In proceedings of PPoPP'14

Concurrency Testing Using Schedule Bounding: an Empirical Study *

Paul Thomson, Alastair F. Donaldson, Adam Betts
Imperial College London
{paul.thomson11,afd,abetts}@imperial.ac.uk

Abstract

We present the first independent empirical study on schedule bounding techniques for systematic concurrency testing (SCT). We have gathered 52 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 to attempt to answer several research questions, including: How effective are the two main schedule bounding techniques, preemption bounding and delay bounding, at bug finding? What challenges are associated with applying SCT to existing code? How effective is schedule bounding compared to a naive random scheduler at finding bugs? Our findings confirm that delay bounding is superior to preemption bounding and that schedule bounding is more effective at finding bugs than unbounded depth-first search. The majority of bugs in SCTBench can be exposed using a small bound (1-3), supporting previous claims, but there is at least one benchmark that requires 5 preemptions. Surprisingly, we found that a naive *random* scheduler is at least as effective as schedule bounding for finding bugs. 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

Keywords Concurrency; systematic concurrency testing; stateless model checking; context bounding

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

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

In recent years, researchers have shown great interest in systematic techniques for testing concurrent programs [7, 12, 26, 32, 34, 36] to expose 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 [31], the ineffectiveness of traditional testing for detecting and reproducing concurrency bugs due to nondeterminism [19], and the desire for automatic, precise analysis, which is hard to achieve using static techniques [1].

Systematic concurrency testing (SCT) [7, 12, 26, 32, 34], also known as *stateless model checking* [12], is used to find and reproduce bugs in multi-threaded software. It has been implemented in a variety of tools, including CHESS [26] and Verisoft [12]. The technique involves repeatedly executing a multi-threaded program, controlling the scheduler so 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, has no false-positives and bugs can be reproduced by forcing the bug-inducing schedule.

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* [23] bounds the number of preemptive context switches that are allowed in a schedule. *Delay bounding* [7] 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. 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 easy to understand. This property has been used in trace simplification [15, 16]. Secondly, it gives bounded coverage guarantees; if the search manages to explore 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 recent works on concurrent software verification employ schedule bounding to improve tractability [6, 20].

The hypothesis that preemption and delay bounding are 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 [7, 23, 26]. Prior work has also shown that delay bounding improves on preemption bounding, allowing additional bugs to be detected [7]. However, these works have 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, these works do not explicitly show that schedule bounding provides benefit over a naive random scheduler for finding bugs.[1]

We believe that these exciting and important claims about the effectiveness of schedule bounding would benefit from further scrutiny using a wider range of publicly available applications. To this end, we present the first independent, fully reproducible empirical study of schedule bounding techniques for SCT. We have put together SCTBench, a set of 52 publicly available benchmarks amenable to systematic concurrency testing, gathered from a combination of stand-alone multi-threaded 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, with a few additions. Our study is based on an extended version of Maple [36], an open source concurrency testing tool. Our aim was to answer the following questions over a large and varied set of benchmarks:

- Can we find the known bugs in the publicly available benchmark suites using SCT?
- How do preemption and delay bounding compare in their effectiveness at finding concurrency bugs?
- How effective is schedule bounding compared to a naive random scheduler at finding bugs?
- How easy is it to apply SCT to various existing code bases in practice?
- Can we find examples of concurrency bugs that require more than three preemptions (the largest number of preemptions required to expose a bug in previous work [7])?

[1] We note that [23] 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.

1.1 Main findings and contribution

We now summarise the main findings of our study. The conclusions we draw of course only relate to the 52 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 2, which are 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) and naive random scheduling (Rand). For each method evaluated, a limit of 10,000 schedules per benchmark is used.

Schedule bounding is similar to naive random scheduling in terms of bug-finding ability. Our assumption prior to this study was that a naive random scheduler would not be effective at finding bugs. This claim is not made explicitly in prior work, but neither is it addressed; prior work (such as [7, 23, 26]) only includes depth-first search or preemption bounding as a baseline for finding bugs.[1] Our findings, summarised in Figure 2b, contradict this assumption: the bugs in 44 benchmarks were found by *both* schedule bounding and a naive random scheduler within 10,000 executions. Schedule bounding and random scheduling each found one additional, distinct, bug. The random scheduler almost always led to faster bug detection than with schedule bounding. This raises two important questions: Does schedule bounding actually aid in bug finding, compared to more naive 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". As noted above, schedule bounding provides several benefits regardless of bug finding ability which are not questioned by our findings.

Many bugs can be found via a small (1-3) schedule bound. Schedule bounding exposed each bug in 45 of the 52 benchmarks and the highest preemption bound required in these cases was three. Thus, a large majority of the bugs in SCTBench can be found with a small schedule bound. This supports previous claims [7, 23, 26]. It also adds weight to the argument that bounded guarantees provided by schedule bounding are useful. However, we note that one benchmark is reported to require a minimum of five preemptions for the bug to manifest. A straightforward depth-first search with no schedule bounding exposed bugs in 33 benchmarks, all of which were also found with schedule bounding.

Delay bounding beats preemption bounding. Delay bounding found all of the 38 bugs that were found by preemption bounding, plus seven that were not (see Figure 2a).

SCT can be difficult to apply. Many interesting benchmarks could not be included in our study, as they use nondeterministic features or additional synchronisation that is not modelled or controlled appropriately by most SCT tools. This in-

cludes network communication, multiple processes, signals (other than pthread condition variables) and event libraries.

Additionally, we found several program modules that could not easily be tested in isolation due to direct dependencies on system functions and other program modules. Thus, creating isolated tests suitable for SCT may require significant effort, especially for those who are not developers of the software under test.

