, this benchmark only has 2 threads and around 10 execution steps, but the bug requires at least 2 preemptions or delays to occur. Note that, in the PCT algorithm, a lower priority thread T1 can enable a higher priority thread T2, in which case T2 will preempt T1, without the need for a priority change point. Nevertheless, for this benchmark, it seems likely at least two priority change points are needed for the bug to occur, which would explain why PCT did not find the bug with $d < 3$. Interestingly, Rand is more effective at finding this bug than PCT. The bug requires a preemption away from thread 1 and then a preemption away from thread 2 so that execution of thread 1 continues. Unfortunately, due to the way in which PCT lowers priorities, the second priority change point may not change the priority ordering between the two threads—it depends on the priorities assigned to the priority change points. For example, assume $d = 3$ and an initial priority mapping of $\{T2 \rightarrow 3, T1 \rightarrow 4\}$, so that T1 has the highest priority. Let the first priority change point change T1's priority to 1, giving a priority mapping of $\{T1 \rightarrow 1, T2 \rightarrow 3\}$ and making T2 the new highest priority thread. Let the second priority change point change T2's priority to 2, giving a priority mapping of $\{T1 \rightarrow 1, T2 \rightarrow 2\}$. The second change point does not change the relative priority ordering between the threads. We speculate that this is the reason why PCT is less effective. This possibly highlights a weakness of the PCT algorithm; on the other hand, PCT was designed carefully to ensure the probabilistic guarantee described in §2.7, so "fixing" this issue while maintaining the guarantee may be non-trivial.

Similar observations can be made about the other benchmarks in Figure 8 and Figure 9 by cross-referencing with Table III; if a bug requires c preemptions or delays, then the bug will usually not be found by PCT with $d - 1 < c$ (fewer than c priority change points). We stress that this is not the case in general; an exception is the bug in `chess.WSQ`, which requires 2 preemptions, but was found by PCT d=2 (only 1 priority change point). Thus, this is an example where a lower priority thread unblocks a higher priority thread, resulting in a preemption. We speculate that this is because the benchmark involves blocking locks (the other CHESS benchmarks use spin locks). Similarly, `radbench.bug2` requires 3 preemptions, but was found with PCT d=3 (2 priority change points).

For many of the benchmarks in shown in Figure 8 and Figure 9, increasing d makes PCT more effective at finding bugs; this suggests that these bugs require certain change points at the right places, but additional change points are unlikely to prevent the bug from occurring. A good example is `parsec.ferret` which, as explained above, requires a thread to be preempted early in the execution and not rescheduled until other threads have completed their tasks. Unlike Rand, PCT is ideally suited to exposing this bug; once the required thread has its priority lowered, it will only be scheduled instead of other enabled threads if all other enabled threads also have their priorities lowered; this benchmark has, on average, 4 enabled threads. Thus, as long as $d < 5$, increasing d simply increases the chance of one of the priority change points occurring at the right place.

The `radbench.bug1` benchmark was found by IDB, PCT d=2 and PCT d=3; very few buggy schedules were found by PCT. The bug requires a thread to be preempted after

destroying a hash table and a second thread to access the hash table, causing a crash; this explains why the bug requires only one delay and why PCT was able to find it with at least one priority change point. It is likely that the large number of scheduling points is what pushes this bug out of reach of the other techniques. PCT d=3 found 7 buggy schedules in `radbench.bug2`; the description of this bug is less clear [Jalbert et al. 2011]. This bug and the bug in `CB.stringbuffer-jdk1.4` are the only ones found by PCT that appear to require d=3 (i.e. 2 priority change points).

PCT d=2 and PCT d=3 were the only techniques to find the bugs in `CS.twostage_100_bad`, `CS.reorder_10_bad` and `CS.reorder_20_bad`. However, these benchmarks have identical counterparts with lower thread counts. Recall that, in controlled concurrency testing, the thread count should be decreased as much as possible while still capturing an interesting concurrency scenario. Thus, these benchmarks are perhaps not realistic test cases for controlled scheduling. Furthermore, IDB found the bugs in the versions of these benchmarks with lower thread counts, plus all the other bugs that were found by PCT d=3. If we ignore these "high thread count" benchmarks, then IDB and PCT found the same number of bugs within the schedule limit; thus, it could be argued that IDB performed similarly to PCT, which is relevant to **RQ1** and **RQ2**. Nevertheless, PCT found these bugs directly, without the thread count having to be reduced, which is an interesting result.

As explained above, the `CS.reorder_X_bad` benchmark (where X is the number of threads launched) are versions of the adversarial delay bounding example given in Figure 2 in §2.4. One priority change point at the right place (and a particular permutation of initial thread priorities) is sufficient for PCT to expose this bug. Thus, PCT manages to find the bug even when the number of threads is doubled (compare `CS.reorder_10_bad` and `CS.reorder_20_bad` in Table IV). This is in contrast to systematic techniques, where increasing the thread count increases the number of schedules explored before the first bug, until the point where the bug is not found within the schedule limit (see Table III).

Recall that, for each value of d that we used with PCT and for each benchmark, we estimated the worst case (smallest) number of buggy schedules that we should find given a bug of depth d, parameters n and k from the benchmark, and our schedule limit of 100,000 (see §2.7 and §6.3). The estimate for each d is only relevant if the bug associated with a benchmark can in fact manifest with depth d. These estimates can be see in Table IV. The minimum value of d for which PCT found a bug provides an *upper bound* on the bug depth; the actual bug depth may be smaller. Assuming that the minimum value of d for which PCT found a bug is, in fact, the depth of the bug, it can be seen that PCT always found many more schedules than the estimated number for that bug. For example, consider `chess.IWSQ` in Table IV. It is likely that this bug has depth d=2, since PCT d=1 was not able to find the bug. Assuming this, the estimated worst case number of buggy schedules that we should find (in the PCT d=2 column) is less than 1, yet the actual number of buggy schedules found was 4,829. In fact, for d=2 and d=3 the majority of the benchmarks had a worst case estimate of less than 1 schedule, suggesting that the bugs should not be found within our schedule limit (yet, most bugs were found). Our results agree with the original evaluation of PCT [Burckhardt et al. 2010], which showed that the number of buggy schedules found in practice is usually much greater than the smallest number of buggy schedules predicted by the formula.

Comparison with the default Maple algorithm. As shown in Figure 3d, MapleAlg missed 20 bugs overall, 19 of which were found by other techniques. This includes benchmarks like `CS.bluetooth_driver_bad` and `CS.circular_buffer_bad`, which were quickly found by most other techniques. Maple livelocked on the CHESS benchmarks; this is presumably a bug in the tool that could be fixed. MapleAlg attempts to force

certain patterns of inter-thread accesses (or *interleaving idioms*) that might lead to concurrency bugs. It is possible that some of the bugs it misses require interleaving idioms that are not included in MapleAlg.

Small schedule bounds. To answer **RQ5**, we note that schedule bounding exposed 45 of the 49 bugs, and 44 of these require a preemption bound of two or less (note that, if a bug can be found with a delay bound of c, then it can also be found with a preemption bound of c, although not necessarily within the schedule limit when using IPB). Furthermore, 42 of these were found using a delay bound of two or less. Thus, a large majority of the bugs in SCTBench can be found with a small preemption or delay bound. This supports previous claims that many bugs can be exposed using a small number of preemptions or delays [Musuvathi and Qadeer 2007a; Musuvathi et al. 2008; Emmi et al. 2011].

The systematic techniques missed the bugs in `CS.reorder_10_bad`, `CS.reorder_20_bad` and `CS.twostage_100_bad`, which, as explained above, are duplicates of other benchmarks but with higher thread counts. The `CS.reorder_X_bad` benchmark is the adversarial delay bounding example given in Figure 2 in §2.4. Thus, these benchmarks require a delay bound of one less than X (where X is the number of threads). However, it is not clear whether such a scenario is likely to occur in real multithreaded programs.

The bug in `radbench.bug2` requires three preemptions or delays to occur (see Table III). The benchmark is a test case for the SpiderMonkey JavaScript engine in Firefox. A bug requiring three preemptions and delays has been reported before in [Emmi et al. 2011] and this was the first time CHESS had found such a bug. Note that we reduced the number of threads in radbench.bug2 from six to two; thus, IPB and IDB explore exactly the same schedules. Nevertheless, two threads is enough to expose the bug.