Data races are common. Many benchmarks feature a large number of data races that are not regarded as bugs. Treating them as errors would be too easy for benchmarking purposes, as they are very common. For the study, we explore the interleavings arising from sequentially consistent outcomes of racy memory accesses in order to expose bugs such as assertion failures and incorrect output.

Bugs may not be detected without additional checks. Some concurrency bugs manifest as out-of-bound 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.

Trivial benchmarks. We argue that certain benchmarks used in prior work are "trivial" (based on certain properties – see Table 2) 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 19 benchmarks were exposed 50% of the time when using a random scheduler, with 10,000 runs. In nine of these cases, the bugs were exposed 100% of the time.

Non-trivial benchmarks. We believe most benchmarks from the CHESS, PARSEC and RADBench suites, as well as the `misc.safestack` benchmark, 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 schedule bounding and other techniques.

1.2 SCTBench and reproducibility of our study

To make our study fully reproducible, we provide the 52 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 terminal schedules, not time, which allows them to be easily compared with other work and tools.

2. Systematic Concurrency Testing

Systematic concurrency testing (SCT) works by repeatedly executing a concurrent program using a custom scheduler, forcing a different thread schedule to be explored on each execution. Execution is serialised, so that concurrency is emulated by interleaving instructions from different threads. 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. Nondeterminism such as user input, network communication, etc. must be fixed or modelled. This continues until all schedules have been explored, or until a time or schedule limit is reached. The search space is over schedules; unlike model checking, program states are not represented. This is appealing because the state of real software is large and difficult to capture.

A schedule $\alpha = \langle \alpha(1), \ldots, \alpha(n) \rangle$ is a list of thread identifiers. We use the following shorthands for lists: $\mathit{length}(\alpha) = n$; $\alpha \cdot t = \langle \alpha(1), \ldots, \alpha(n), t \rangle$; $\mathit{last}(\alpha) = \alpha(n)$. The element $\alpha(i)$ refers to the thread that is executing at step i in the execution of the multi-threaded 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 in the context of $T0$, one step in $T1$ and then a step in $T0$. A step corresponds to a particular thread executing a *visible* operation [12], such as a synchronisation 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 [12]. 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 is referred to as a *terminal schedule*.

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 $\alpha(i) \neq \alpha(i-1)$. 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*.

Preemption bounding Preemption bounding [23] bounds the number of preemptions in a schedule. Let the preemption count *PC* of a schedule be defined 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 k, any schedule α with $\mathit{PC}(\alpha) > k$ will not be explored.

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

Figure 1: Simple multi-threaded program.

Example 1. *Consider Figure 1, which shows a simple multi-threaded 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 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.*

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 [7] 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 the deterministic scheduler that is non-preemptive and when blocked chooses the next enabled thread in thread creation order in a round-robin fashion. We assume this instantiation of delay bounding because it has been used in previous work [7] 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 $\mathit{distance}(x, y)$ 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 $\{0, 1, 2, 3\}$, $\mathit{distance}(1, 0)$ is 3. For a schedule α and a thread id t, let $\mathit{delays}(\alpha, t)$ yield the number of delays required to schedule thread t at the state reached by α:

$$\begin{aligned}\mathit{delays}(\alpha, t) = \quad & |\{x : 0 \leq x < \mathit{distance}(\mathit{last}(\alpha), t)) \\ & \wedge (\mathit{last}(\alpha) + x) \bmod N \in \mathit{enabled}(\alpha)\}|\end{aligned}$$

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

Define the delay count *DC* of a schedule recursively; a schedule of length zero or one has no delays, otherwise:

$$\mathit{DC}(\alpha \cdot t) = \mathit{DC}(\alpha) + \mathit{delays}(\alpha, t)$$

With a delay bound of k, any schedule α with $\mathit{DC}(\alpha) > k$ 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.

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 assume that T2 comprises the same statements as T1, which we label as: f) x=1; g) y=1. 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, note that 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 [7] suggests that adversarial examples like this are not common in practice. Our results (§6) also support this.*

Theoretical Complexity Upper-bounds for the number of terminal schedules produced by SCT techniques are described in [7, 23]. 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 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, it performs well (in terms of number of schedules) even when programs create a large number of threads.

Finding bugs The intuition behind schedule bounding is that it greatly reduces the number of schedules, but still allows many bugs to be found [7, 23, 26]. The reasoning is that only a *few* preemptions are needed at the *right* places in order to enforce an ordering that causes the bug to manifest. Performing a preemption elsewhere will have little impact. A complete depth-first search becomes infeasible as the execution length increases due to the large number of context switches, many of which are likely to be irrelevant.

Iterative schedule bounding Schedule bounding can be performed iteratively [23], where all schedules with zero preemptions or delays are all executed, followed by those with one preemption or delay, etc. until there are no more schedules or 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. The partial order for iterative delay bounding with respect to *DC* is analogous. Thus, iterative schedule bounding is a heuristic that aims to expose buggy schedules before the time or schedule limit is reached, based on the hypothesis discussed above.

In this study, we perform iterative schedule bounding to compare preemption and delay bounding.

3. Modifications to Maple

We chose to use a modified version of the Maple tool [36] to conduct our experimental study. Maple is a concurrency testing tool framework for pthread [21] programs. It uses the dynamic instrumentation library, PIN [22], to test binaries without the need for recompilation. One of the modules, *systematic*, is a re-implementation of the CHESS [26] algorithm for preemption bounding. The main reason for using Maple, instead of CHESS, is that it targets pthread programs. This allows us to test a wide variety of open source multi-threaded benchmarks and programs. Previous evaluations [7, 23, 26] 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, although 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.[2]

Delay bounding We modified Maple to add support for delay bounding, following a similar design to the existing support for preemption bounding. At each scheduling point, Maple conceptually constructs several schedules consisting of the current schedule concatenated with an enabled thread t. These are added to a set and will be explored on subsequent executions. If switching to thread t will cause the delay bound to be exceeded (as explained in §2), the schedule is not added to the set.

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

Depth-first search Even with a schedule bound, there are many possible orders in which to explore schedules. Maple's systematic mode only supports a depth-first search, as this allows a stack to be used to efficiently record which schedules still need to be explored. Since the stack is deeply ingrained in Maple's data structures and algorithms, we did not attempt to implement other search strategies. We note that the initial terminal schedule explored by iterative preemption bounding, iterative delay bounding and unbounded depth-first search is the same for all techniques (a non-preemptive round-robin schedule). We discuss the impact of depth-first search on our study further in §5.

Random scheduler Maple also includes a naive random scheduler mode, where, at each scheduling point, one enabled thread is randomly chosen from the set of enabled threads to execute a visible operation. Unlike schedule fuzzing, where randomisation is used to peturb the OS scheduler, this yields truly (pseudo-)random schedules because scheduling nondeterminism is fully controlled. No information is saved by the random scheduler for subsequent executions, so it is possible that the same schedule will be explored multiple times over many runs. This could be rectified by modifying Maple to record a history of schedules during random scheduling, but such a change would not be straightforward due to the way in which the tool is designed. As a result, with random scheduling the search cannot "complete", even for programs with a small number of schedules.

We include random scheduling as a baseline for non-systematic approaches, and to provide further insight on the complexity of the benchmarks.

Maple algorithm The default concurrency testing used by Maple (which we refer to as the *Maple algorithm*) is not systematic: it performs several profiling runs, recording patterns of inter-thread dependencies through shared-memory accesses [36]. 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). Although the focus of our study is on SCT techniques, we also compare with the Maple algorithm since it is readily available in the tool.

4. Benchmark Collection

We have collected a wide range of pthread benchmarks from previous work and other sources. Table 1 summarises the benchmark suites (with duplicates removed), indicating where it was necessary to skip benchmarks due to the difficulty of applying SCT, or otherwise. "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).

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	6	5 Chromium browser; 4 networking.
SPLASH-2	Parallel workloads	3	9 (see text)

Table 1: An overview of the benchmark suites used in the study.

4.1 Details of benchmark suites

Concurrency Bugs (CB) Benchmarks [35] Includes buggy versions of programs such as `aget` (a file downloader) and `pbzip2` (a file compressions 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 [26] A set of test cases for a work stealing queue, originally implemented for the Cilk multithreaded programming system [10] under Windows. The `WorkStealQueue` (`WSQ`) benchmark has been used frequently to evaluate concurrency testing tools [4, 23–27]. 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 [6] Examples used to evaluate the ESBMC tool [6], 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 [34] Used to evaluate the INSPECT concurrency testing tool. We skipped `swarm_isort64` that 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[3] 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 [7], which was the first bug found by CHESS that required that many preemptions.

The `ctrace` test case, obtained from the authors of [18], exposes a bug in the ctrace multithreaded debugging library.

PARSEC 2.0 Benchmarks [2] 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 §4.2). We configured the `streamcluster` benchmarks to use non-spinning synchronisation 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 [17] Consists of 15 tests that expose bugs in several applications. The 6 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. The others were skipped due to use of networking and multiple processes. Several tested the Chromium browser; the use of a GUI leads to nondeterminism that cannot be controlled or modelled by any SCT tools we know of. Some of the benchmarks were stress tests; we reduced the number of threads and other parameters as much as possible.

SPLASH-2 [33] Three of these benchmarks have been used in previous work [4, 29]. 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 reduce the values of input parameters, such as the number of particles in `barnes` and the size of the matrix in `lu`, so the tests complete quickly on our implementation, without exhausting memory. We discuss this further in §6.

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

4.2 Effort Required For SCT

We encountered a range of issues when trying to apply systematic concurrency testing to the benchmarks. These are general limitations of SCT, not of our method specifically, and all SCT tools that we know of would have similar issues.

Environment modelling System calls that interact with the environment, and hence can give nondeterministic results, must be modelled or fixed to return deterministic values. Similarly, 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 various sources of nondeterminism is a key issue in applying SCT to existing code bases. In contrast, non-systematic techniques (discussed in §7) are able to handle such nondeterminism.

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 webserver; the server module that we inspected had many dependencies on other parts of the server and directly called system functions, 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.

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 some SCT tools like CHESS, that require a repeatable function to test, 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.

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 suite, which is tested in the `streamcluster3` benchmark. Detecting certain types of out-of-bound memory accesses, such as accesses to the stack or data segments, is difficult, as information about the bounds of these regions is lost during compilation. Thus, our implementation had many false positives. However, a more serious issue was that 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 SCT 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 out-of-bound accesses in the `streamcluster3` benchmark and in `fsbench_bad` in the CS benchmarks. Out-of-bound accesses to synchronisation objects, such as mutexes, are still detected. This proved to be useful in `pbzip2` from the CS benchmarks.

Data races We found that 33 of the 52 benchmarks contained data races. There are many compelling arguments against the tolerance of data races [3], and technically, according to the C++11 standard, the existence of a data race in a C++ program means that the behaviour of the *entire* program is undefined. Nevertheless, in practice, programs that exhibit races are often compiled in predictable ways by standard compilers so that many data races are not regarded as bugs by software developers. A particular pattern we noticed was that data races often occur on flags used in ad-hoc busy-wait synchronisation, where one thread keeps reading a variable until the value changes. In principle the "benign" races could be rectified through the use of C++11 relaxed atomics, the "busy wait" use of data races could be formalised using C++11 acquire/release atomics, and synchronisation operations could be added to eliminate the buggy cases. However, telling the difference between benign and buggy data races is non-trivial in practice [18, 28]. 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. 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 usefulness as test cases.