The bug in `misc.safestack` was missed by all techniques and reportedly requires five preemptions and three threads. Given this information, we tried running PCT with $d = 6$ for 100,000 executions, but the bug did not occur. We reproduced the bug using Relacy[7], a weak memory data race detector that performs either systematic or controlled random scheduling for C++ programs that use C++ atomics. The bug was found using the random scheduling mode after 75,058 schedules. It is unclear why Maple's random scheduler did not find the bug. It is possible that the number of scheduling points with Maple is higher, as Relacy only inserts scheduling points before atomic operations.

SPLASH-2 benchmarks. As explained in §4.1, we reduced the input values in the SPLASH-2 benchmarks; this resulted in fewer scheduling points and allowed our data race detector to complete, without exhausting memory. Due to these changes, the results are not directly comparable with other experiments that use the SPLASH-2 benchmarks (unless parameters are similarly reduced). However, the bugs are found by all systematic techniques after just two schedules; this would be the same, regardless of parameter values. Therefore, the # *schedules to first bug* data for the systematic techniques are comparable to other techniques.

7. RELATED WORK

Background on controlled scheduling was discussed in §2. We now discuss similar prior work and other relevant concurrency testing techniques.

A prior study created a benchmark suite of concurrent programs to evaluate the bug detection capabilities of several tools and techniques [Rungta and Mercer 2009]. Our