5. Experimental Method

Our experimental evaluation aims to compare a straightforward depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB) and the use of a naive random scheduler (Rand). We also test the default Maple algorithm (MapleAlg). Bugs are deadlocks, crashes or assertion failures (including those that identify incorrect output). Each benchmark contains a concurrency bug and goes through the following phases:

Data Race Detection Phase When checking safety properties, it is sound to only consider scheduling points before each synchronisation operation, such as locking a mutex, as long as execution aborts with an error as soon as a data race is detected [26]. 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 num-

ber of data races in the benchmarks (see §4.2), which would make bug-finding trivial and arguably not meaningful.

As in previous work [36], we circumvent this issue by performing dynamic data race detection to identify a reasonable subset of load and store instructions that participate in data races. We treat these instructions as visible operations during SCT. 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 the IPB, IDB, DFS and Rand phases. We also tried detecting races during SCT, but this caused an additional slow-down of up to 8x, as Maple's race detector is not optimised for this scenario.

Thus SCT explores nondeterminism arising due to 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 of IPB, IDB, DFS and Rand, since the same information about data races is used by all of these techniques; 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.

Iterative Preemption Bounding (IPB) Phase We next perform SCT on the benchmark using iterative preemption bounding, with a schedule limit. By repeatedly executing the program, restarting after each execution, we first explore all terminal schedules that have zero preemptions, followed by all schedules that have one preemption, etc. until either the schedule limit 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). As explained in §3, 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.

We use a limit of 10,000 terminal schedules to enable a full experimental run over our large set of benchmarks to complete on a cluster within 24 hours. We chose to use a schedule limit instead of a time limit because there are many factors and potential optimisation opportunities that can affect the time needed for a benchmark to complete, and the cluster we have access to shares its machines with other jobs, making accurate time measurement difficult. On the other hand, the number of terminal 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.

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

Depth-First Search (DFS) Phase We perform a depth-first search, with no schedule bounding and a limit of 10,000 terminal schedules. This provides a point of comparison for schedule bounding.

Random scheduler (Rand) Phase We run each benchmark 10,000 times using Maple's naive random scheduler mode. This allows us to compare the systematic techniques against a straightforward non-systematic 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.

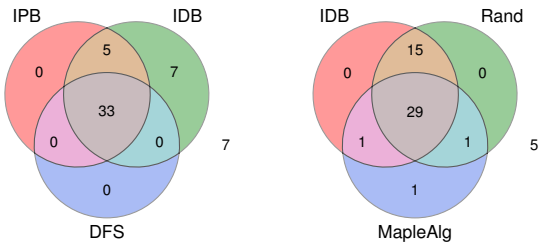
Notes on depth-first search and partial order reduction As discussed in §3, the SCT methods we evaluate are built on top of Maple's default depth-first search strategy. Although depth-first search is just one possible search strategy, and different strategies could give different results, we argue that 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. If a bug is revealed at bound c then, by enumerating 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) [11] is a commonly used technique in concurrency testing [9, 11, 24, 26]. We do not attempt to study the various POR techniques, to avoid an explosion of combinations of methods and because the relationship between POR and schedule bounding is complex and the topic of recent and ongoing work [5, 14, 24].

6. Experimental Results

Experimental platform 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 latest commit [4]. The benchmarks, scripts and the modified version of Maple used in our experiments can be obtained from `http://sites.google.com/site/sctbenchmarks`.

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



(a) Systematic techniques. (b) IDB vs. others.

Figure 2: Venn diagram showing number of benchmarks in which the bugs were found with the various techniques.

Overview of results The Venn diagrams in Figure 2 give a concise summary of the bug-finding ability of the techniques. When we say that a technique found x bugs, we mean that the technique found each bug in x benchmarks.

Figure 2a summarises the bugs found by the systematic techniques. IPB was superior to DFS, finding all 33 bugs found by DFS, plus an additional 5. IDB beat both DFS and IPB, finding all 38 bugs found by these techniques, plus an additional 7. The bugs in 7 benchmarks were missed by all systematic techniques, which we discuss below.

Figure 2b shows the bugs found by schedule bounding (IDB), a naive random scheduler (Rand) and the default Maple algorithm (MapleAlg). The bugs in 44 benchmarks were found by both IDB and Rand. IDB and Rand each found 1 additional, distinct, bug. Thus, these techniques performed similarly in terms of number of bugs found. We discuss this surprising result in detail below. MapleAlg found 31 bugs that were found by the other techniques, plus 1 additional bug. However, it missed 15 bugs that were found by the other techniques. There were 5 bugs missed by *all* techniques, but 3 of these are identical to benchmarks in which we did find bugs, except that they run a larger number of threads; the remaining 2 benchmarks, `radbench.bug1` and `misc.safestack`, are discussed below.

Detailed results The full set of experimental data gathered for our benchmarks is shown in Table 3. We use *schedules* to refer to *terminal schedules*, for brevity. As explained in §5, we focus on the number of schedules explored rather than time taken for analysis. The execution of a single benchmark during SCT varied between 1-7 seconds depending on the benchmark; there was negligible variance between runtimes for multiple executions of the same benchmark. The longest time taken to perform ten data race detection runs for a single benchmark was five minutes, but race detection was significantly faster in most cases. Race detection could be made more efficient using an optimised, state-of-the-art method. Because race analysis results are shared between all systematic techniques and Rand, the time for race analysis is not relevant when comparing these methods.

For each benchmark, *# 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, over all runs of the benchmark. The *# max scheduling points* column shows the maximum number of visible operations for which more than one thread was enabled, over all systematic testing. 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, 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 10,000 (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 does not apply to *all* schedules.

For the Rand results, the *# schedules* column is omitted, as it is always 10,000. Note that *# schedules to first bug* and *# buggy schedules* may contain duplicate schedules.

For the Maple algorithm, we report whether the bug was found (*found?*), 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 '-'.

Benchmark Properties The *# max enabled threads* and *# max scheduling points* columns from Table 3 can be used to estimate the total number of schedules and, perhaps, the complexity of the benchmark. With at most n enabled threads and at most k scheduling points, there are at most n^k terminal schedules. On the other hand, if most of the schedules are buggy (see the *% buggy* column in Table 3) then the number of schedules is not necessarily a good indication of bug complexity. For example, `CS.din_phil3_sat` has a relatively high number of schedules, but since 87% of them are buggy, 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

Property	**# benchmarks**
Bug found with DB $= 0$	14
Total terminal schedules $<$ 10,000	16
$>$ 50% of random schedules were buggy	19
Every random schedule was buggy	9

Table 2: Benchmarks where bug-finding is arguably trivial.

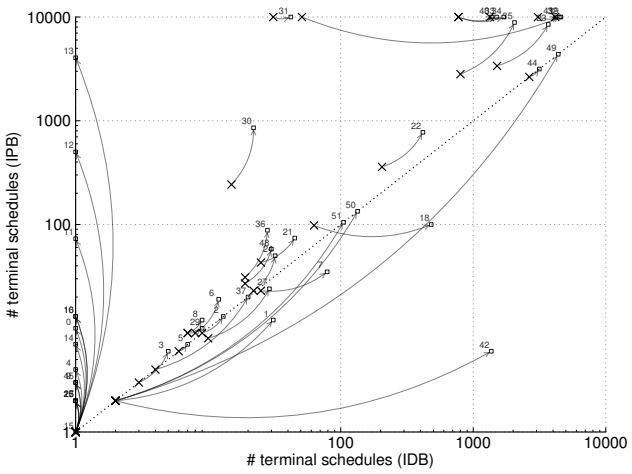


Figure 3: Shows # schedules to the first bug (cross) connected to the total # schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

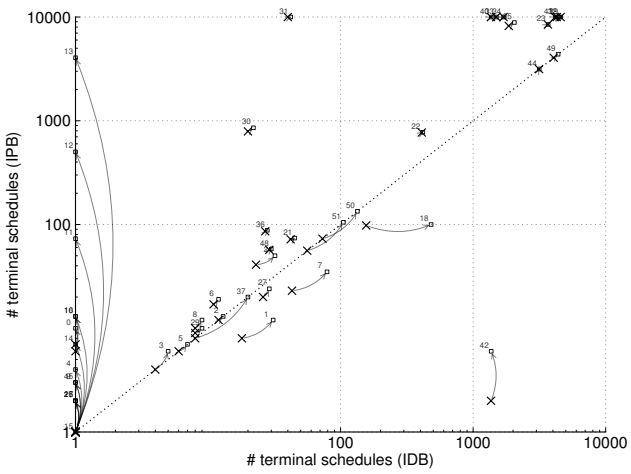


Figure 4: Shows total # non-buggy schedules (cross) connected to the total # schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

DFS results is problematic because DFS is biased towards exploring deep context switches.

Table 2 provides some further insight into the complexity of the benchmarks, using properties derived from Table 3. Bugs found with a delay bound of zero 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 10,000 terminal schedules (for DFS) will always be exhaustively explored by all systematic techniques, so the bug will always be found. Techniques can still be compared on how quickly they find the bugs. Bugs that were exposed more than 50% of the time when using the random scheduler could arguably be classified as "easy-to-find". Bugs that were exposed 100% of the time when using the random scheduler are almost certainly trivial to detect; indeed, Table 3 shows that all of these benchmarks were buggy for all schedules over *all* techniques, suggesting that these bugs are not even schedule-dependent.

In our view the relatively trivial nature of some of the bugs exhibited by our benchmarks has not been made clear in prior works that study these examples. 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.

IPB vs. IDB Figure 3 compares IPB and IDB by plotting data from the following columns in Table 3: *# schedules to first bug* (as a cross) and *# schedules* (as a square). Each benchmark, for which at least one technique found a bug, is depicted as a line connecting a cross and a square. Where the bug was not found by one of the techniques, this is indicated with a cross at 10,000 (the schedule limit discussed in §5). Each square is labelled with its benchmark *id* from Table 3. The cross indicates which technique was faster at finding the bug (with depth-first search as the underlying search strategy); 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 bound that found the bug. For example, when exploring benchmark 30 with IPB, the first buggy schedule was found after 243 schedules. 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 even if we were using an underlying search strategy other than depth-first search. Notice that a number of benchmarks appear at $(x,\ 10,000)$, with $x < 10,000$: this is where IPB failed to find a bug and IDB succeeded.

The bug-finding ability of the techniques in Figure 3 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 4 we consider the *worst-case* bug-finding ability. Each cross plots, for IDB and IPB, the total number of schedules within the bound exposing the bug that are *not* buggy. This corresponds to the difference between the *# schedules* and *# buggy schedules* columns presented in Table 3, 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 3.

Overall, IDB finds all bugs found by IPB, plus seven that were missed. In Figure 3, 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. In the worst case (Figure 4), 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 1366 schedules. Table 3 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 3) is the adversarial delay bounding example given in §2; 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 random scheduling Rand performed similarly to IDB in terms of bugs found (Figure 2b). Over *all* the benchmarks, it can be seen in Table 3 that Rand was nearly always similar to or much faster than IDB in terms of *# schedules to first bug*. This said, for any particular benchmark the *# schedules to first bug* value for Rand should be treated with caution due to the role of randomness in selecting the bug-inducing schedule.