[7] http://www.1024cores.net/home/relacy-race-detector

id	name	# max threads (n)	# max enabled threads	# max steps (k)	IPB bound	IPB # schedules to first bug	IPB # schedules	IPB # new schedules	IPB # buggy schedules	IDB bound	IDB # schedules to first bug	IDB # schedules	IDB # new schedules	IDB # buggy schedules	DFS # schedules to first bug	DFS # schedules	DFS # buggy schedules	DFS % buggy
0	CB.aget-bug2	4	3	24	0	1	10	10	4	0	1	1	1	1	1	46486	29513	63%
1	CB.pbzip2-0.9.4	4	4	54	0	2	12	12	4	1	2	31	30	13	2	L	68226	*68%
2	CB.stringbuffer-jdk1.4	2	2	10	2	9	13	8	1	2	9	13	8	1	7	24	1	4%
3	CS.account_bad	4	3	8	0	3	6	6	2	1	3	5	4	1	3	28	4	14%
4	CS.arithmetic_prog_bad	3	2	20	0	1	4	4	4	0	1	1	1	1	1	19680	19680	100%
5	CS.bluetooth_driver_bad	2	2	13	1	6	7	6	1	1	6	7	6	1	36	177	10	5%
6	CS.carter01_bad	5	3	19	1	9	19	16	2	1	8	12	11	1	8	1708	49	2%
7	CS.circular_buffer_bad	3	2	31	1	23	35	32	12	2	25	79	56	36	20	3991	2043	51%
8	CS.deadlock01_bad	3	2	11	1	9	12	9	2	1	7	9	8	1	10	46	3	6%
9	CS.din_phil2_sat	3	2	21	0	1	3	3	3	0	1	1	1	1	1	5336	4686	87%
10	CS.din_phil3_sat	4	3	32	0	1	13	13	13	0	1	1	1	1	1	L	85542	*85%
11	CS.din_phil4_sat	5	4	43	0	1	73	73	73	0	1	1	1	1	1	L	86231	*86%
12	CS.din_phil5_sat	6	5	39	0	1	501	501	501	0	1	1	1	1	1	L	L	*100%
13	CS.din_phil6_sat	7	6	49	0	1	4051	4051	4051	0	1	1	1	1	1	L	L	*100%
14	CS.din_phil7_sat	8	7	59	0	1	7	7	7	0	1	1	1	1	1	924	924	100%
15	CS.fsbench_bad	28	27	155	0	1	1	1	1	0	1	1	1	1	1	L	L	*100%
16	CS.lazy01_bad	4	3	11	0	1	13	13	6	0	1	1	1	1	1	118	81	68%
17	CS.phase01_bad	3	2	11	0	1	2	2	2	0	1	1	1	1	1	17	17	100%
18	CS.queue_bad	3	2	83	1	98	100	97	2	2	63	482	420	326	43	L	59036	*59%
19	CS.reorder_10_bad	11	10	40	0	✗	L	L	0	5	✗	L	38129	0	✗	L	0	*0%
20	CS.reorder_20_bad	21	20	89	0	✗	L	L	0	4	✗	L	21023	0	✗	L	0	*0%
21	CS.reorder_3_bad	4	3	12	1	43	74	61	2	2	25	45	35	3	126	2494	23	<1%
22	CS.reorder_4_bad	5	4	16	1	359	774	701	3	3	205	417	330	7	6409	L	86	*<1%
23	CS.reorder_5_bad	6	5	20	1	3378	8483	7982	4	4	1513	3681	2843	15	✗	L	0	*0%
24	CS.stack_bad	3	2	43	1	23	50	47	9	1	22	32	31	9	22	L	6361	*6%
25	CS.sync01_bad	3	2	9	0	1	2	2	2	0	1	1	1	1	1	6	6	100%
26	CS.sync02_bad	3	2	18	0	1	2	2	2	0	1	1	1	1	1	88	88	100%
27	CS.token_ring_bad	5	4	11	0	8	24	24	4	2	10	29	22	3	8	280	57	20%
28	CS.twostage_100_bad	101	100	792	0	✗	L	L	0	2	✗	L	99304	0	✗	L	0	*0%
29	CS.twostage_bad	3	2	11	1	9	10	7	1	1	7	9	8	1	13	87	3	3%
30	CS.wronglock_3_bad	5	4	25	1	243	856	783	66	1	15	22	21	2	3233	L	3006	*3%
31	CS.wronglock_bad	9	8	49	0	✗	L	L	0	1	31	42	41	2	✗	L	0	*0%
32	chess.IWSQ	3	3	169	1	✗	L	99997	0	2	2990	4378	4264	192	✗	L	0	*0%
33	chess.IWSQWS	3	1	660	1	✗	10000	9997	0	1	219	471	470	1	✗	10000	0	*0%
34	chess.SWSQ	3	1	2406	1	✗	10000	9997	0	1	773	1698	1697	1	✗	10000	0	*0%
35	chess.WSQ	3	3	161	2	2814	8852	8626	640	2	801	2048	1974	192	✗	L	0	*0%
36	inspect.qsort_mt	3	3	81	1	31	88	84	2	1	19	28	27	1	75861	L	2127	*2%
37	misc.ctrace-test	3	2	22	1	4	20	19	12	1	4	20	19	12	4	20	12	60%
38	misc.safestack	4	3	117	1	✗	L	99987	0	3	✗	L	95958	0	✗	L	0	*0%
39	parsec.ferret	11	11	24453	0	✗	L	L	0	1	51	4575	4574	1	✗	L	0	*0%
40	parsec.streamcluster	5	2	1373	1	7951	16072	16066	19	1	1336	1372	1371	10	✗	L	0	*0%
41	parsec.streamcluster2	7	3	4177	0	✗	L	L	0	1	4153	4175	4174	20	✗	L	0	*0%
42	parsec.streamcluster3	5	2	1373	0	2	6	6	4	1	2	1359	1358	4	2	L	60785	*60%
43	radbench.bug1	4	3	21889	1	✗	L	99962	0	1	616	14206	14205	1	✗	L	0	*0%
44	radbench.bug2	2	2	171	3	59354	72704	69895	48	3	59354	72704	69895	48	✗	L	0	*0%
45	radbench.bug6	3	3	101	1	84	168	165	3	1	60	86	85	3	✗	L	0	*0%
46	splash2.barnes	2	2	4449	1	2	4378	4377	326	1	2	4378	4377	326	2	L	23504	*23%
47	splash2.fft	2	2	152	1	2	134	133	61	1	2	134	133	61	2	L	75434	*75%
48	splash2.lu	2	2	140	1	2	105	104	49	1	2	105	104	49	2	L	49887	*49%

Table III: Experimental results for systematic concurrency testing using iterative preemption bounding (IPB), iterative delay bounding (IDB) and unbounded depth-first search (DFS). Entries marked 'L' indicate 100,000, our schedule limit. A '✗' indicates that no bug was found. A percentage prefixed with '*' does not apply to *all* schedules, only those that were explored via DFS before the schedule limit was reached.