We had not anticipated that a random scheduler would be so effective at finding bugs. A possible intuition for this is as follows. If a bug can be exposed with just one delay, say, then there is a single key preemption that exposes the bug. Any schedule where (a) the key preemption occurs, and (b) additional preemptions are irrelevant to the bug, 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. On the other hand, `radbench.bug2` (discussed below) requires three preemptions but was still found by Rand.

The CHESS benchmarks, used for evaluation in the introduction of preemption bounding [23], test several versions of a work stealing queue. Depth-first search fails for `chess.WSQ`, while IPB succeeds (as in prior work). However, Rand is also able to find the bug; prior work did not compare against a random scheduler in terms of bug finding ability. The remaining CHESS benchmarks are more complex (lock-free) versions of `chess.WSQ`, which were also used in prior work. IPB and DFS fail on these, while IDB and Rand are, again, both successful in finding the bugs. Rand found the bugs in fewer terminal schedules than IDB and IPB for all the CHESS benchmarks.

The bug in the `parsec.ferret` benchmark 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 3, only one buggy schedule was found; thus, the delay must occur at a specific visible operation.

The bug in `radbench.bug4` is missed by IDB, but found by Rand. The bug is caused by a shared mutex being lazily initialised by two threads at once, without synchronisation. This can lead to a double-unlock or similar error. From Table 3, it can be seen that this bug requires more than one delay. The benchmark has a relatively large number of scheduling points, such that the number of schedules with at most two delays exceeds the schedule limit.

There are several benchmarks for which the percentage of buggy schedules encountered during DFS (see *% buggy* in Table 3) is very similar to the percentage of buggy schedules observed for Rand. For example, 4% vs. 5% in `CB.stringbuffer-jdk1.4`, and 14% vs. 10% in `CS.account_bad`: However, there are counter-examples: `CS.carter01_bad`: 2% vs. 48%; `CS.deadlock01_bad`: 6% vs. 40%. Since the majority of benchmarks cannot be explored exhaustively and DFS is biased towards exploring deep context switches, it is impossible to estimate the percentage of buggy schedules for most of the benchmarks.

Comparison with the default Maple algorithm As shown in Figure 2b, MapleAlg missed 15 bugs that were found by the other techniques, and found 32 bugs, including 1 that was missed by the others. MapleAlg is impressive considering the low number of schedules it explores. For example, all other techniques missed the bug in `radbench.bug5` after 10,000 terminal schedules. In contrast, MapleAlg found it after just 14 schedules. MapleAlg attempts to force certain patterns of inter-thread accesses (or *interleaving idioms*) that might lead to concurrency bugs. This allows it to expose

many bugs quickly. It is possible that the bugs it misses require interleaving idioms that are not included in MapleAlg.

Discussion No technique found the bugs in 19, 20 and 28. However, these bugs can be exposed using a lower number of threads (as shown by the other versions of these benchmarks), so these results are arguably less useful.

The schedule bounding results reveal that the bug in `radbench.bug2` requires at least three delays or preemptions. The benchmark was modified to use just two threads in total and IPB and IDB explored the same schedules. This matches largest number of preemptions required to expose a bug found in previous work [7]. However, the `misc.safestack` benchmark reportedly requires five preemptions and three threads in order for the bug to manifest. We reproduced the bug using Relacy[5], a weak memory data race detector that performs either systematic or random search for C++ programs that use C++ atomics.

The bug in `radbench.bug1` requires a thread to be preempted after destroying a hash table and a second thread to access the hash table, causing a crash. From the description, the bug *may* only require one delay, but it is likely that the large number of scheduling points is what pushes this bug out of reach of all the techniques tested.

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 are accurate.

7. Related Work

To our knowledge, ours is the first independent empirical study to compare schedule bounding techniques. Background and related work on systematic schedule bounding was discussed in §2. We now discuss other relevant approaches to reducing thread schedules in order to find bugs.

Partial-order reduction (POR) [11] reduces the number of schedules that need to be explored without missing errors. It relies on the fact that executions are a partial-order of operations, and explores only *one* schedule of each partial-order. Dynamic POR [9] computes persistent sets [11] during systematic search; as *dependencies* between operations are detected, additional schedules are considered. Happens-before graph caching [24, 26] is similar to state-hashing [13], except the partial-order of synchronisation operations is used as an approximation of the state, resulting in a reduction similar to sleep-sets [11]. The combination of dynamic POR and schedule bounding is the topic of recent research [5, 14, 24].

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

The PCT algorithm [4] executes programs using a randomised priority-based scheduler. A bounded number of priority change points are inserted at random depths during the execution, forcing certain thread interleavings. Crucially, the depths of the change points are chosen *uniformly* over the length of the execution, unlike a traditional random scheduler that makes a random choice at every execution step. This allows bugs to be detected with a probability that is inverse exponential in the number of change points c. It allows bugs with $c + 1$ ordering constraints to be found; this number is shown empirically to be small for many interesting concurrency bugs. The parallel PCT algorithm [27] improves on this work by allowing parallel execution of many threads, as opposed to always serialising execution.

In addition to PCT, there has been a wide-range of work on other non-systematic approaches, including [29, 30, 32, 36]. Like parallel PCT, these approaches are appealing as they allow parallel execution of many threads and can handle complex synchronisation and nondeterminism.

Our study has briefly touched on dynamic race detection issues. A discussion of this wide area is out of scope here, but we refer to [8] for the state-of-the-art.