id	name	# max threads (n)	# max enabled threads	# max steps (k)	Rand		PCT d=1			PCT d=2			PCT d=3			MapleAlg		
					# schedules to first bug	# buggy schedules	# schedules to first bug	# buggy schedules	est. worst case # buggy	# schedules to first bug	# buggy schedules	est. worst case # buggy	# schedules to first bug	# buggy schedules	est. worst case # buggy	found?	# schedules	total time (seconds)
0	CB.aget-bug2	4	3	24	4	48591	7	25053	43	7	40313	1	5	46938	<1	✓	17	37
1	CB.pbzip2-0.9.4	4	4	54	1	41771	1	16466	8	1	22971	<1	5	27385	<1	✓	4	20
2	CB.stringbuffer-jdk1.4	2	2	10	23	6308	✗	0	500	✗	0	50	1	1979	5	✓	9	7
3	CS.account_bad	4	3	8	8	11912	5	25060	390	5	21936	48	5	19527	6	✓	20	12
4	CS.arithmetic_prog_bad	3	2	20	1	L	1	L	83	1	L	4	1	L	<1	✓	1	1
5	CS.bluetooth_driver_bad	2	2	13	8	6436	✗	0	295	11	3871	22	11	5968	1	✗	11	7
6	CS.carter01_bad	5	3	19	1	46877	✗	0	55	9	16028	2	3	29243	<1	✓	6	5
7	CS.circular_buffer_bad	3	2	31	1	91146	✗	0	34	1	12818	1	1	28763	<1	✗	17	12
8	CS.deadlock01_bad	3	2	11	1	37405	✗	0	275	15	9020	25	15	17234	2	✗	7	5
9	CS.din_phil2_sat	3	2	21	1	96860	1	L	75	2	95337	3	2	93558	<1	✓	1	1
10	CS.din_phil3_sat	4	3	32	1	92850	1	L	24	1	93792	<1	1	90207	<1	✓	1	1
11	CS.din_phil4_sat	5	4	43	1	88754	1	L	10	1	93040	<1	1	88414	<1	✓	1	1
12	CS.din_phil5_sat	6	5	39	1	L	1	L	10	1	L	<1	1	L	<1	✓	1	1
13	CS.din_phil6_sat	7	6	49	1	L	1	L	5	1	L	<1	1	L	<1	✓	1	1
14	CS.din_phil7_sat	8	7	59	1	L	1	L	3	1	L	<1	1	L	<1	✓	1	1
15	CS.fsbench_bad	28	27	155	1	L	1	L	<1	1	L	<1	1	L	<1	✓	1	1
16	CS.lazy01_bad	4	3	11	2	60626	1	49847	206	1	53343	18	1	56197	1	✓	1	1
17	CS.phase01_bad	3	2	11	1	L	1	L	275	1	L	25	1	L	2	✓	1	1
18	CS.queue_bad	3	2	83	1	99986	✗	0	4	38	818	<1	6	14046	<1	✓	2	1
19	CS.reorder_10_bad	11	10	40	✗	0	✗	0	5	439	89	<1	439	135	<1	✗	11	7
20	CS.reorder_20_bad	21	20	89	✗	0	✗	0	<1	219	131	<1	219	224	<1	✗	11	7
21	CS.reorder_3_bad	4	3	12	39	2498	✗	0	173	168	2653	14	115	4559	1	✗	10	7
22	CS.reorder_4_bad	5	4	16	68	726	✗	0	78	86	1257	4	7	2053	<1	✗	11	8
23	CS.reorder_5_bad	6	5	20	68	202	✗	0	41	7	688	2	7	1057	<1	✗	11	7
24	CS.stack_bad	3	2	43	2	60949	✗	0	18	2	27680	<1	2	40159	<1	✗	10	8
25	CS.sync01_bad	3	2	9	1	L	1	L	411	1	L	45	1	L	5	✓	1	1
26	CS.sync02_bad	3	2	18	1	L	1	L	102	1	L	5	1	L	<1	✓	1	1
27	CS.token_ring_bad	5	4	11	9	13004	44	8238	165	11	13655	15	11	16908	1	✓	5	4
28	CS.twostage_100_bad	101	100	792	✗	0	✗	0	<1	10548	5	<1	10548	6	<1	✗	11	9
29	CS.twostage_bad	3	2	11	15	7848	✗	0	275	15	12097	25	15	19840	2	✓	8	5
30	CS.wronglock_3_bad	5	4	25	1	31302	✗	0	32	13	6442	1	4	11107	<1	✓	6	4
31	CS.wronglock_bad	9	8	49	1	32534	✗	0	4	29	3636	<1	24	6693	<1	✓	6	4
32	chess.IWSQ	3	3	169	19	133	✗	0	1	61	4829	<1	24	8069	<1	✗	7	-
33	chess.IWSQWS	3	1	660	3	1538	✗	0	<1	584	8	<1	616	19	<1	✗	9	-
34	chess.SWSQ	3	1	2406	15	88	✗	0	<1	1109	2	<1	612	11	<1	✗	7	-
35	chess.WSQ	3	3	161	392	106	✗	0	1	61	4993	<1	24	8357	<1	✗	12	12
36	inspect.qsort_mt	3	3	81	72	1024	✗	0	5	109	1271	<1	109	2346	<1	✗	142	102
37	misc.ctrace-test	3	2	22	1	24487	2193	7	68	5	27307	3	5	33607	<1	✓	1	1
38	misc.safestack	4	3	117	✗	0	✗	0	1	✗	0	<1	✗	0	<1	✗	23	16
39	parsec.ferret	11	11	24453	✗	0	3	39389	<1	3	63027	<1	3	69745	<1	✓	27	205
40	parsec.streamcluster	5	2	1373	1	68746	1	49831	<1	1	50194	<1	1	50428	<1	✓	1	2
41	parsec.streamcluster2	7	3	4177	21	12514	2	50135	<1	2	50096	<1	2	50075	<1	✗	24	149
42	parsec.streamcluster3	5	2	1373	2	34448	1	50081	<1	1	50081	<1	1	50081	<1	✓	1	1
43	radbench.bug1	4	3	21889	✗	0	✗	0	<1	3084	8	<1	79190	1	<1	✗	583	13811
44	radbench.bug2	2	2	171	27071	9	✗	0	1	✗	0	<1	1813	54	<1	✗	239	950
45	radbench.bug6	3	3	101	1	30211	✗	0	3	15	4543	<1	15	7675	<1	✗	11	10
46	splash2.barnes	2	2	4449	2	49933	2	49967	<1	2	49967	<1	2	49967	<1	✓	1	1
47	splash2.fft	2	2	152	2	62188	2	49967	2	2	50007	<1	2	50017	<1	✓	2	2
48	splash2.lu	2	2	140	1	97329	2	49967	2	2	53574	<1	2	56605	<1	✓	2	3