8. Conclusions and Future Work

We have presented the first independent empirical study on schedule bounding techniques for systematic concurrency testing. Our most surprising finding is that a naive *random* scheduler performs at least as well the more sophisticated iterative schedule bounding approach, when trying to expose bugs within 10,000 terminal schedules. This may indicate that the benchmarks typically used to evaluate concurrency testing tools are not adequate, as they contain bugs that can be found fairly easily through random search. On the other hand, we have proposed an intuition for why bugs that can be exposed with few preemptions may be exposed by a high percentage of schedules, and thus are amenable to exposure through randomisation.

Our findings confirm results in previous work: that schedule bounding is superior to depth-first search; many, but not all, bugs can be found using a small schedule bound; and delay bounding beats preemption bounding.

In future work we plan to expand SCTBench to conduct larger studies, and to study additional methods, such as various partial-order reduction techniques that reduce the number of schedules explored during systematic testing, as well as non-systematic approaches to concurrency testing.

Acknowledgements We are grateful to the PPoPP reviewers for their useful comments, and especially to reviewer #1 who suggested that we try random scheduling, which led to interesting results. We are also grateful for feedback on this work from Ethel Bardsley, Nathan Chong, Pantazis Deligiannis, Tony Field, Jeroen Ketema and Shaz Qadeer.

id	name				IPB					IDB					DFS					Rand		MapleAlg		
		# threads	# max enabled threads	# max scheduling points	bound	# schedules to first bug	# schedules	# new schedules	# buggy schedules	bound	# schedules to first bug	# schedules	# new schedules	# buggy schedules	# schedules to first bug	# schedules	# buggy schedules	% buggy	# schedules to first bug	# buggy schedules	found?	# schedules	total time (seconds)
0	CB.aget-bug2	4	3	23	0	1	10	10	4	0	1	1	1	1	1	L	6698	*66%	2	4874	✓	17	37
1	CB.pbzip2-0.9.4	4	4	38	0	2	12	12	4	1	2	31	30	13	2	L	6245	*62%	1	4263	✓	4	20
2	CB.stringbuffer-jdk1.4	2	2	6	2	9	13	8	1	2	9	13	8	1	7	24	1	4%	5	577	✓	9	7
3	CS.account_bad	4	3	5	0	3	6	6	2	1	3	5	4	1	3	28	4	14%	3	1089	✓	20	12
4	CS.arithmetic_prog_bad	3	2	20	0	1	4	4	4	0	1	1	1	1	1	L	L	*100%	1	L	✓	1	1
5	CS.bluetooth_driver_bad	2	2	9	1	6	7	6	1	1	6	7	6	1	36	177	10	5%	45	562	✗	11	7
6	CS.carter01_bad	5	3	14	1	9	19	16	2	1	8	12	11	1	8	1708	49	2%	3	4898	✓	6	5
7	CS.circular_buffer_bad	3	2	26	1	23	35	32	12	2	25	79	56	36	20	3991	2043	51%	3	9013	✗	17	12
8	CS.deadlock01_bad	3	2	8	1	9	12	9	2	1	7	9	8	1	10	46	3	6%	1	4095	✗	7	5
9	CS.din_phil2_sat	3	2	17	0	1	3	3	3	0	1	1	1	1	1	5336	4686	87%	1	9700	✓	1	1
10	CS.din_phil3_sat	4	3	25	0	1	13	13	13	0	1	1	1	1	1	L	8710	*87%	1	9270	✓	1	1
11	CS.din_phil4_sat	5	4	36	0	1	73	73	73	0	1	1	1	1	1	L	9353	*93%	1	8756	✓	1	1
12	CS.din_phil5_sat	6	5	39	0	1	501	501	501	0	1	1	1	1	1	L	L	*100%	1	L	✓	1	1
13	CS.din_phil6_sat	7	6	49	0	1	4051	4051	4051	0	1	1	1	1	1	L	L	*100%	1	L	✓	1	1
14	CS.din_phil7_sat	8	7	10	0	1	7	7	7	0	1	1	1	1	1	924	924	100%	1	L	✓	1	1
15	CS.fsbench_bad	28	27	155	0	1	1	1	1	0	1	1	1	1	1	L	L	*100%	1	L	✓	1	1
16	CS.lazy01_bad	4	3	7	0	1	13	13	6	0	1	1	1	1	1	118	81	68%	1	6018	✓	1	1
17	CS.phase01_bad	3	2	6	0	1	2	2	2	0	1	1	1	1	1	17	17	100%	1	L	✓	1	1
18	CS.queue_bad	3	2	61	1	98	100	97	2	2	63	482	420	326	43	L	6405	*64%	1	9996	✓	2	1
19	CS.reorder_10_bad	11	10	38	0	✗	L	L	0	4	✗	L	3217	0	✗	L	0	*0%	✗	0	✗	11	7
20	CS.reorder_20_bad	21	20	87	0	✗	L	L	0	3	✗	L	6916	0	✗	L	0	*0%	✗	0	✗	11	7
21	CS.reorder_3_bad	4	3	10	1	43	74	61	2	2	25	45	35	3	126	2494	23	<1%	25	270	✗	10	7
22	CS.reorder_4_bad	5	4	14	1	359	774	701	3	3	205	417	330	7	6409	L	4	*<1%	9	63	✗	11	8
23	CS.reorder_5_bad	6	5	18	1	3378	8483	7982	4	4	1513	3681	2843	15	✗	L	0	*0%	123	22	✗	11	7
24	CS.stack_bad	3	2	31	1	23	50	47	9	1	22	32	31	9	22	L	512	*5%	1	5877	✗	10	8
25	CS.sync01_bad	3	2	2	0	1	2	2	2	0	1	1	1	1	1	6	6	100%	1	L	✓	1	1
26	CS.sync02_bad	3	2	9	0	1	2	2	2	0	1	1	1	1	1	88	88	100%	1	L	✓	1	1
27	CS.token_ring_bad	5	4	8	0	8	24	24	4	2	10	29	22	3	8	280	57	20%	6	1103	✓	5	4
28	CS.twostage_100_bad	101	100	792	0	✗	L	L	0	2	✗	L	9304	0	✗	L	0	*0%	✗	0	✗	11	9
29	CS.twostage_bad	3	2	8	1	9	10	7	1	1	7	9	8	1	13	87	3	3%	5	837	✓	8	5
30	CS.wronglock_3_bad	5	4	22	1	243	856	783	66	1	15	22	21	2	3233	L	94	*<1%	8	3010	✓	6	4
31	CS.wronglock_bad	9	8	46	0	✗	L	L	0	1	31	42	41	2	✗	L	0	*0%	5	3065	✓	6	4
32	chess.IWSQ	3	3	120	1	✗	L	9997	0	2	3077	4466	4351	192	✗	L	0	*0%	1496	15	✗	7	-
33	chess.IWSQWS	3	3	1497	1	✗	L	9997	0	1	773	1498	1497	1	✗	L	0	*0%	2	646	✗	9	-
34	chess.SWSQ	3	3	1697	1	✗	L	9997	0	1	773	1698	1697	1	✗	L	0	*0%	140	85	✗	7	-
35	chess.WSQ	3	3	90	2	2814	8852	8626	640	2	801	2048	1974	192	✗	L	0	*0%	355	8	✗	12	12
36	inspect.qsort_mt	3	3	33	1	31	88	84	2	1	19	28	27	1	✗	L	0	*0%	132	108	✗	142	102
37	misc.ctrace-test	3	2	19	1	4	20	19	12	1	4	20	19	12	4	20	12	60%	2	2641	✓	1	1
38	misc.safestack	4	3	114	1	✗	L	9987	0	3	✗	L	5958	0	✗	L	0	*0%	✗	0	✗	23	16
39	parsec.ferret	11	11	24453	0	✗	L	L	0	1	51	4575	4574	1	✗	L	0	*0%	✗	0	✓	27	205
40	parsec.streamcluster	5	2	1373	1	✗	L	9994	0	1	1336	1372	1371	10	✗	L	0	*0%	2	7122	✓	1	2
41	parsec.streamcluster2	7	3	4177	0	✗	L	L	0	1	4155	4177	4176	20	✗	L	0	*0%	31	347	✗	24	149
42	parsec.streamcluster3	5	2	1373	0	2	6	6	4	1	2	1369	1368	4	2	L	6078	*60%	4	3435	✓	1	1
43	radbench.bug1	4	3	14214	1	✗	L	9962	0	1	✗	L	9999	0	✗	L	0	*0%	✗	0	✗	629	8797
44	radbench.bug2	2	2	41	3	2647	3154	2808	8	3	2647	3154	2808	8	✗	L	0	*0%	2267	5	✗	220	804
45	radbench.bug3	3	2	239	0	1	3	3	3	0	1	1	1	1	1	L	L	*100%	1	L	✓	32	96
46	radbench.bug4	3	3	209	2	✗	L	9658	0	2	✗	L	9852	0	✗	L	0	*0%	377	9	✓	16	23
47	radbench.bug5	7	3	856	1	✗	L	9936	0	2	✗	L	9210	0	✗	L	0	*0%	✗	0	✓	14	18
48	radbench.bug6	3	3	29	1	27	58	55	1	1	19	30	29	1	✗	L	0	*0%	12	1306	✗	34	39
49	splash2.barnes	2	2	4408	1	2	4378	4377	326	1	2	4378	4377	326	2	L	2484	*24%	3	4982	✓	1	1
50	splash2.fft	2	2	136	1	2	134	133	61	1	2	134	133	61	2	L	7429	*74%	2	6214	✓	2	2
51	splash2.lu	2	2	114	1	2	105	104	49	1	2	105	104	49	2	L	4993	*49%	1	9741	✓	2	3