Table IV: Experimental results for non-systematic testing with a controlled random scheduler (Rand), PCT for each $d \in \{1, 2, 3\}$, and using the Maple algorithm (MapleAlg). Entries marked 'L' indicate 100,000, our schedule limit. A '✗' indicates that no bug was found. In the MapleAlg results, '-' indicates that the Maple tool timed out after 24 hours.

49 test programs are drawn from 35 distinct bugs in pthread benchmarks written in C/C++, while the prior study uses 12 distinct bugs in benchmarks written in both Java and C#.[8] Thus, our study is over a larger set of benchmarks, which are mostly distinct from the set used in the prior study. Furthermore, 8 of our benchmarks are derived from open source desktop libraries and applications and a further 7 are from parallel performance benchmark suites (the PARSEC and SPLASH2 benchmarks). The C# benchmarks from the prior study are standalone synthetic test cases. Our study is focused on comparing five controlled testing techniques (or seven controlled testing techniques if the different parameter values for PCT are treated as distinct techniques), implemented in the same controlled testing framework within the same tool, plus the Maple algorithm. This allows us to compare the *techniques* fairly in a single tool (as opposed to comparing several distinct tools that may implement the techniques in different manners), because each technique operates on the same low level implementation, e.g. they use the same notion of scheduling points. In contrast, the prior study tests six techniques implemented over four tools. The prior study claims to test schedule bounding on 12 benchmarks by running CHESS with specific preemption bounds for each benchmark, although the study only includes results tables for 3 of the benchmarks. We test schedule bounding more thoroughly; we use iterative preemption bounding and iterative delay bounding to explore schedules with various preemption and delay bounds, and report results for both techniques on all benchmarks.