Table 3: Experimental results for systematic concurrency testing using iterative preemption bounding (IPB), iterative delay bounding (IDB) and unbounded depth-first search (DFS), and non-systematic testing with a naive random scheduler (Rand) and using the Maple algorithm (MapleAlg). Entries marked 'L' indicate 10,000, our schedule limit. A '✗' indicates that no bug was found. In the MapleAlg results, '-' indicates that the Maple tool timed out after 24 hours. A percentage prefixed with '*' does not apply to *all* schedules, only those that were explored via DFS before the schedule limit was reached.

References

[1] A. Bessey et al. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2): 66–75, 2010.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[3] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, pages 1–6, 2011.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[5] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *OOPSLA*, pages 833–848, 2013.

[6] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, pages 331–340, 2011.

[7] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

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

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

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, 1996.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, 1997.

[13] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *PSTV*, pages 339–344, 1987.

[14] G. J. Holzmann and M. Florian. Model checking with bounded context switching. *Formal Asp. Comput.*, 23(3):365–389, 2011.

[15] J. Huang and C. Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *SAS*, pages 163–179, 2011.

[16] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE*, FSE '10, pages 57–66, 2010.

[17] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. RADBench: a concurrency bug benchmark suite. In *HotPar*, pages 1–6, 2011.

[18] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[19] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *PADTAD*, pages 48–58, 2010.

[20] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[21] B. Lewis and D. J. Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, 1998.

[22] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[23] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[24] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007.

[25] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI*, pages 362–371, 2008.

[26] M. Musuvathi et al. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[27] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, pages 543–554, 2012.

[28] S. Narayanasamy et al. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, 2007.

[29] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

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

[31] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[32] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE*, pages 221–230, 2011.

[33] S. C. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[34] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah, 2008.

[35] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336, 2009.

[36] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA*, pages 485–502, 2012.