Partial-order reduction (POR) [Godefroid 1996] reduces the number of schedules that need to be explored soundly (i.e. without missing bugs, assuming the search completes). It relies on the fact that schedules can be represented as a partial-order of operations, where each partial-order reaches the same state. Such techniques attempt to explore only *one* schedule from each partial-order. Dynamic POR [Flanagan and Godefroid 2005] computes persistent sets [Godefroid 1996] during systematic search; as *dependencies* between operations are detected, additional schedules are considered. Happens-before graph caching [Musuvathi et al. 2008; Musuvathi and Qadeer 2007b] is similar to state-hashing [Holzmann 1987], except the partial-order of synchronization operations is used as an approximation of the state, resulting in a reduction similar to sleep-sets [Godefroid 1996]. The combination of dynamic POR and schedule bounding is complex and the topic of recent research [Coons et al. 2013; Musuvathi and Qadeer 2007b; Holzmann and Florian 2011]. Relaxations of the happens-before relation have also been used to further reduce the number of schedules that need to be considered during systematic concurrency testing [Thomson and Donaldson 2015; Huang 2015].

The parallel PCT algorithm [Nagarakatte et al. 2012] improves the PCT algorithm by allowing parallel execution of many threads, as opposed to always serializing execution. This provides increased execution speed but maintains the probabilistic guarantee from PCT. We focus on controlled testing techniques where the program is serialized; since we report the number of terminal schedules, increased execution speed does not affect our results.

In addition to the Maple algorithm, there has been a wide-range of work on other non-controlled approaches, including [Edelstein et al. 2002; Sen 2008; Park et al. 2009]. Like parallel PCT, these approaches are appealing as they allow parallel execution of many threads and can handle complex synchronization and nondeterminism.

Randomization has been shown to be effective for search diversification in stateful model checking, where it can be used to allow independent searches to occur in parallel

[8]The companion website for the prior study shows 17 benchmarks that were translated to C#, although only 12 were used in the published study; translation was necessary so that the benchmarks could be used with CHESS.

for improved coverage on multicore systems within a predefined time limit [Holzmann et al. 2011]. In our study, we use a schedule limit instead of a time limit; it is worth noting that PCT and controlled random scheduling are both trivially parallelizable, and that systematic techniques can also be parallelized with additional effort [Simsa et al. 2013].

One conclusion from our study is that random scheduling may be a useful technique and should be tried more often; random scheduling has been used in recent work (along with systematic concurrency testing) to find bugs in asynchronous programs [Deligiannis et al. 2015].

We do not consider relaxed memory models in this study; as in prior work [Musuvathi et al. 2008; Yu et al. 2012], we assume sequential consistency. Finding weak memory bugs would at least require instrumenting memory accesses (similar to performing data race detection during controlled scheduling), which would have been far too slow using Maple's built-in support for this. Recent work has shown an efficient approach for testing relaxed memory models with SCT using dynamic partial-order reduction [Zhang et al. 2015].

Our study has briefly touched on dynamic data race detection issues. A discussion of this wide area is out of scope here, but we refer to [Flanagan and Freund 2009] for the state-of-the-art.

8. FUTURE WORK

We have presented an independent empirical study on concurrency testing using controlled schedulers. In future work we believe it would be fruitful to expand SCTBench through the addition of further non-trivial benchmarks, to enable larger studies to be conducted. In future reproduction studies, we recommend evaluating the various partial-order reduction techniques that have been proposed in recent years to soundly reduce the size of the schedule-space.

Acknowledgements. We thank the anonymous reviewers, especially reviewer #1 who rightfully suggested that we should show results for various schedule limits. We thank Ethel Bardsley, Nathan Chong, Pantazis Deligiannis, Tony Field, Jeroen Ketema and Shaz Qadeer, for their thorough comments and suggestions on an earlier draft of this work. We are also grateful to reviewers of the PPoPP'14 conference version of this work [Thomson et al. 2014] for their useful comments, and especially to reviewer #1 of the PPoPP'14 submission who suggested that we try controlled random scheduling, which led to interesting results. We relied on the Imperial College High Performance Computing Service[9] for this work: the HPC cluster was critical to performing a study on this scale.

REFERENCES

Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2014. Context-Bounded Analysis of TSO Systems. In *ETAPS*. 21–38.

Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. 2006. Producing Scheduling That Causes Concurrent Programs to Fail. In *PADTAD*. 37–40.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (2010), 66–75.

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[9] http://www.imperial.ac.uk/ict/services/teachingandresearchservices/highperformancecomputing

Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *HotPar*. 1–6.
Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*. 167–178.
Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. 2013. Bounded Partial-order Reduction. In *OOPSLA*. 833–848.
Lucas Cordeiro and Bernd Fischer. 2011. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In *ICSE*. 331–340.
Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In *PLDI*. 154–164.
Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. 2002. Multithreaded Java Program Test Generation. *IBM Syst. J.* 41, 1 (2002), 111–125.
Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded Scheduling. In *POPL*. 411–422.
Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*. 121–133.
Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*. 110–121.
Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.
Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer.
Patrice Godefroid. 1997. Model Checking for Programming Languages using VeriSoft. In *POPL*. 174–186.
G.J. Holzmann, R. Joshi, and A. Groce. 2011. Swarm Verification Techniques. *Software Engineering, IEEE Transactions on* 37, 6 (2011), 845–857.
Gerard J. Holzmann. 1987. On Limits and Possibilities of Automated Protocol Analysis. In *PSTV*. 339–344.
Gerard J. Holzmann and Mihai Florian. 2011. Model Checking with Bounded Context Switching. *Formal Asp. Comput.* 23, 3 (2011), 365–389.
Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI*. 165–174.
Jeff Huang and Charles Zhang. 2011. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. In *SAS*. 163–179.
Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. 2011. RADBench: A Concurrency Bug Benchmark Suite. In *HotPar*. 1–6.
Nicholas Jalbert and Koushik Sen. 2010. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. In *FSE (FSE '10)*. 57–66.
Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*. 185–198.
Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. 2010. A Platform for Search-based Testing of Concurrent Software. In *PADTAD*. 48–58.
Akash Lal and Thomas W. Reps. 2009. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *Formal Methods in System Design* 35, 1 (2009), 73–97.
Bil Lewis and Daniel J. Berg. 1998. *Multithreaded Programming with Pthreads*. Prentice-Hall.
Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*. 329–339.

Chi-Keung Luk and others. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*. 190–200.
Madanlal Musuvathi and Shaz Qadeer. 2007a. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*. 446–455.
M. Musuvathi and S. Qadeer. 2007b. *Partial-order Reduction for Context-bounded State Exploration*. Technical Report MSR-TR-2007-12. Microsoft Research.
Madanlal Musuvathi and Shaz Qadeer. 2008. Fair Stateless Model Checking. In *PLDI*. 362–371.
Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*. 267–280.
Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. 2012. Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection. In *PLDI*. 543–554.
Satish Narayanasamy and others. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*. 22–31.
S. Park, S. Lu, and Y. Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*. 25–36.
Neha Rungta and Eric G. Mercer. 2009. Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs. In *PADTAD*. ACM, 9:1–9:10.
Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *PLDI*. 11–21.
Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. 2013. Scalable Dynamic Partial Order Reduction. In *Runtime Verification*, Shaz Qadeer and Serdar Tasiran (Eds.). Lecture Notes in Computer Science, Vol. 7687. Springer Berlin Heidelberg, 19–34.
Herb Sutter and James Larus. 2005. Software and the Concurrency Revolution. *ACM Queue* 3, 7 (2005), 54–62.
Paul Thomson and Alastair F. Donaldson. 2015. The lazy happens-before relation: better partial-order reduction for systematic concurrency testing. In *PPoPP*. 259–260.
Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency Testing Using Schedule Bounding: An Empirical Study. In *PPoPP*. 15–28.
Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage Guided Systematic Concurrency Testing. In *ICSE*. 221–230.
Steven Cameron Woo and others. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*. 24–36.
Y. Yang, X. Chen, and G. Gopalakrishnan. 2008. *Inspect: A Runtime Model Checker for Multithreaded C Programs*. Technical Report UUCS-08-004. University of Utah.
Jie Yu and Satish Narayanasamy. 2009. A Case For an Interleaving Constrained Shared-memory Multi-processor. In *ISCA*. 325–336.
Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *OOPSLA*. 485–502.
Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *PLDI*.

Received February 2007; revised March 2009; accepted June 2009